



# The World's Fastest DSP Core: Breaking the 100 GMAC/s Barrier

Hot Chips 23 - August 2011

Chris Rowen

Dan Nicolaescu, Rajiv Ravindran, David Heine, Grant Martin, James Kim, Dror Maydan, Nupur Andrews, Bill Huffman, Vakis Papaparaskeva, Shay Gal-On, Peter Nuth, Pushkar Patwardhan, and Manish Paradkar

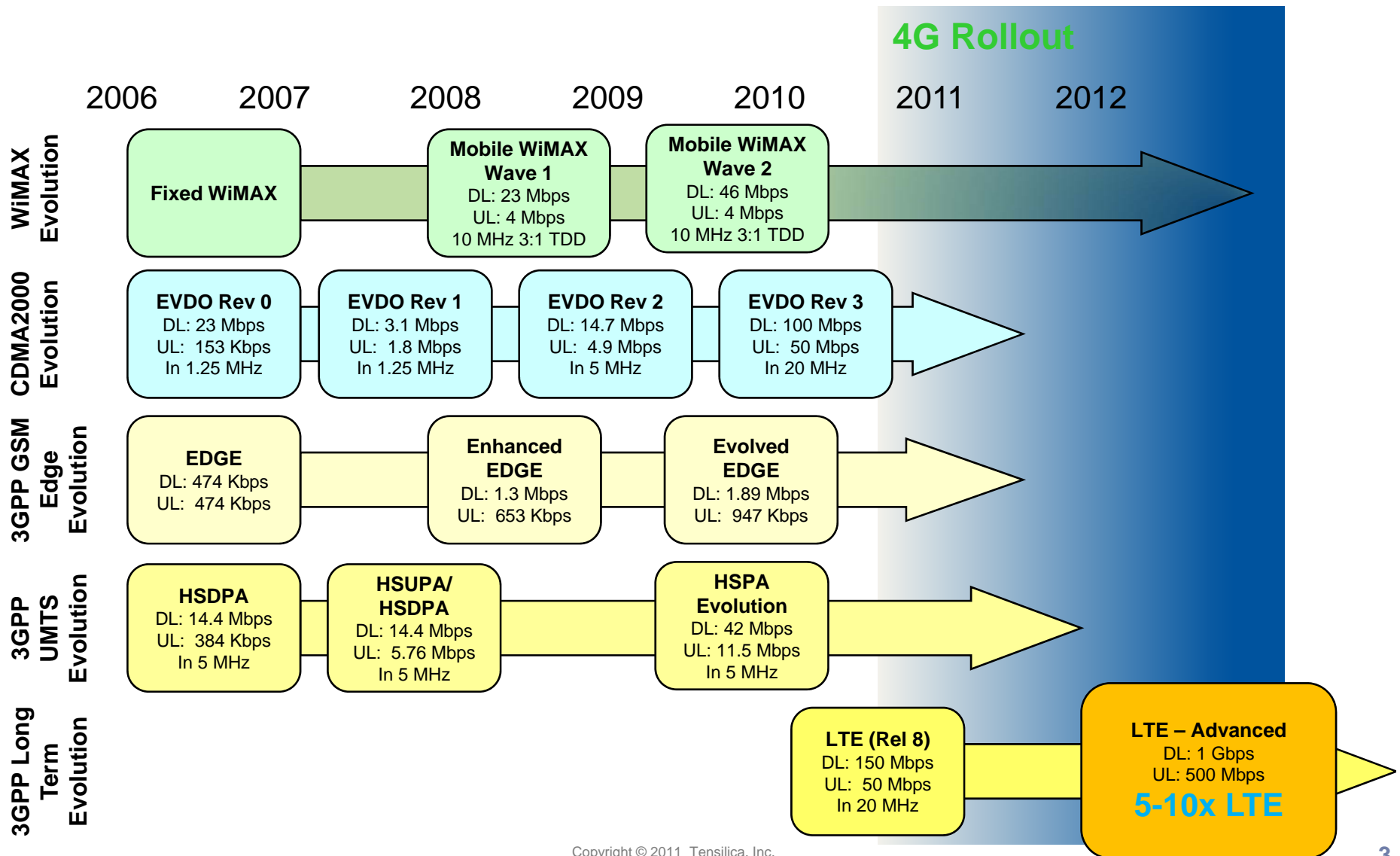
Tensilica Inc.

