The eTIE Language Extension

Agile for Domain-Specific Accelerator Design

He XIAO

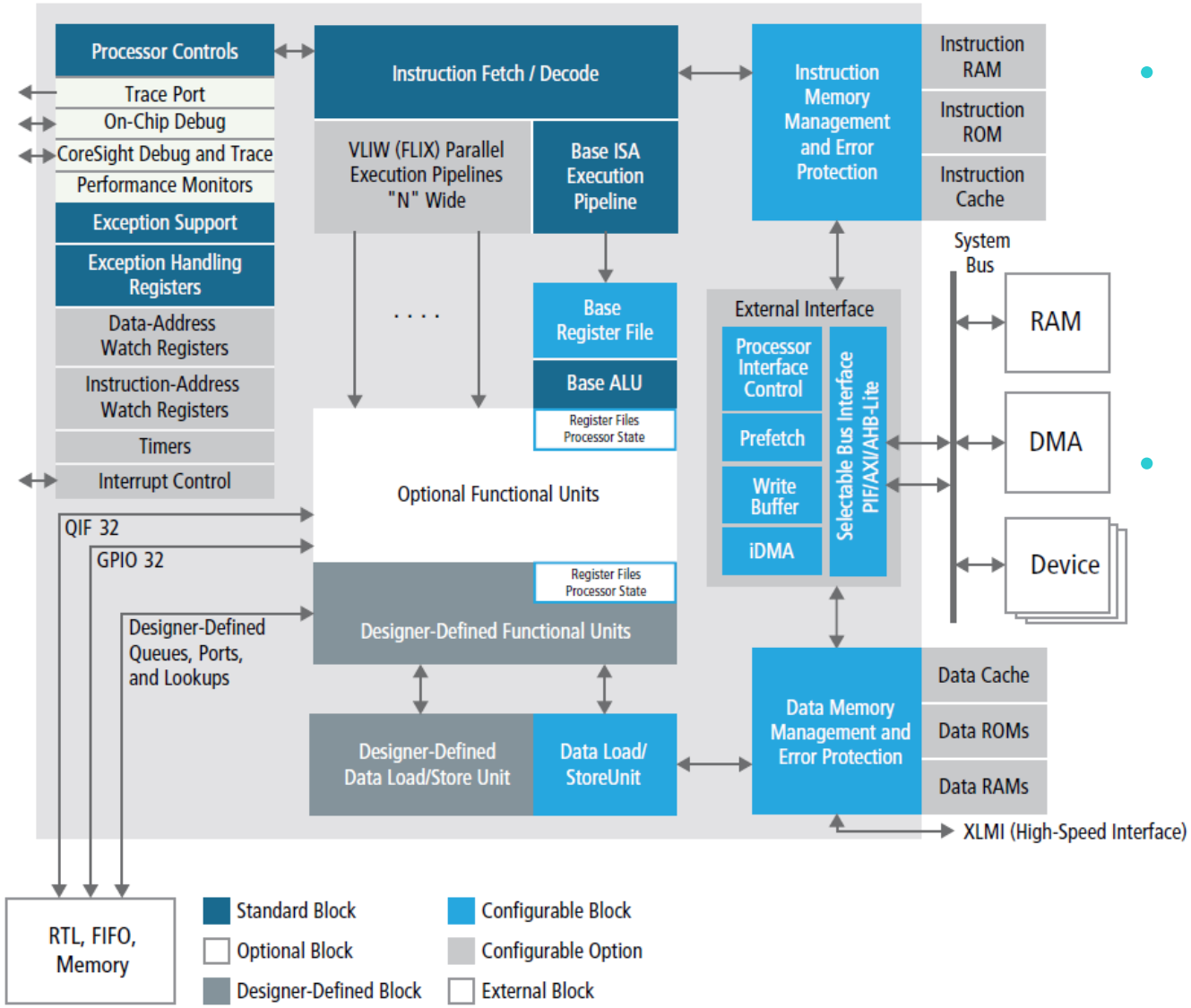
Feb. 28th 2021

cadence®

Tensilica DSP and TIE Overview

Tensilica Xtensa DSP Architecture

From a RISC to a VLIW, Vector Machine



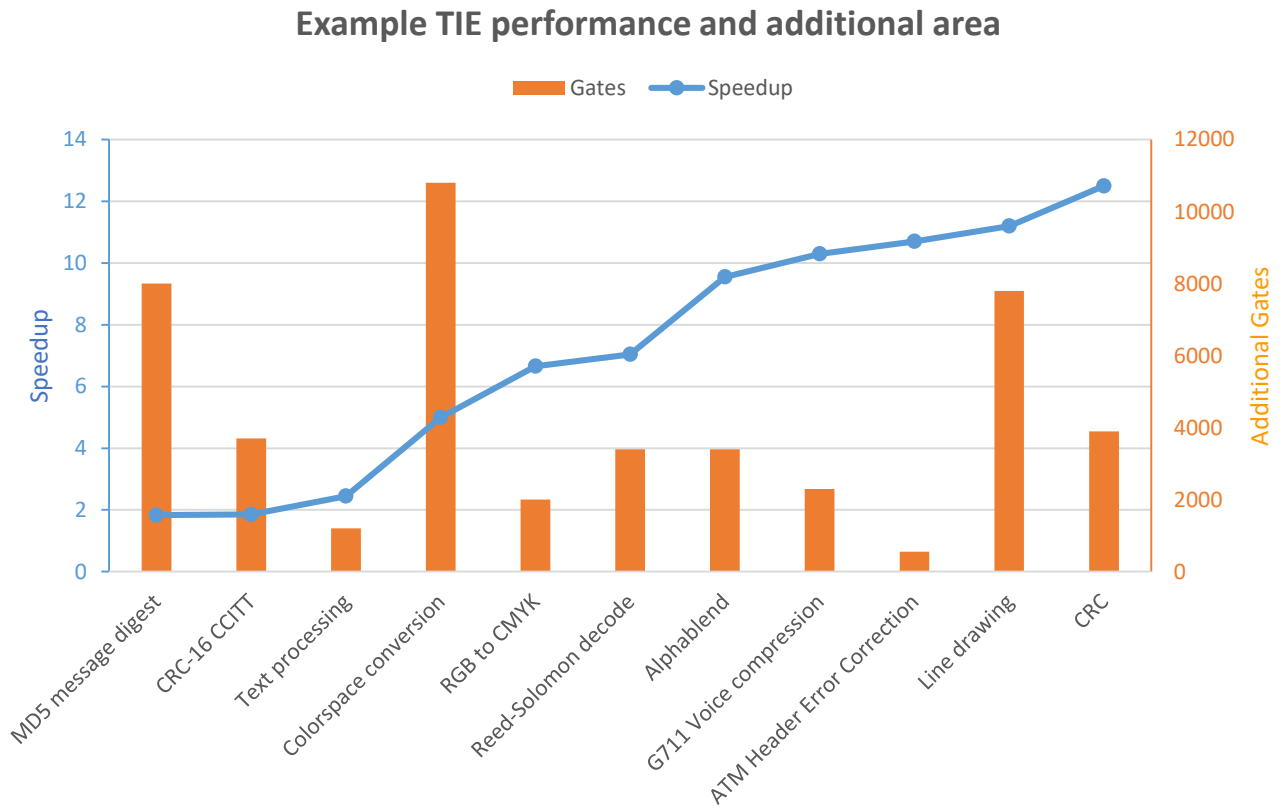
- Cadence Xtensa Processor is configurable and extensible processor:
  - Add performance, flexibility and longevity
  - Automatic hardware and software generation
    - C compiler, ISS, debugger
    - Optimized RTL and verification framework
- Using a simple Verilog-like language (**TIE**) you can define…
  - Input/output queues and ports
  - Local scratchpad memories
  - Fast lookup tables
  - Simple single-cycle instructions
  - Complex multi-cycle instructions
  - SIMD instructions for vectorization
  - VLIW-like operations for exploiting instruction level parallelism

What is TIE?