# Outline

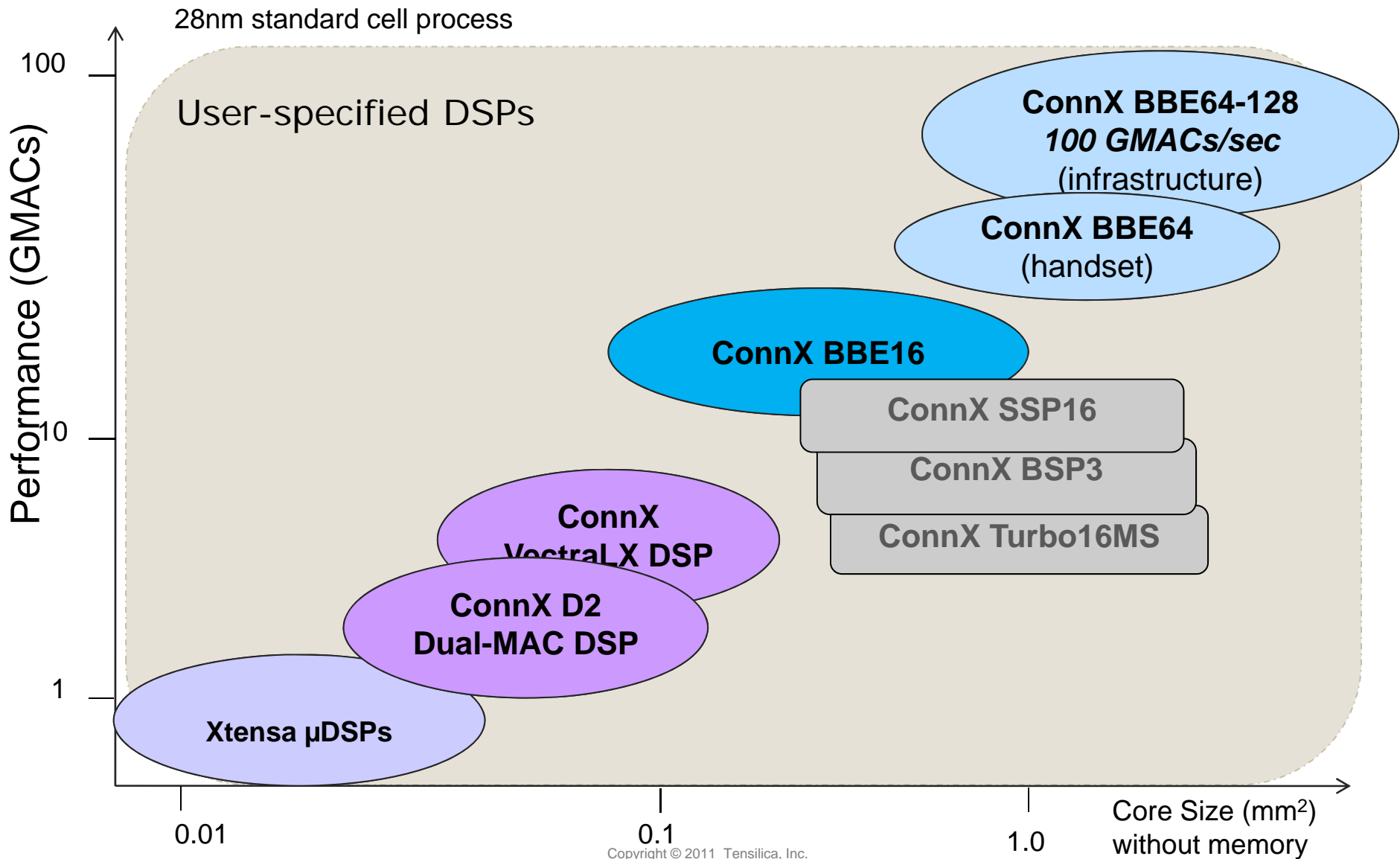
---

- Introduction: 4G Wireless Processing for LTE and LTE-Advanced
- BBE64 Fundamental Goals
- Instruction Set Design:
  - SIMD//VLIW
  - Programmer State and Register Organization
  - Operations
  - Example: Select and FIR operations
  - Execution Rate Summary
- BBE64: Where Does the Hardware Go?
- BBE64 Software
  - SIMD-Width Independent Programming Model
  - LTE-Advanced Execution Profile
  - Kernel Examples
- Implementation and Verification Timelines
- Tools for Processor Automation

# Evolution of Major Cellular Standards



# Product Context: Specialized and General Baseband DSPs

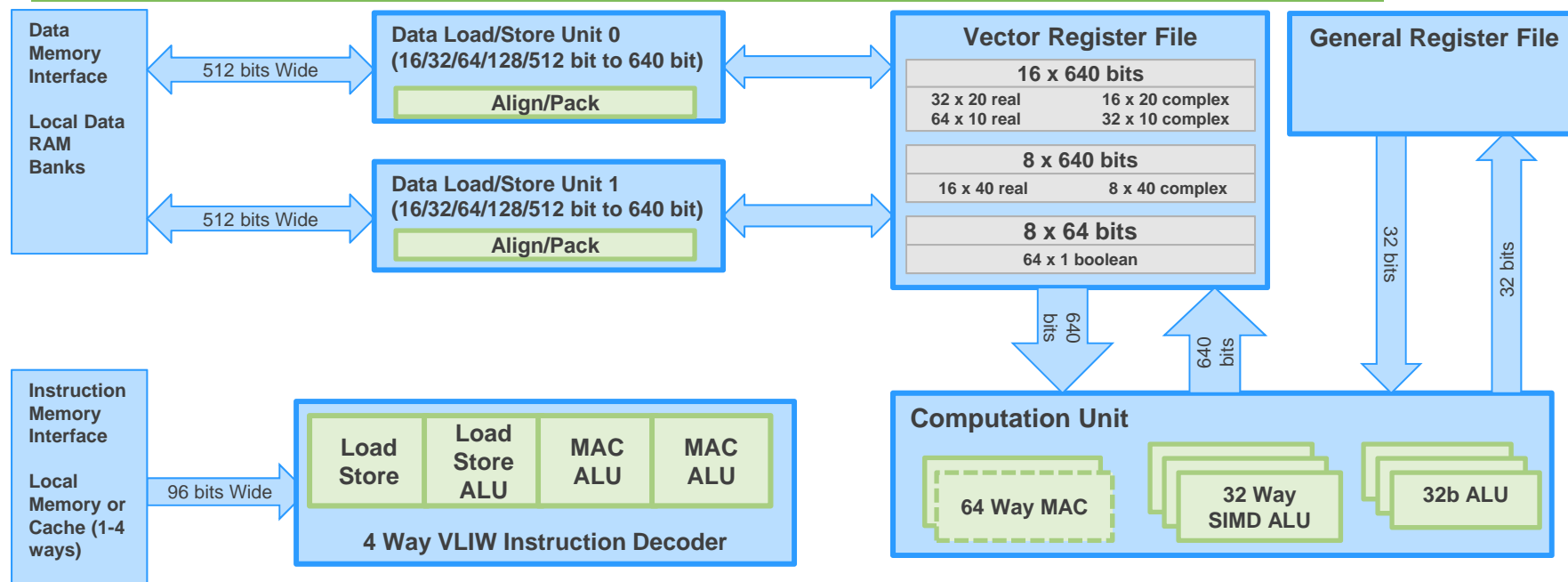


## BBE64 Family Design Goals and Philosophy

---

- World-leading DSP performance for baseband PHY in handsets (aka “User Equipment” or “UE”) and infrastructure (aka base-stations or “eNodeB”)
- Up to 1GHz in available 28nm fast standard cell process x 128 MAC
- Combine SIMD, VLIW and configurable instruction set features for large applications “sweet-spot”.
- Leverage high memory system bandwidth of Xtensa LX4 – 1024b per cycle
- Good control code performance with multi-issue base ops, including multiple load/store per cycle
- Offer both a broad range of built-in options and user-defined extensions to fit:  
Two initial base configurations BBE64-UE, BBE64-128
- Leverage advanced vectorizing compilers, C scalar/vector data-types, operator-overloading and optimized intrinsics to eliminate need for assembly coding
- ConnX BBE16 upward compatibility
- Fully synthesizable RTL, with complete system modeling, verification and back-end flows environment