- TIE is a technology to extend functionality of Xtensa Processors via best-in-class automation
- TIE stands for Tensilica Instruction Extension
- Includes
  - TIE Language – A Verilog-based language that allows users to define processor extensions
  - TIE Compiler and automated methodology to convert TIE description in to
    - Software tools (C/C++ compiler, debugger, instruction set simulator)
    - Hardware (RTL)
- For more information beyond this session, refer to..
  - TIE Language Reference Manual
  - TIE Language User's Guide
  - Online Training Module (TIE)

Advantage of TIE

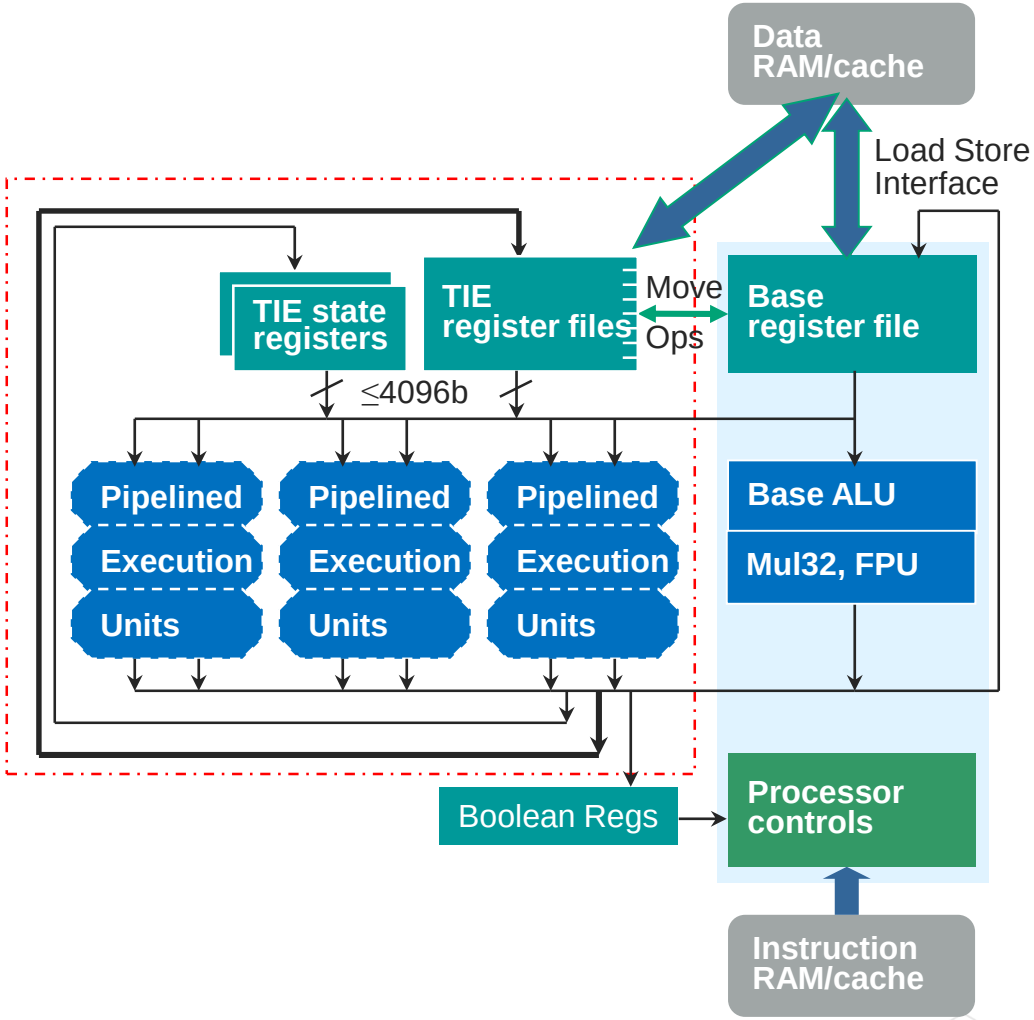
- Extensions tailored to application requirements and PPA targets
- Programmability
  - State machine becomes S/W
  - Datapath (i.e., compute) similar to RTL
- Verification
  - User only verifies datapath
  - Use S/W for verification
- Immediate feedback – compile and use extensions in a matter of minutes



Types of Extensions

Fusion, Vector (SIMD), VLIW Machines

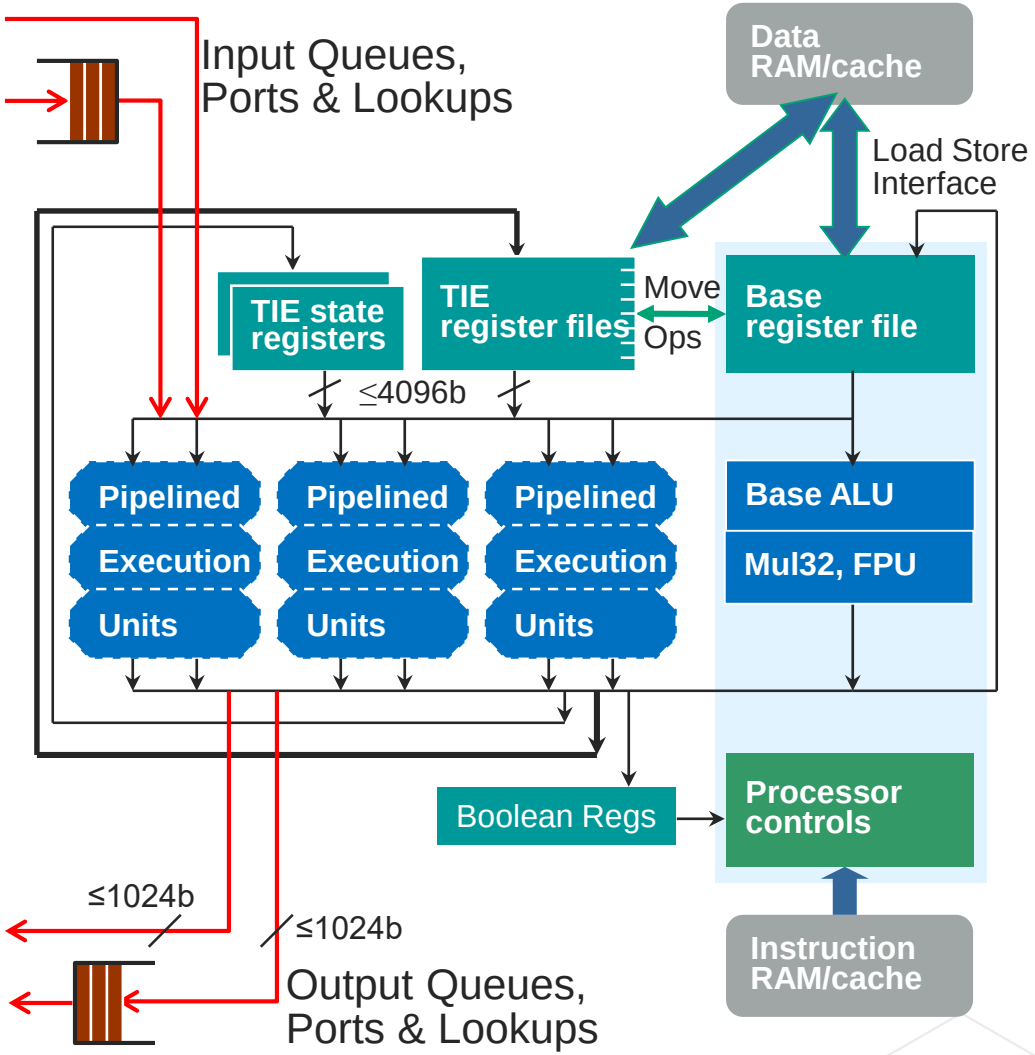
- Custom execution units integrated in to the processor pipeline
  - Single-cycle or pipelined multi-cycle
  - Fusion or SIMD-type computation
- Support for SIMD and Wide data by creating register banks up to 4096 bits wide
  - Single-instance or multi-entry
- Custom load and store operations
  - Move up to 512 bits per access
  - Custom addressing modes
- VLIW-like operations for exploiting instruction level parallelism



Types of Extensions

SOC Connectivity

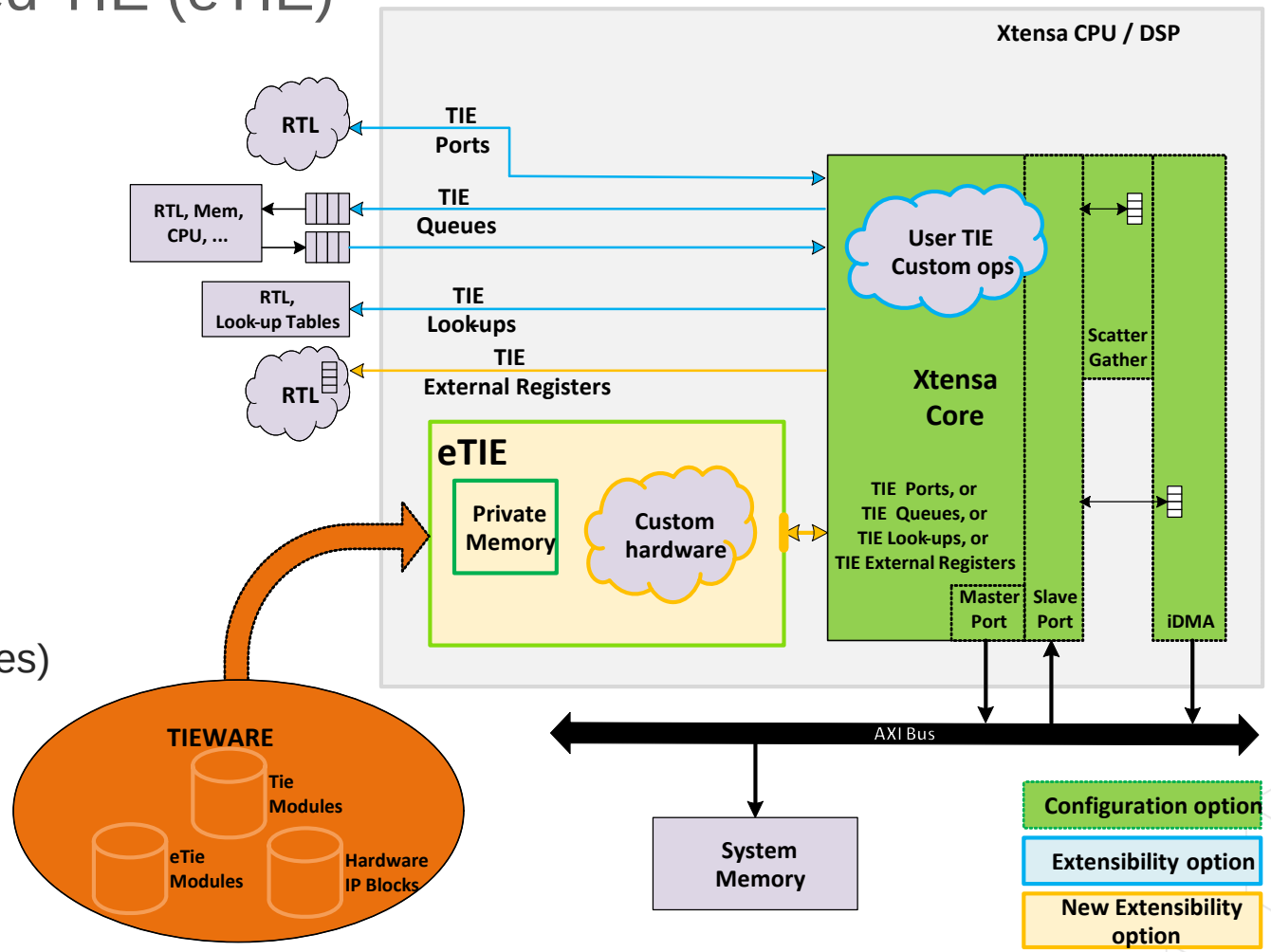
- Low latency, high bandwidth data transfers
  - Move data directly to/from custom execution units
  - Bypass load/store unit (AXI)
- Choice of interfaces
  - Queues for FIFO-type handshaking
  - Lookups for atomic request-response transactions
  - General purpose I/O for read/write
- Up to 1024 interfaces per core
  - Each up to 1024 bits wide



Types of Extensions

Hardware Coprocessors using extended TIE (eTIE)

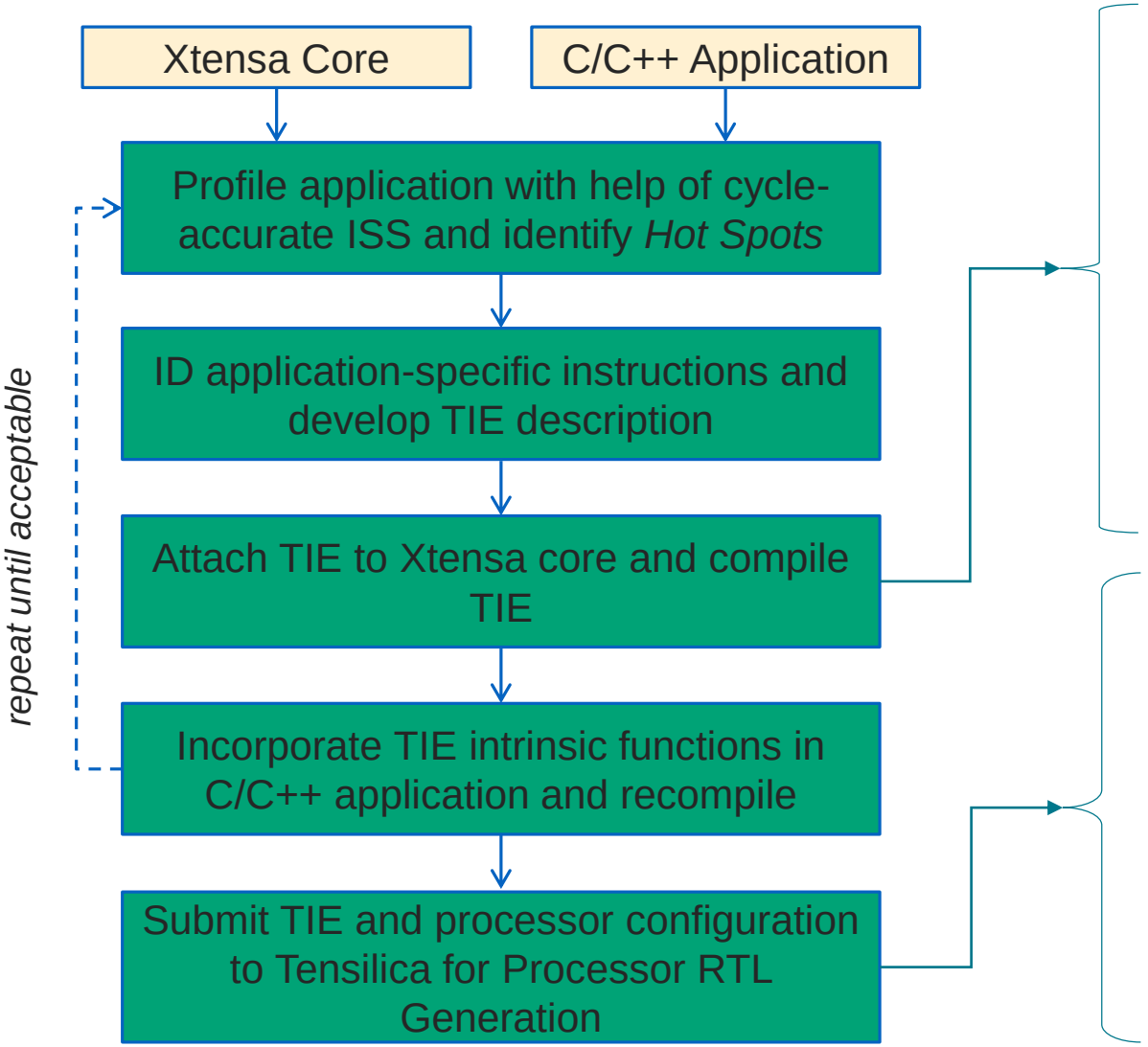
- Accelerators that execute autonomously of the processor pipeline
  - Variable and state-dependent execution
  - Decision logic implemented in hardware
  - Signal error or exception conditions via interrupt
- Data from a variety of sources
  - Direct from processor via TIE interfaces
  - Direct from private memory or internal flops
  - Generated during execution (e.g., FFT twiddle tables)
- Close connection between hardware description and software tools
  - eTIE hardware description modeled in ISS
  - Compiler visibility of eTIE resource scheduling
  - Debugger awareness of eTIE hardware resources



eTIE deep-dive session: https://wiki.ith.intel.com/display/RDSE/Tensilica

TIE Development Walkthrough

Typical TIE Development Flow

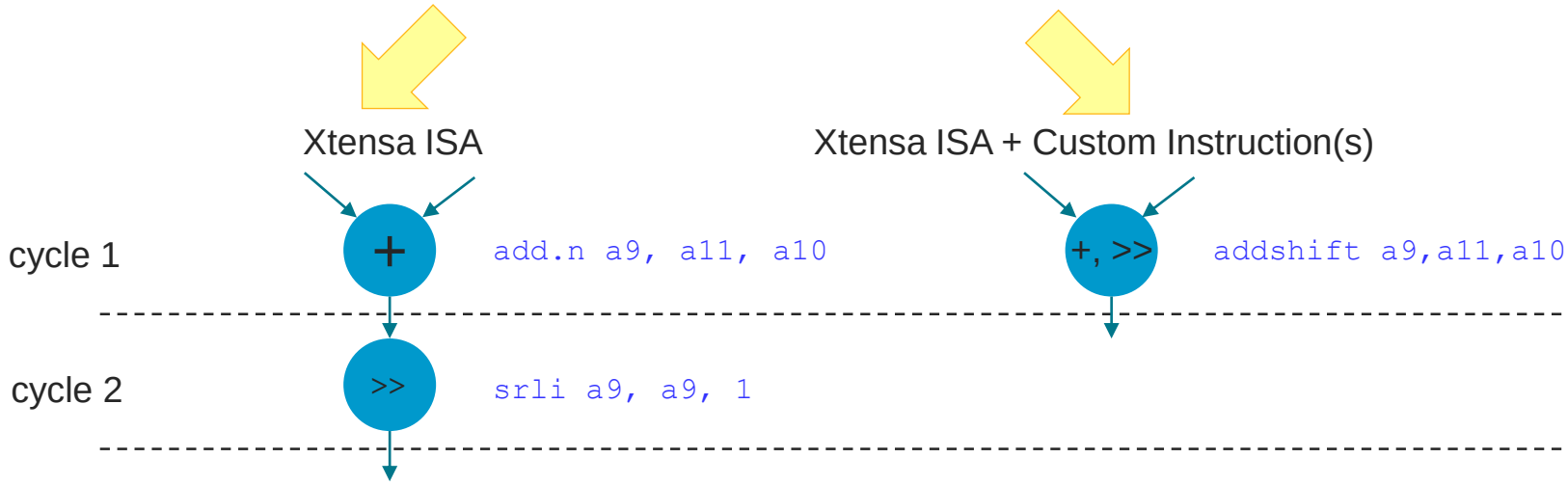


- "Local" TDK Flow
  - TC adds the new instruction to Xtensa core
    - Use Xtensa Xplorer or command-line
    - Typically takes a few minutes
  - Produces a new set of Xtensa software tools
  - Instructions can be used right away in application
  - Also produced → RTL of TIE extensions
- XPG Flow
  - Produces RTL of processor w/ TIE extensions
  - Processor build is done by automated Tensilica server (takes several hours)
  - Submission is done from within Xtensa Xplorer IDE