# ConnX BBE64 Block Diagram



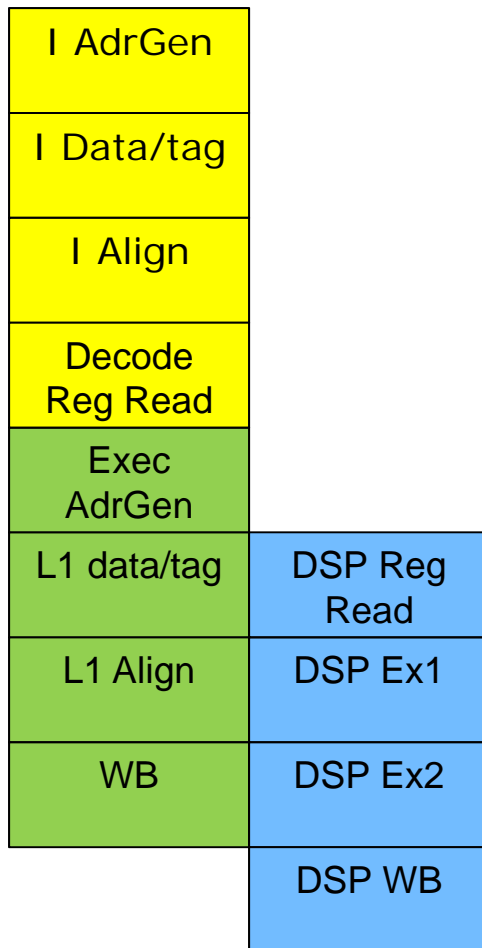
## Optimized Architecture for DSP Applications

- 4-way VLIW x 32-way SIMD 128 DSP ops/cycle
- 16/24/96b 4-issue VLIW – almost any instruction in any slot
- 128 MAC ops/cycle for matrix and filter functions (BBE64-128)
- Guard-bits on all DSP data for numerical accuracy
- Protected pipeline: interlocks/bypasses for robustness
- Support for all data types from C
  - Complex/real
  - Scalar/vector
  - Fractional/integer

## High Bandwidth Configurable Memory Subsystem Interface

- Dual load/stores with dual 512b memory interfaces
- Full bandwidth on packed and unaligned data vectors
- DMA support for local data memory
- Extensible with special memory ports and direct-connect data queues: 4 x 640b per cycle

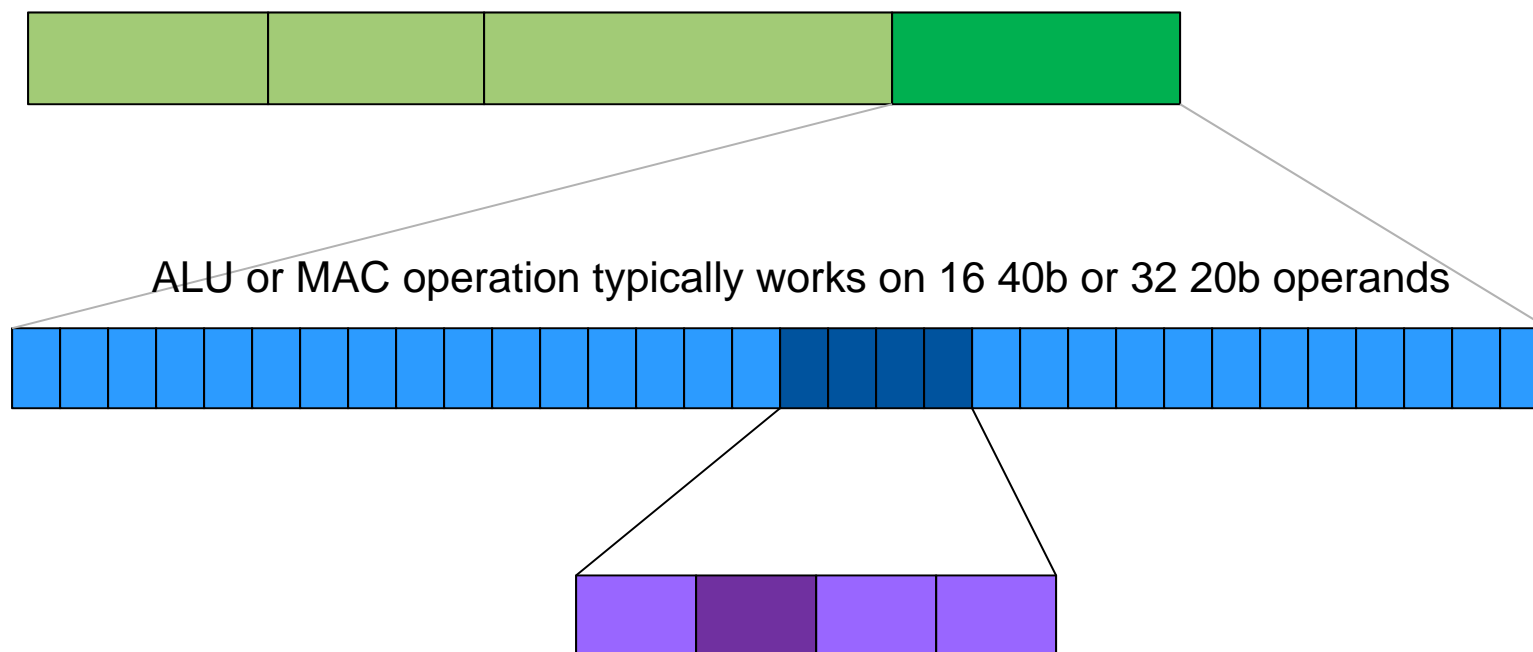
# BBE64 Pipeline



- Two pipeline options:
- 9 stage pipeline – higher MHz or larger memories
- 7 stage pipeline – lower power and area
- Wide static in-order issue
- No branch prediction, but zero-overhead loops and SIMD predication
- Simple length encoding enables single-stage instruction decode and register specifier extraction
- DSP operations start with load return: zero load-use bubbles
- Simplified ALU/MAC operations allow DSP pipe reduction to two stages + writeback for reduced regfile cost – fewer values in flight, better utilization of slots

# Typical Data Parallelism in SIMD/VLIW Ops

VLIW Instruction typically encodes 1-2 load/store, 2-3 ALU and MAC operations



Each 80b may represent 4 20b real, 2 20b+20b complex, 2 40b real, or 1 40b+40b complex

Two 18b x 18b multipliers per 20b element – paired real multiply or half a complex multiply

# Baseline DSP instruction set summary

## *550 distinct DSP ops above base Xtensa ISA*

### Load:

- **LV<m>[SU][TF].{I,IP,IC,X,XP}**: load vector of 16b elements
- **LP<m>[SU][TF].{I,IP,IC,X,XP}**: load complex pair of 16b elements
- **LS<m>[SU][TF].{I,IP,IC,X,XP}**: load scalar 16b element
- **LA<m>[SU][TF].{I,IP,IC,X,XP}**: load unaligned vector of 16 elements
- Plus specialty loads for guard-bits, boolean vectors, compressed data

**<m>**= 16X32, 32X16

**{I,IP,IC,X,XP}** addressing modes:

- .**I**: base register + immed offset
- .**IP**: base register + immed offset, post-update
- .**X**: base register + register index
- .**XP**: base register + register index, post update
- .**IC**: base register+ increment, post update in circular buffer

All instructions prefixed with “**BBE\_**”

### Store:

- **SV<m>[SU][TF].{I,IP,IC,X,XP}**: store vector of 16b elements
- **SP<m>[SU][TF].{I,IP,IC,X,XP}**: store complex pair of 16b elements
- **SS<m>[SU][TF].{I,IP,IC,X,XP}**: store scalar 16b element
- **SA<m>[SU][TF].{I,IP,IC,X,XP}**: store unaligned vector of 16 elements
- **SV<m>PACK[QSR].{I,IP,IC,X,XP}**: pack 40b elements to 16b and store vector
- **SP<m>PACK[QSR].{I,IP,IC,X,XP}**: pack 40b elements to 16b and store pair
- **SS<m>PACK[QSR].{I,IP,IC,X,XP}**: pack 40b elements to 16b and store scalar
- Plus specialty stores for guard-bits, boolean vectors, expanded data, bit-reversed addressing

# Baseline DSP instruction set highlights

## ALU:

- **ABS<s>**- absolute value
- **[R]ADD<s>[TF]** - add
- **AND<s>**- bitwise and
- **CONJ<s>**- conjugate complex
- **EQ<s>[C]** – set boolean on equal
- **LT<s>**- set boolean on less than
- **LE<s>**- set boolean on less than or equal
- **[R]MAX[U]<s>[TF]** - maximum
- **[R]MIN[U]<s>[TF]** -minimum
- **MOV<s>[TF][C]** – move conditional
- **NAND<s>[TF]** – bitwise not and
- **NEG<s>[TF]** - negate
- **NEQ<s>[C]** – set boolean on not equal
- **[R]NSA[U]<s>[C]** – normalize shift amt
- **OR<s>**- bitwise or
- **SAT<s>[SU]** – saturate to memory size
- **SL[ALS]<s>{B,BR}** – shift left
- **SR[ALS]<s>{B,BR,I}** – shift right
- **SUB<s>[TF]** - subtract
- **XOR<s>**- bitwise xor

## Multiply:

- **MAG32X18C{PACKQ,PACKS}** complex magnitude
- **MUL32X18{C,J,JC}{,PACKQ,PACKS}** multiply
- **MULA32X18{C,J}{,PACKQ,PACKS}** multiply-add
- **MULSIGN<s>**
- **POLY\_STEP32X20.STEP** – Polynomial series expansion with parallel table lookup

## Data Organization:

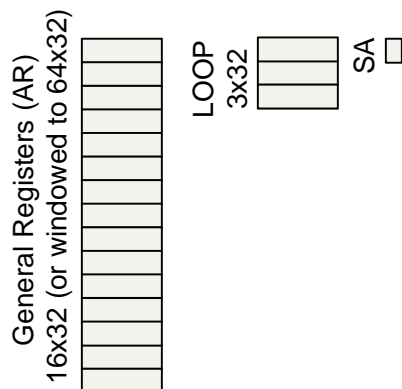
- **PACK[LQRS]32X40** – pack 40b to 20b
- **UNPACK32X20** – unpack from 20b to 40b
- **SEL<s>[I]** - select elements from vector pair
- **SHFL<s>[I]** – select elements from vector
- Plus similar operations for boolean vectors

<s>= 16X40, 32X20

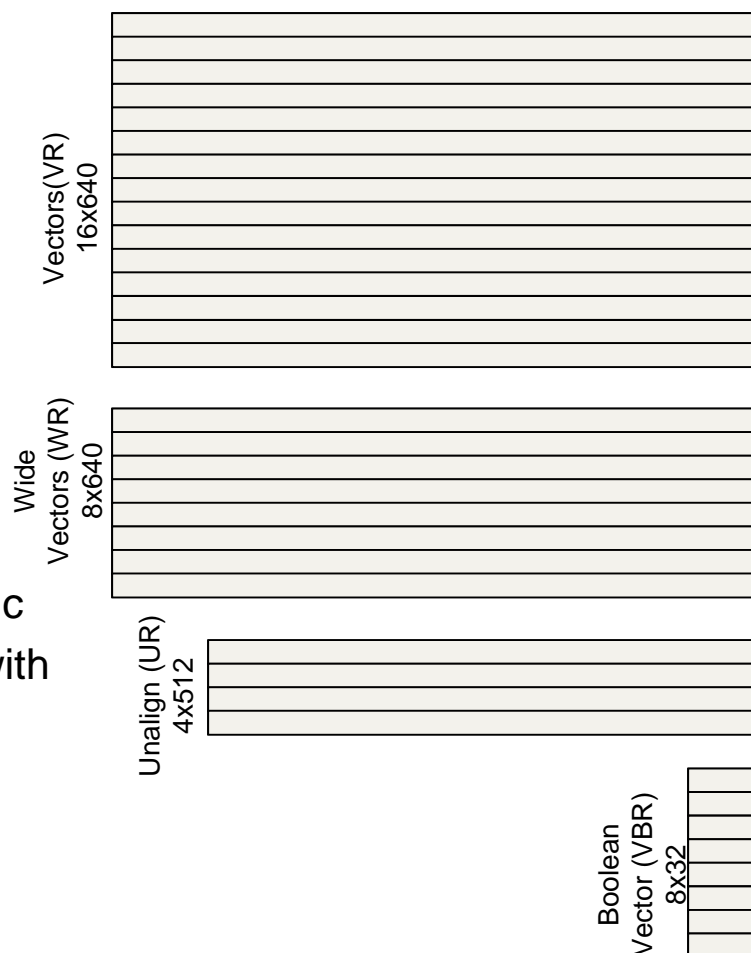
All operations prefixed with “**BBE\_**”