Fusing operations to improve performance

C Code

```
unsigned int a[N], b[N], c[N], i ;

for (i = 0; i < N; i++) {
     c[i] = (a[i] + b[i]) / 2 ;
}
```



The basics of a new TIE operation, *addshift*

avg.tie

```
operation addshift {out AR avg, in AR A, in AR B} {} {
  assign avg = (A + B) >> 1 ;
}
```

- User specifies three elements of a TIE operation
  - Unique name
  - List of operands – input(s) and output(s)
  - Computation performed on input(s) to produce output(s)
- TIE Compiler handles everything else!
  - New opcode *addshift* is added to ISA
  - Operation RTL is generated from TIE description
  - Operation RTL is integrated in to processor pipeline
    - Input and output operands wired up to respective architectural resources (e.g., register files)
    - Interlock (i.e., stall) and forwarding/bypass logic wired up
  - Software tools (e.g., compiler, simulator, etc.) updated to handle *addshift* opcode

Operands of TIE operation

avg.tie

```
operation addshift {/* explicit */out AR avg, in AR A, in AR B} {/* implicit */} {
  assign avg = (A + B) >> 1 ;
}
```

- Explicit
  - Require encoding bits within instruction word (e.g., register file accesses or immediates)
  - Readable (or writable) via Assembly or C code
  - Number of operands is only limited only by available encoding space
- Implicit
  - Singleton resources that don't require encoding bits (e.g., memory or TIE interfaces)
  - Not readable (or writable) directly via Assembly or C code
  - Number of operands is theoretically unlimited
- Operands can be inputs (i.e., read), outputs (i.e., write), or inout (read and write)

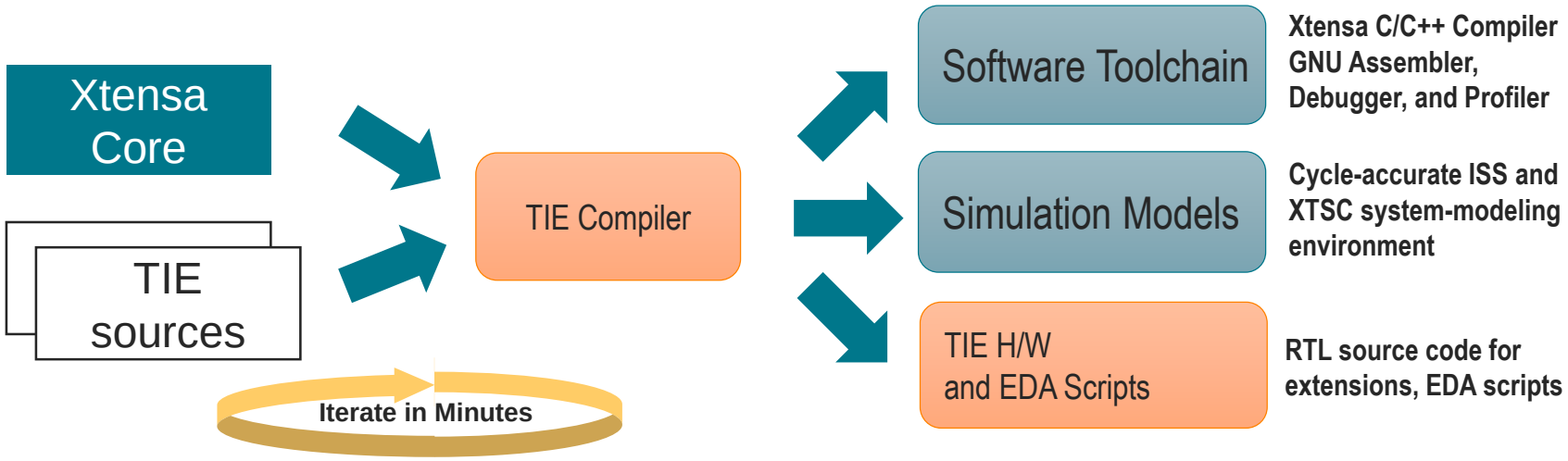
Computation in TIE operation

avg.tie

```
operation addshift {/* explicit */out AR avg, in AR A, in AR B} {/* implicit */} {
   assign avg = (A + B) >> 1 ;
}
```

- Behavioral description of computation datapath
- Syntax is similar to Verilog
  - Computation is described use one or more continuous assignments
  - Temporary variables are represented using TIE wire
  - Sequential logic is represented using TIEflop
  - Result of expression in assignment statement is either an output operand or temporary wire
  - Rich set of operators to form expressions
- Computation can be single-cycle (default) or multi-cycle
- Multi-cycle computation is fully-pipelined
  - Allows back-to-back instructions for maximum throughput

Compiling TIE



- Compilation is “local” - i.e., with local installation of TIE Compiler
  - CL (command-line) or invoke within Xtensa Xplorer
  - “Attach” TIE sources to Xtensa core and compile!
  - Process is quick – a few minutes
- Compilation produces a new set of Xtensa software tools
- Can use instruction extensions in software right away

Usage in Software

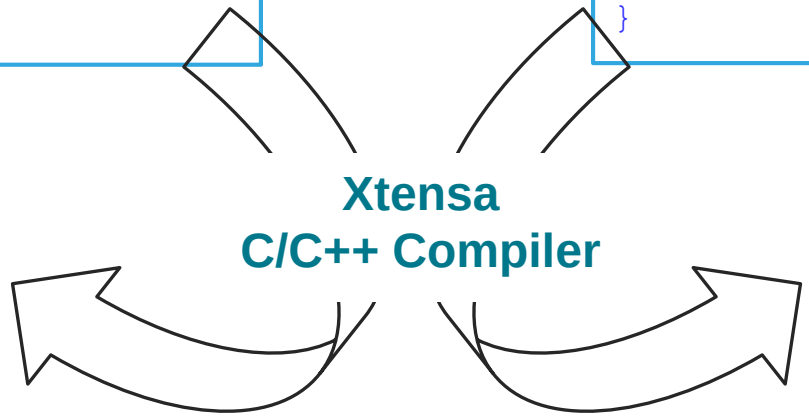
Regular C/C++ code

```
unsigned int a[N], b[N], c[N], i;

for (i=0; i<N; i++) {
   c[i] = (a[i] + b[i]) / 2;
}
```

C/C++ code incorporating Instruction Extensions

```
unsigned int a[N], b[N], c[N], i;

for (i=0; i<N; i++) {
   c[i] = addshift(a[i], b[i]);
}
```

C intrinsic of *addshift* instruction




```
…
l32i.n a11, a11, 0
l32i.n  a9, a9, 0
add.n  a9, a9, a11
srli  a9, a9, 1
s32i.n  a9, a10, 0
…
```

```
…
l32i.n   a9, a9, 0
l32i.n   a11, a11, 0
addshift a9, a9, a11
s32i.n   a9, a10, 0
…
…
```

addshift opcode

- TIE-specific extensions to ANSI C/C++ standard
  - TIE Intrinsics → similar to C intrinsics
  - TIE ctype → user-defined data types
- Write in C or C++. No Assembly programming needed.

TIE Intrinsic

- Similar to C intrinsic used for compiler-inferred architecture-specific optimizations on C/C++ code
- Xtensa C/C++ compiler responsible for
  - Type checking of input/output arguments
  - Data movement, if necessary
  - Translation into TIE operation(s)
- Implementation is inlined in generated Assembly

```
unsigned int a[N], b[N], c[N], i;

for (i=0; i<N; i++) {
   c[i] = addshift(a[i], b[i]);
}
```

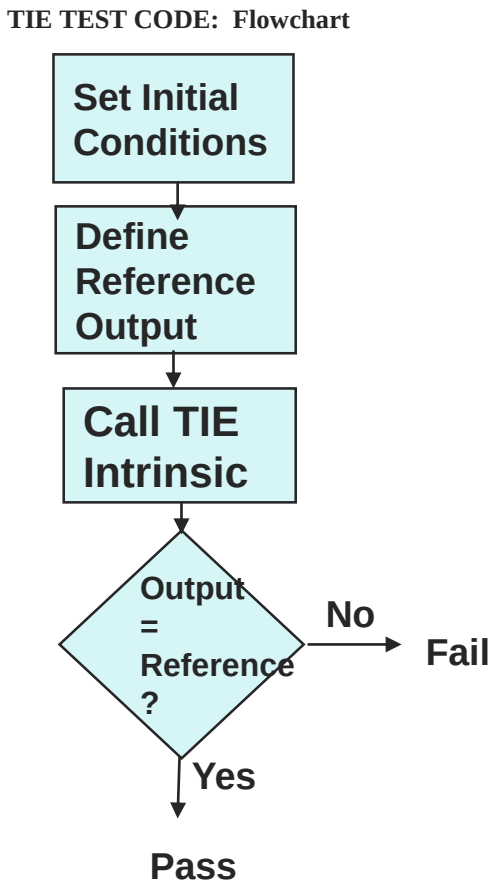
```
…
l32i.n   a9, a9, 0
l32i.n   a11, a11, 0
addshift a9, a9, a11
s32i.n   a9, a10, 0
…
…
```