# Programmers Model: Registers

Base Xtensa ISA (user)

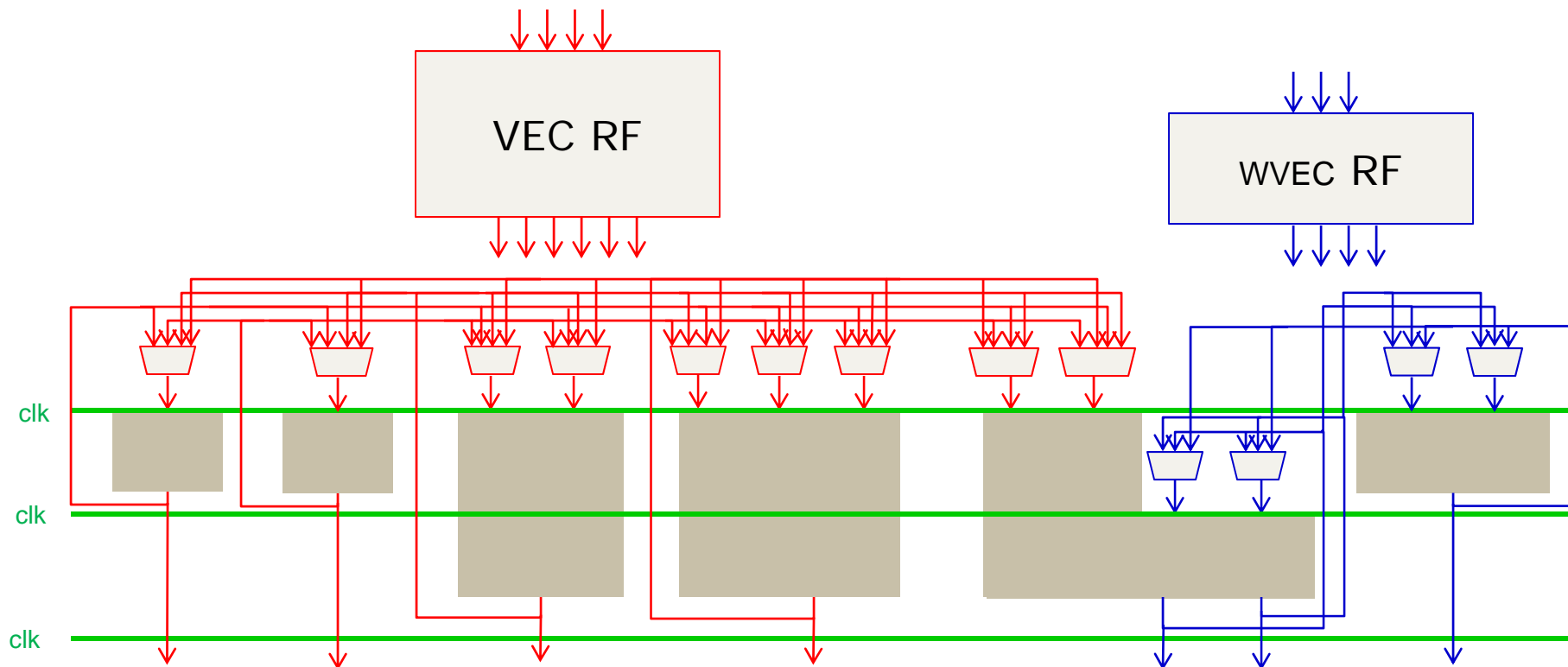


BBE64 ISA Extension



- Large vector register file supports deep software pipelining and reduced memory traffic
- Partitioned register file for added bandwidth with less register bloat (narrow/wide operands)
- Unalignment registers enable full bandwidth streaming of arbitrary alignment and size
- Vector booleans for flexible SIMD and VLIW predication
- Aggregate regfile bandwidth at 1GHz: >12Tbps

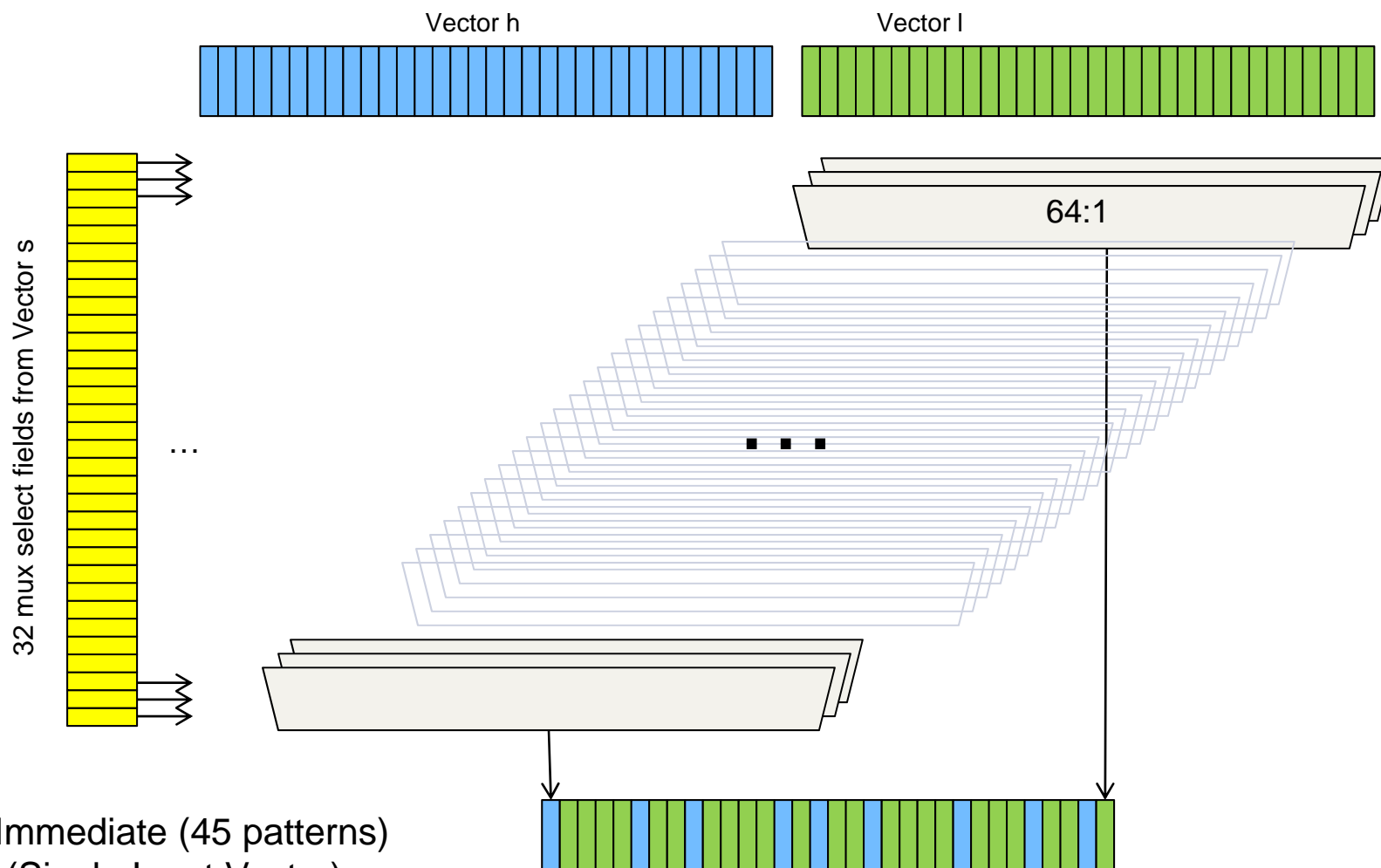
# Register File Organization



Register file partitioning reduces abstracted bypass structure

# Data Reorganization Key to SIMD: Selection

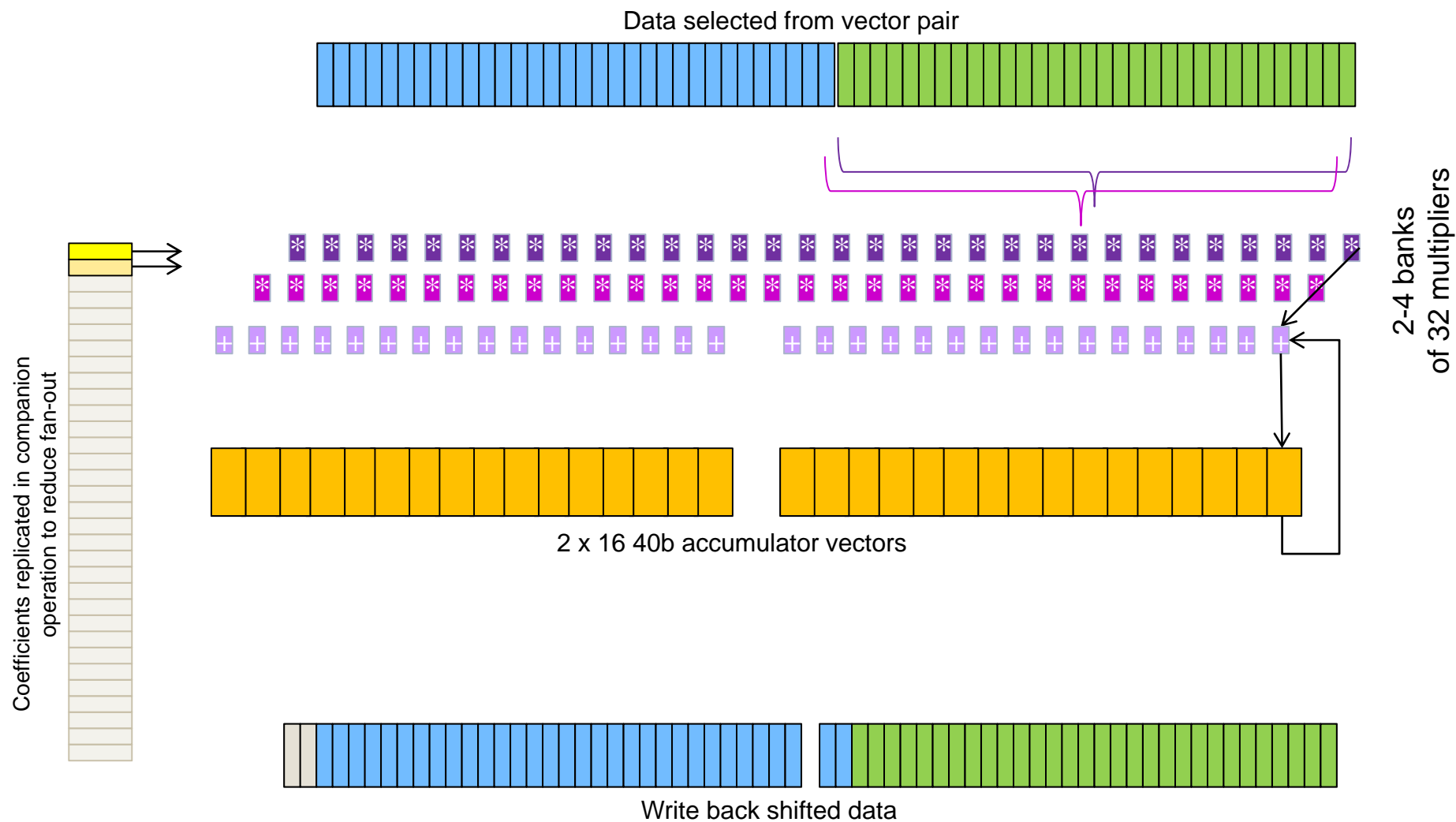
Example: operation **BBE\_SEL32X20** {out vec c, in vec h, in vec l, in vec s}