Options for Simulating C/C++ code with TIE

- Xtensa Instruction Set Simulator (ISS) automatically updated by TC
  - Simulation model derived from original TIE description
  - Cycle-accurate and functional modes
- Standalone ISS sufficient for extensions “local” to the processor
- Options for extensions that use TIE interfaces to external h/w
  - Standalone ISS with data files
  - XTSC (SystemC) modeling environment where external devices can be modeled
    - ISS is embedded within XTSC
  - Co-simulation (RTL plus XTSC)
- Native (i.e., x86) compilation and simulation using cstubs library
  - ANSI C/C++ routines that are functionally equivalent to TIE instructions

TIE Verification

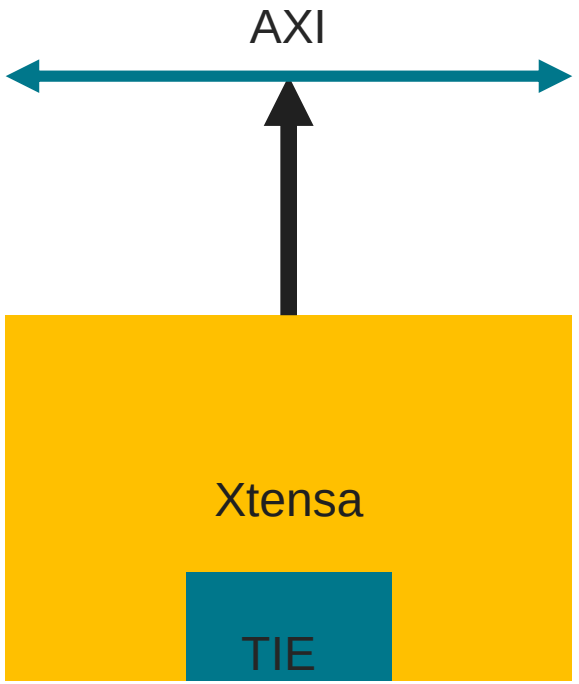
- TIE compiler ensures that the RTL and ISS match TIE description
- However… TIE developer will
  - Verify that TIE description does what you want it to do
  - Verify that the processor (including TIE) behaves properly after synthesis
- Verification strategy
  - Start with unit-level verification using ISS (or XTSC)
    - Verify each TIE operation
    - Use C reference or test vectors
  - Run subset of tests on RTL simulator for sanity
  - Formal as-needed



The eTIE Language Extention

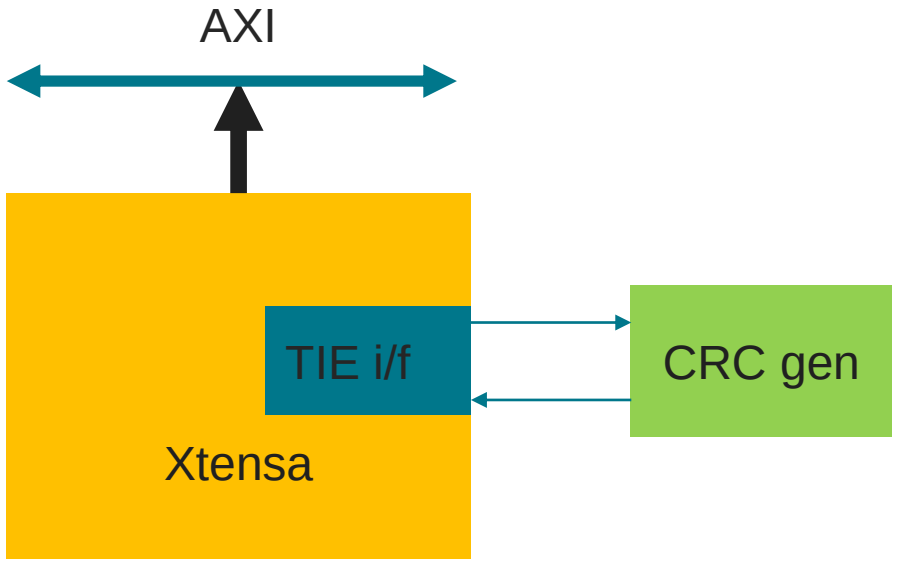
Innovating with TIE Today

- Design Problem: Implement a state machine-like function and couple it to a processor
  - E.g., CRC checker-generator
  - Execution as pure C code on general purpose CPU is likely to be inefficient
    - Consumes excessive processor resources
- Option 1: TIE
  - Capture computationally intensive tasks in TIE
    - Specialized XOR's, shifts
  - Control loop in software
  - ✓ Significant speedup over pure software approach
  - ✓ Control code flexibility
  - Performance less than that of pure RTL approach
    - Branching penalties and pointer fetch from memory can be bottlenecks



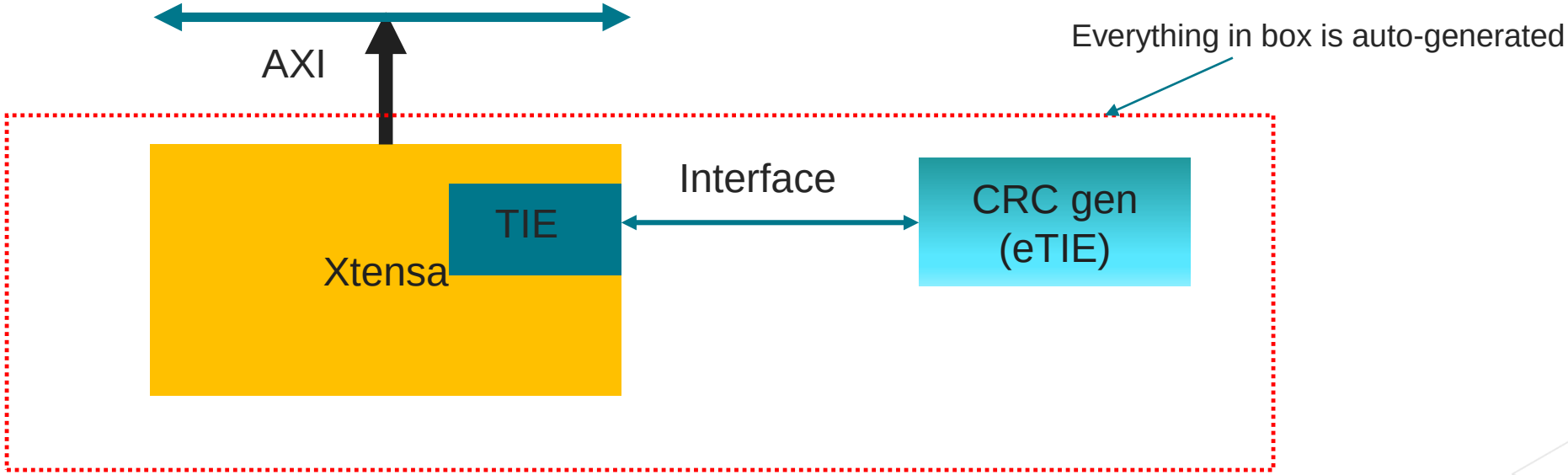
Innovating with TIE Today (cont.)

- Option 2: Hardware accelerator with TIE interfaces
  - CRC hardware accelerator designed in RTL
  - Data path supplied via TIE interfaces (queue or lookup)
  - ✓ Can achieve RTL-level performance
  - ✓ Relieves bottlenecks on system bus
  - - Less software flexibility than Option 1
  - - Time consuming to explore hardware vs. software tradeoffs
    - - Iterative approach



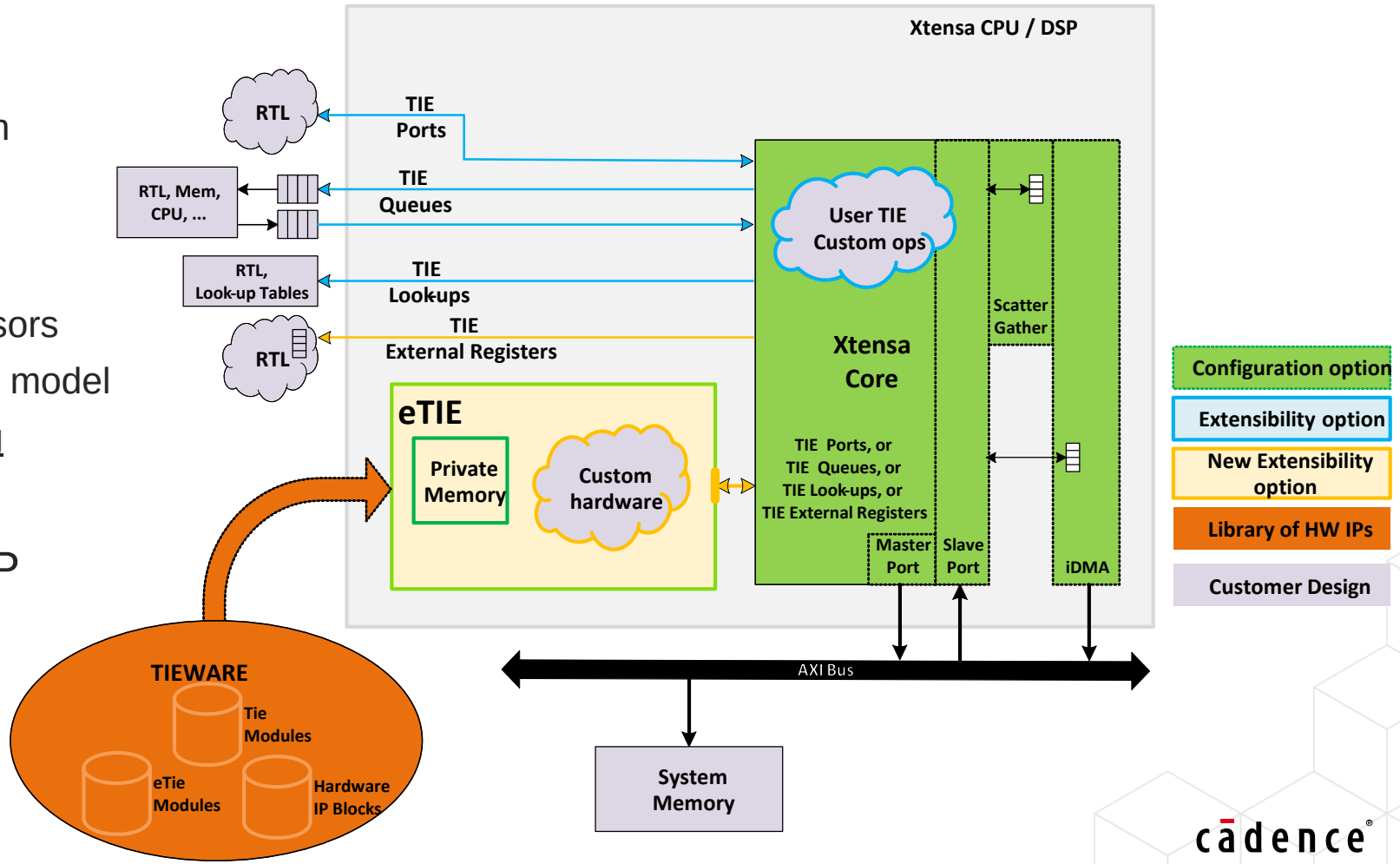
Introducing eTIE

- What it if was possible to design the external hardware accelerator using the TIE language?
  - One source for processor extensions, TIE interfaces, and external accelerator
  - Instruction set simulator can model functionality and performance of accelerator
  - Enables architects and designers to quickly explore performance vs. power vs. flexibility tradeoffs
  - Compiler can be aware of accelerator latency and optimize code schedules
  - Generated RTL would match software model



eTIE – New Extensibility option with Xtensa

- Enables generation of tightly-coupled hardware coprocessors
  - Specialized functions
  - Massive operational throughput
  - Intelligent IOs of increased dimension
- Auto-generation of development environment and tools
  - Cycle accurate C-model of coprocessors
  - Integrated coprocessor programming model
  - All with compiler, software and debug development tool chain support
- Library of pre-verified hardware IP components (TIEWare)



eTIE Anatomy

Running on Xtensa processor

```
queue INDATA 32 in
queue OUTDATA 32 out

operation QPUSH{in AR arr} {out OUTDATA}
{
   assign OUTDATA = arr;
}

operation QPOP{out AR art} {in INDATA}
{
   assign art = INDATA;
}

lookup DEVICE_REG {37, Wstage} {32 Wstage+1}

operation MEM_WR{ in AR ars, in AR arr } { out DEVICE_REG_Out, in
DEVICE_REG_In}
{
   assign DEVICE_REG_Out = {ars, arr[3:0], 1'b1};
}

operation MEM_RD{ out AR art, in AR arr } { out DEVICE_REG_Out, in
DEVICE_REG_In}
{
   assign DEVICE_REG_Out = {32b0, arr[3:0], 1'b0};
   assign art = DEVICE_REG_In;
}
```

Binding processor with HW accelerator

Instructions on processor

Hardware accelerator

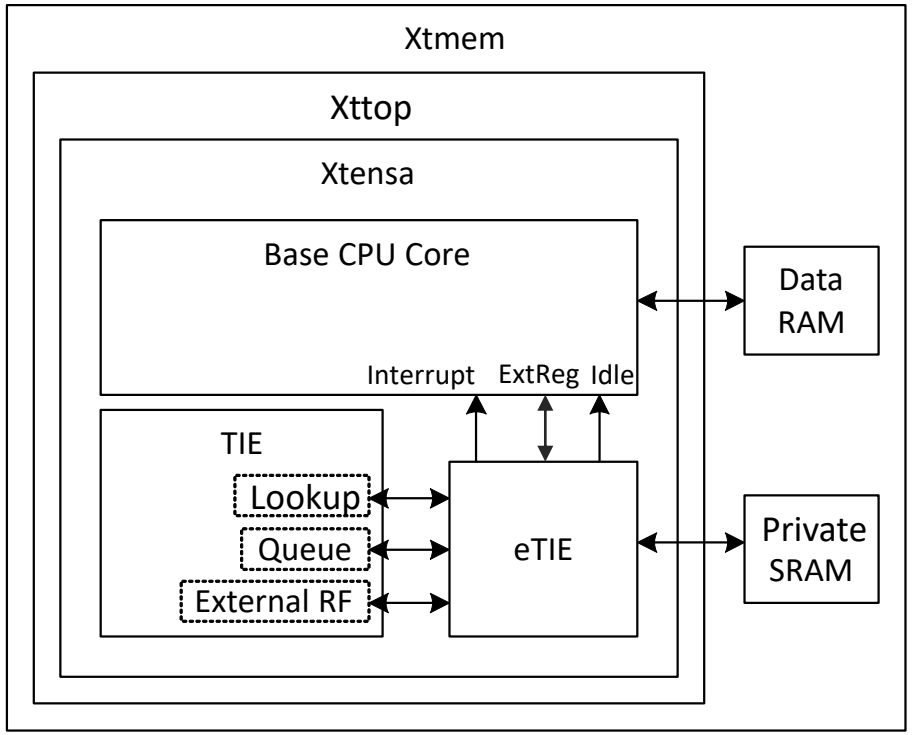
eTIE logic implemented by user (shown as "…" here)

```
package "" "Hardware TIE" ""“
module divide_quo_rem_32_s1(
…
) {
…
}
module divide_quo_rem_32_s2(
…
) {
…
}

module fifo (
…
) {
   …
}

property top_module {ha_div32}
module ha_div32(
queue INDATA,
queue OUTDATA,
lookup DEVICE_REG
) {

   wire csr0_en = DEVICE_REG.In[3:0] == 4’b0;
   wire [27:0] csr0_data = DEVICE_REG.In[31:4];
   assign DEVICE_REG.out = csr0_en ? csr0 : 32’b0;
   wire in_fifo_write = OUTDATA.Req;
   wire [31:0] in_fifo_data = OUTDATA.Data;
   assign OUTDATA.Full = in_fifo_full;
…
}
endpackage ""
```

Top-level module

Primary TIE to eTIE interfaces

- Built-in interfaces
  - Interrupt
  - External register (RER/WER)
- Instruction controlled interfaces
  - TIE queue
  - TIE Lookup
  - External register file port
- Control interfaces
  - Idle



eTIE Benefits

- Builds upon Tensilica's proven and best-in-class extension capability
- Automatically generates software model of Xtensa + eTIE
  - Enables up front analysis of hardware and software tradeoffs
  - Simulation environment is automatically generated
  - Easily debug functionality using TIE constructs
- Extends processor functionality in novel ways
  - Add independent computation engines
  - Hardware can be directly coupled to, and controlled by, processor
    - No need to design and connect interface between processor and hardware

eTIE Example

Use Model #1 – eTIE “Instruction”

- eTIE implements an instruction “foo” that takes *N* cycles to complete
  - *N* can be variable
  - Computation can be pipelined (i.e., *M* outstanding “foo”)
  - E.g., cordic, divide, etc.
- TIE instructions used for interacting with eTIE through TIE interfaces
  - Request → Send data to eTIE and start “function” compute
  - Response ← Get result back *N* cycles later
- Compiler is aware of the latency (*N*) of computation
  - Schedules independent instructions between *request* and *response*
- Compiler can also schedule and interleave multiple (*M*) requests

Use Model #2 – eTIE “standalone accelerator”