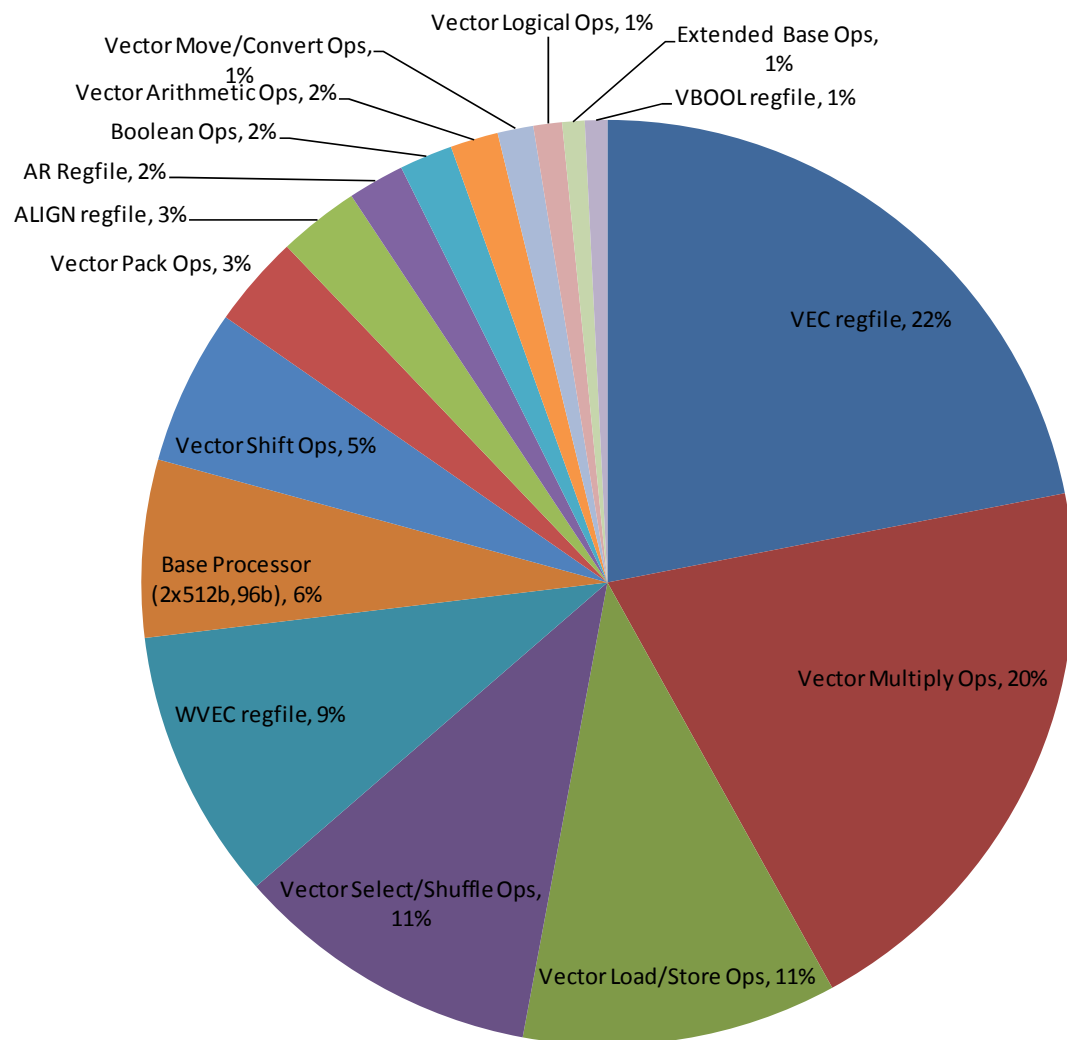
Options:

- Select Immediate (45 patterns)
- Shuffle (Single Input Vector)
- Shuffle Immediate (75 patterns)