- eTIE implements an accelerator that processes a large data set
  - Can take 100s of cycles to process dataset
  - Data set, processed results, and any temporary data is stored in eTIE’s private memory
  - E.g., AES, FFT, matrix engine
- eTIE accelerator execution is decoupled from software on Xtensa
  - eTIE signals completion (or error events) to Xtensa via interrupts
- TIE instructions used for:
  - Sending/receiving data to/from eTIE’s private memory
  - Cmd/status query

eTIE Development Flow

- Build base configuration and optionally configure eTIE interrupts
- Write TIE and eTIE (can be one file or multiple files)
- Iterate in the *well-known* TDK flow
  - Run TIE Compiler
    - Automatically insert `TIEprint` on all eTIE internal wires and module arguments with optional `-waveform` flag
  - Write software and simulate core (TIE) and accelerator (eTIE) in Instruction Set Simulator
    - Core simulation can be sped-up with TurboXim, but eTIE is always cycle-accurate
  - Debug
    - Convert eTIE's TIEprint log to vcd and view in waveform viewer (e.g., SimVision) *post*-simulation
      - Cadence provides `etieprint2vcd` utility
    - View eTIE's internal wires using `info tie` command in a debug (i.e., `xt-gdb`) session
    - Use external register interface to peek/poke private memory and internal state (assuming eTIE logic support)
      - Via RER/WER instructions in C code
      - Via `mmio-rd`/`mmio-wr` in `xt-gdb`
  - Synthesize eTIE in TDK synthesis flow for early area/timing
- Submit to XPG

Writing eTIE and TIE

Fixed-Point CORDIC

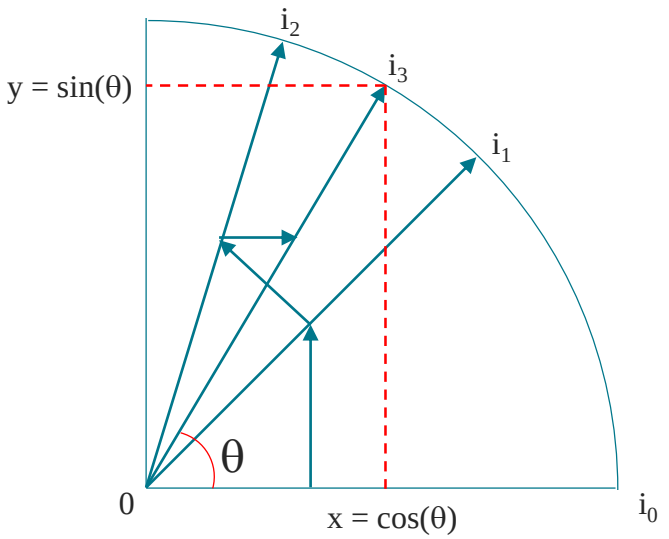
- Implement Givens rotation-based COordinate Rotation DIgital Computer
  - Algorithm to compute Sine and Cosine
  - Hardware efficient → uses shift-add (no multipliers)
  - Iterative → increasing accuracy with increasing iteration count

$$x[i+1] = x[i] - \sigma_i 2^{-i} y[i]$$
$$y[i+1] = y[i] - \sigma_i 2^{-i} x[i]$$
$$z[i+1] = z[i] - \sigma_i \tan^{-1} 2^{-i}$$

where,

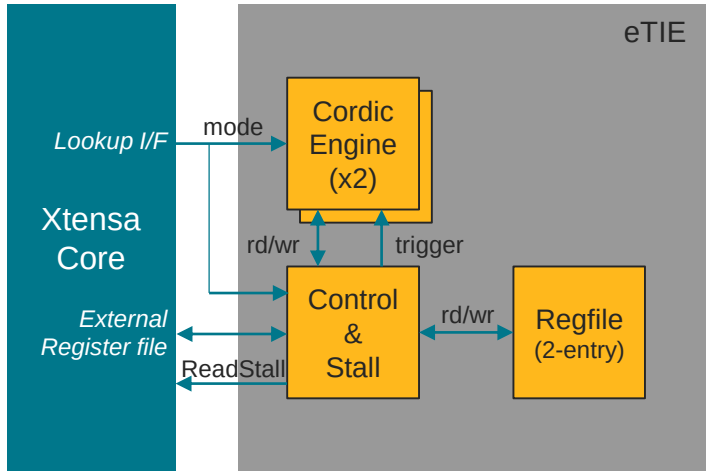
$i = 0, 1, \ldots N\text{-}1$, *and N is total number of iterations*

$\sigma_i = -1$ *if* $z[i] < 0$, *1 otherwise*



Block Diagram of Cordic Implementation in eTIE

- eTIE
  - TWO Cordic compute engines
    - Sine (or cosine) result produced every 32 cycles/engine
    - One Cordic iteration/cycle
  - 2-entry register file – accessible from Xtensa via *External Register File* Interface
    - One entry per Cordic compute engine
    - Regfile write copies input operand from Xtensa and starts computation
    - Cordic engine updates corresponding regfile entry when result available
    - Regfile read from Xtensa reads result of computation
      - Ability to stall Xtensa pipeline if result is not available
  - 2-bit lookup interface to distinguish between normal writes and sine/cosine writes to register file
- Extensions to Xtensa Pipeline
  - TIE Interfaces
    - Lookup for computation mode control (i.e., sine/cosine/etc.)
    - External register file to copy operands to h/w engine and read results back
  - TIE operations to work with the TIE interfaces



TIE+eTIE Design Walkthrough

- TIE source can contain eTIE and regular TIE
- eTIE is enclosed in ETIE package
- eTIE has a single top level TIE module
  - `property etie_top_module {<module name>}`
  - Argument list contains all TIE I/Fs used between Xtensa and eTIE in the design
- eTIE can contain multiple TIE modules
  - A TIE module may instantiate another TIE module
  - Similar to Verilog Module; used to create hierarchy under eTIE top module

```
//----------------------------------------
// Regular TIE - Implemented in Xtensa
//----------------------------------------


//----------------------------------------
// eTie
//----------------------------------------

// Use "package to indicate eTIE
package "etie" "ETIE" ""

// Use "property" to indicate top module of eTIE
property etie_top_module {etie_top}

module etie_top (
   regfile EREG,
   lookup MODE
) {
   // ...
   cordic #(2) cordic (res, busy, done, start,
angle_in, cordic_mode );
}

module cordic (
// ...
) {
// ...
}

endpackage "etie"
```

Cordic eTIE Module

- Described with TIE module
  - Multiple inputs and outputs
  - Multi-cycle
- Computes sine/cosine for a given input angle
  - Input is 32-bit fixed point value in `angle_in`
  - `mode_in` determines if sine or cosine is computed
- Result is produced in `res_out` 32 cycles later
  - Indicated by `done_out`
- While computation is in progress, `busy_out` stays asserted

```
module cordic(
  out [31:0]     res_out,  // 32 bit result
  out            busy_out, // Indicate cordic module is busy
  out            done_out, // Indicate cordic computation is done
  in             start_in, // Pulse signal to trigger start
  in [31:0]      angle_in, // Input angle
  in             mode_in   // Input mode.  0 - sin   1- cos
)
{
    // Implementation (discussed on next slide)
}
```

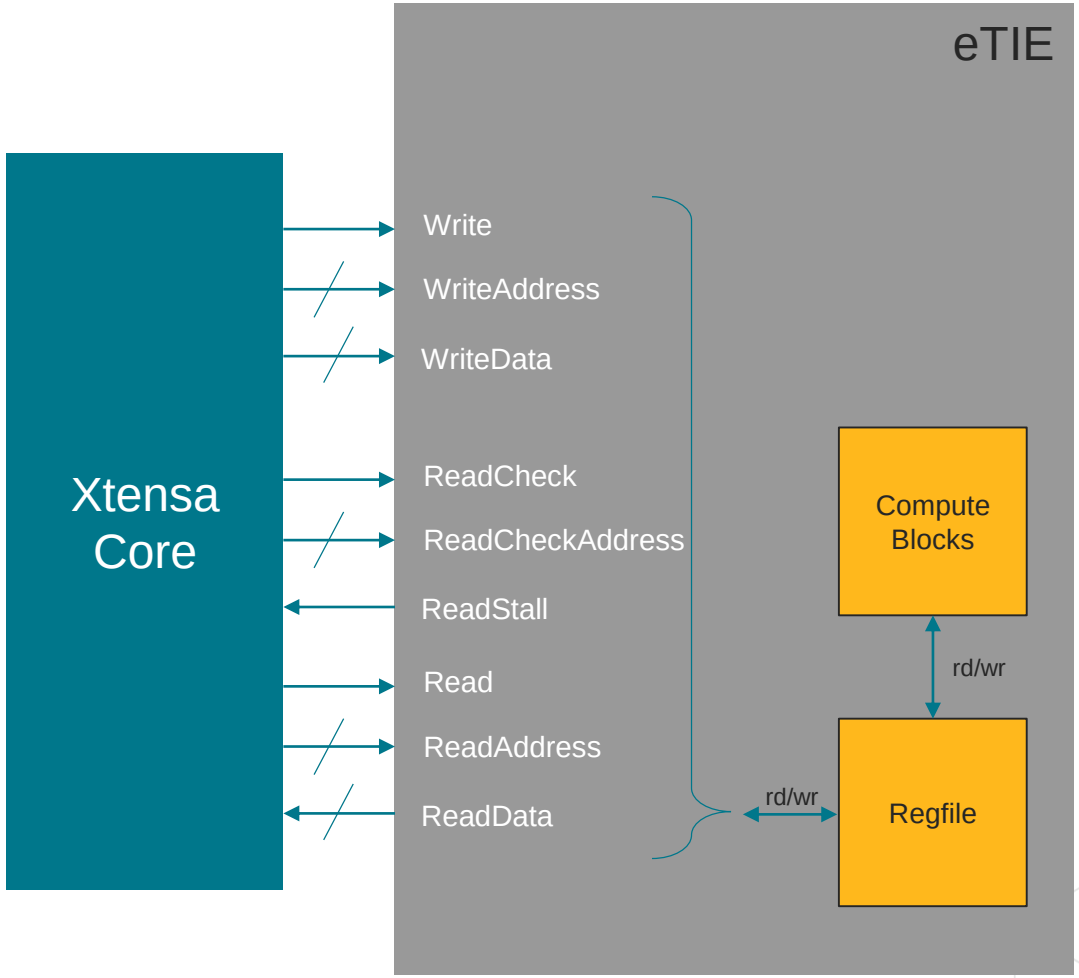
Cordic eTIE Module (contd.)

- Multi-cycle control and dataflow expressed in TIE Language
- `wire` datatype
  - Can also be a `reg` implicitly in sequential logic
- Continuous assignment
  - No concept of blocking/non-blocking Verilog assignments
  - Most Verilog operators supported
- Sequential logic implemented using `TIEflop` or `TIEenFlop` built-in modules
- If-else and case statements implemented using `TIEsel` and/or `TIEmux` built-in modules
- TIE perl preprocessor (`tpp`) can be used

```
module cordic(
  …
)
{
 `
…`
wire [31:0] angle_next;
wire [31:0] angle;

wire [4:0]  count_next;
wire [4:0]  count;
assign count_next = start_in ? 5'b0 : busy ? count + 5'b1 : count;

assign count = TIEenFlop(count_next, start_in || busy);

// beta lookup table
; for(my $i=0; $i<32; $i++) {
  wire [31:0] beta_lut`$i` = `$BETA[$i]`;
; }

wire [31:0] beta = TIEmux(count,
; for(my $i=0; $i<32; $i++) {
  beta_lut`$i` `$i==31?');':','`
; }
assign angle_next = start_in ? angle_in : busy ?
(direction_negative ? (angle + beta) : (angle - beta)) : angle ;
assign angle   = TIEenFlop(angle_next, start_in || busy) ;
…
}
```

Usage of External Register File

- Within eTIE
  - Used alongside compute blocks for data in/out and flow control
  - Use model
    1. Request initiated by *non-blocking* regfile write
    2. Response through *blocking* regfile read to same entry
    - Two internal phases - `check` and (subsequent) `read`
    - eTIE can stall processor pipeline in `check` phase if response not available
  - Regfile depth controls maximum #outstanding
  - Actual storage is implemented in eTIE
- From perspective of TIE Instruction, external regfile looks like any other register file; fixed use/def schedule however
- Compiler manages multiple outstanding requests as separate external regfile entries
- A *typical* latency is specified in TIE between request (write) and response (read), allowing compiler to schedule other instructions in between
  - Response (read) latency can be extended by eTIE via hardware protocol and processor will stall



Implementation in eTIE

```
module etie_top
(
      ...
      regfile EREG
)
{
//-------------------------------------------------------------------------
// External Regfile's register entry 0 (entry 1 not shown for simplicity)
//-------------------------------------------------------------------------

// Define external register 0 of regile
wire [31: 0] ereg0;
wire [31: 0] ereg0_next;

// Decode to register write signals
wire ereg0_wr = EREG.Write && (EREG.WriteAddr == 1'd0);
wire start0 =  ereg0_wr && (op_sin_delay || op_cos_delay);
wire cordic0_wr = done[0];

// Write register
assign ereg0_next = cordic0_wr ? res[31:0] : ereg0_wr ? EREG.WriteData: ereg0;
assign ereg0 = TIEenFlop(ereg0_next, ereg0_wr|cordic0_wr );

// Read register
assign EREG.ReadData = TIEmux(EREG.ReadAddr, ereg0, ereg1) ;

// Read Stall Logic
wire stall_mux;
assign stall_mux = TIEmux(EREG.ReadCheckAddr,(busy[0]||start0),(busy[1]||
start1)) ;
assign EREG.ReadStall = EREG.ReadCheck ? stall_mux : 1'd0;
...
}
```

- Top eTIE Module declares external register file in argument list
- Each individual port can be accessed in eTIE Module

- Regfile write updates register file
- Optionally starts computation
  - Note that some writes are related to compiler-mandated ld/st/mv.

- Regfile is implemented in eTIE using built-in `TIEenFlop` module to

- Regfile read is split in to Check and Read
  - In check phase, eTIE can use `ReadStall` to stall Xtensa pipeline
  - In read phase, eTIE returns data

TIE Instructions

```
lookup MODE  {2, Wstage-1} {2, Wstage}
regfile EREG {
    width : 32,
    depth : 2,
    short_name : er,
    external : 1,
    direct_bypass : 0
}
```

```
operation SIN_EREG {out EREG ereg, in AR ars} {out MODE_Out, in MODE_In} {
    assign MODE_Out = {2'd1};
    assign ereg = ars;
}
schedule sch_etie_def {WR_EREG, SIN_EREG} {
    use ars Wstage;
    def ereg  Wstage;
}
```

```
ctype ereg 32 32 EREG default
operation WR_EREG {out EREG ereg, in AR ars} {out MODE_Out, in MODE_In} {
    assign MODE_Out = {2'd3};
    assign ereg = ars;
}
proto ereg_loadi {out ereg a, in ereg* b, in immediate c}{uint32 tmp}{
    L32I tmp, b, c;
    WR_EREG a, tmp;
}
```

```
proto SIN_EREG_DELAY {out ereg a, in int32 b, in immediate delay} {} {
   SIN_EREG a, b;
}
property variable_def SIN_EREG_DELAY a delay
```

- TIE interfaces that communicate with eTIE
  - Lookup
  - Regfile is declared `external`

- TIE operations to work with these TIE interfaces
  - External regfile `use`/`def` is fixed
- Can also be FLIXed

- Can declare `ctype` for external regfile
- Need to provide compiler-mandated load/store/move protos
  - Load/store to external regfile is always through Xtensa internal regfile

- Compiler can be provided a typical latency between request (write) and response (read) through property `variable_def` and use in `proto`
  - Useful *hint* for scheduling
  - Latency can be changed in C code as needed

TDK Flow – Compile eTIE+TIE, Simulate, Debug

Compiling TIE+eTIE

- Run TIE Compiler in Xtensa Xplorer UI or command-line
- Automatically insert `TIEprint` on all eTIE internal wires and arguments with `-waveform` flag

```
tc -d <tdk_dir> -waveform cordic.tie
```

Software for Cordic eTIE

- Request (`SIN_EREG_DELAY`) and response (`RD_EREG`) are related by ctype variable (e.g., `ereg sin0`)
- Typical latency *compiler hint* for a cordic computation is an argument to the request (`SIN_EREG_DELAY`)
- Compiler uses latency to minimize pipeline bubble between request and response during scheduling
- Compiler will schedule outstanding requests up to the #entries in external register file

```
ereg sin0, sin1, sin2, sin3;
int i;

for(i=0; i <= ITERATIONS-1; i=i+4) {

  // Invoke TIE
  sin0 = SIN_EREG_DELAY(rad_input[i+0], 32);
  sin1 = SIN_EREG_DELAY(rad_input[i+1], 32) ;
  sin2 = SIN_EREG_DELAY(rad_input[i+2], 32);
  sin3 = SIN_EREG_DELAY(rad_input[i+3], 32) ;

  cordic_sin_actual[i+0] = RD_EREG(sin0);
  cordic_sin_actual[i+1] = RD_EREG(sin1);
  cordic_sin_actual[i+2] = RD_EREG(sin2);
  cordic_sin_actual[i+3] = RD_EREG(sin3);
}
```

```
20001073: { loopgtz    a8, 200010eb ; movi a2, 0; movi a3, 90; mov.n a9, a14 }
20001083: l32i.n       a12, a10, 0
20001085: sin_ereg     er0, a12
20001088: { l32i.n     a13, a10, 8; l32i.n a15, a10, 4 }
20001090: rd_ereg      a8, er0
20001093: sin_ereg     er0, a15
20001096: sin_ereg     er1, a13
20001099: { s32i.n     a8, a1, 8; l32i.n a11, a10, 12 }
200010a1: rd_ereg      a12, er1
200010a4: sin_ereg     er1, a11
200010a7: { s32i.n     a12, a1, 0; l32i.n a15, a1, 8 }
200010af:  rd_ereg     a8, er1
200010b2: wr_ereg      er1, a15
200010b5: { s32i.n     a8, a1, 4; l32i.n a13, a1, 0 }
200010bd: rd_ereg      a11, er1
```

Simulating TIE+eTIE

- Simulate core (TIE) and accelerator (eTIE) in ISS
  - XTSC is not required
  - eTIE simulation is always cycle-accurate
  - Core (TIE) simulation can be cycle-accurate or functional (i.e., TurboXim)
- eTIE `TIEprint` can be written to logfile
- Convert to `vcd` for inspection in waveform viewer (e.g., SimVision)
  - Can choose to ALL or selected eTIE modules

```
xt-run cordic --etieprint=etie.log -Xtensa-params=<tdk_dir>

tieprint2vcd -i=etie.log [-m=m0,m1,m2,…]
```

Viewing `vcd` in Waveform Viewer