# Accelerated Operation Examples: Real FIR

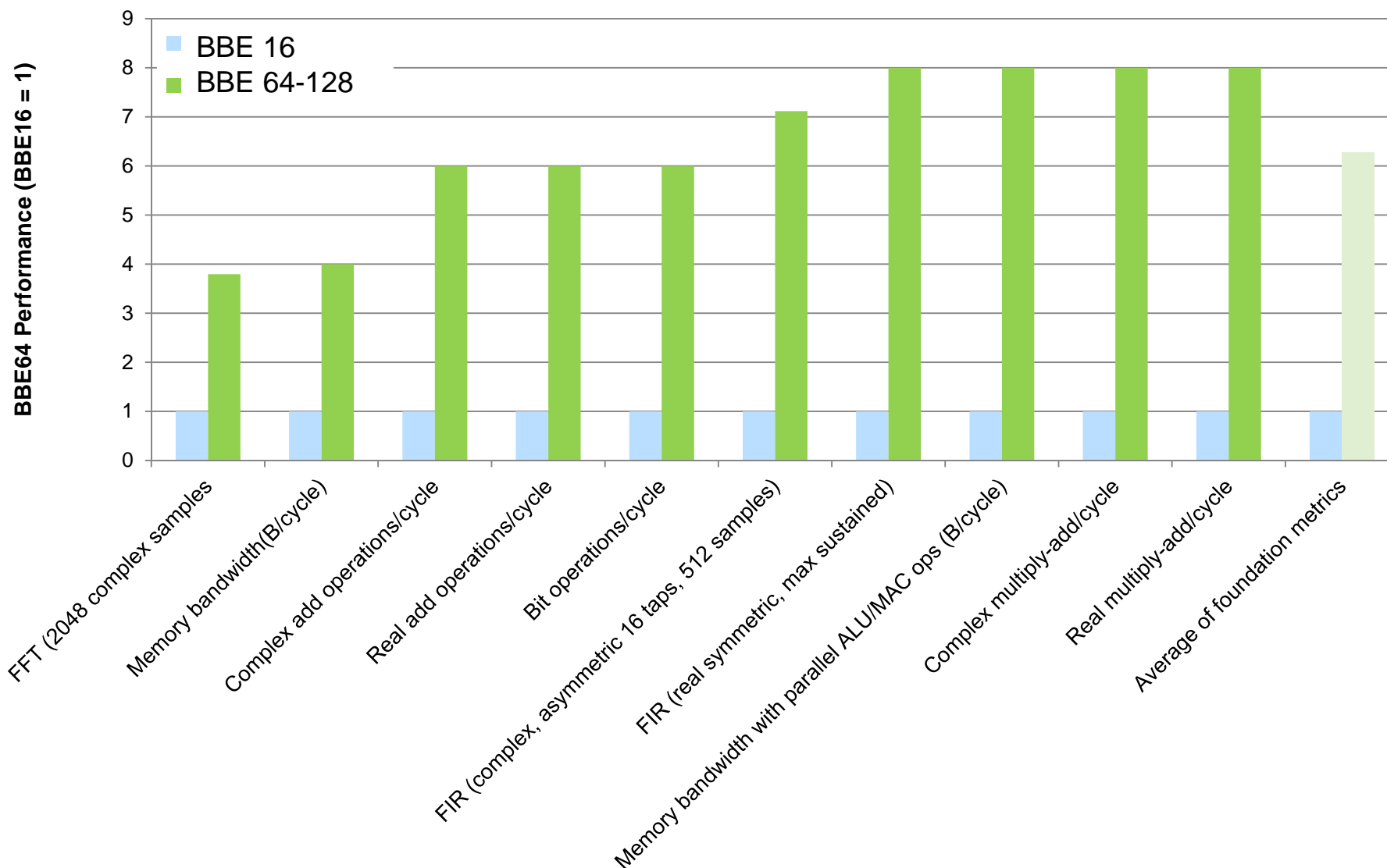


# Where Does the Hardware Go?



- Register files are pretty big
- As expected, multipliers dominate execution unit size
- Select and shift are significant
- Base processor and extensions for 3-way base ISA execution are very small
  - Invites more aggressive control code resource allocation

# Relative Performance on Basic Metrics



Preliminary - subject to change

**Performance Metric**

Copyright © 2011 Tensilica, Inc.

# Vector-Length Independent Programming

- ▀ BBE64 is one of a family of compatible machines:
  - 4-way to 32-way SIMD
  - Common code base for HW, verification, libraries, software tools
- ▀ Allows common code across DSP versions
- ▀ Much code is automatically vectorizable by compiler to a version's vector length
- ▀ Some code most conveniently expressed in vector form
  - Compiler still does operation selection, software pipelining, scheduling, register allocation etc.

Where explicit vector representation makes sense:

- Data marshaling by selecting operands within vectors
- Parallel operations with vector compression and expansion
- Explicit vector reductions
- Some special-purpose operations

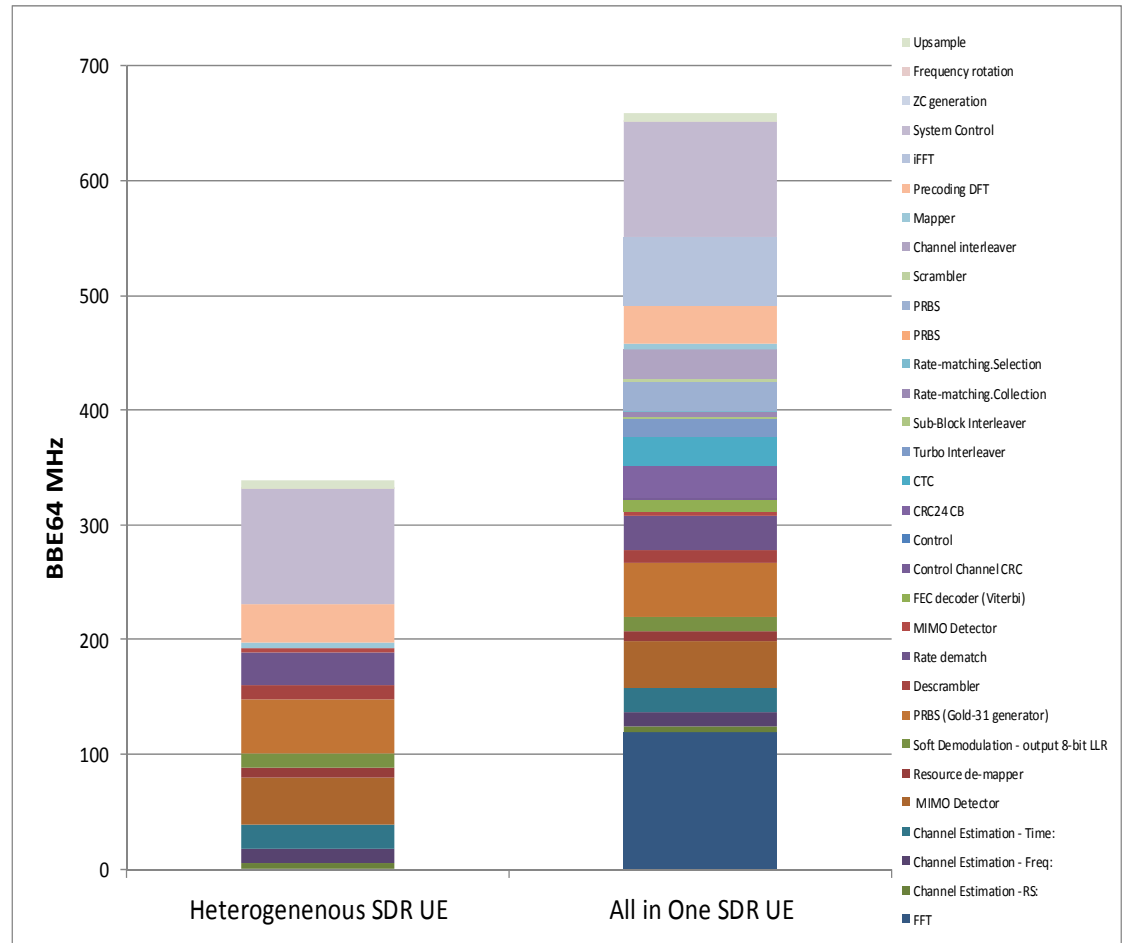
## Simple case:

```
xb_vecNx20 vin;  
xb_vecNx40 vout = 0;  
for (i=0;i<N/BBE_SIMD;i++) {  
    vout += vin[i] * vin[i];  
}  
*out_p = BBE_RADDN_2X40(vout);
```

# Profiling BBE64 for LTE-Advanced 2x2 MIMO x 100MHz



- User Equipment is power-obsessed, but flexibility increasingly important: want to run 3G, WiFi on same hardware
- Many possible baseband PHY subsystem design approaches for LTE-Advanced. Two styles:
  - **Heterogenous SDR:** offload transmit bit-level processing and simple DSP (e.g. FFT) to specialized engines – *lower MHz, better power*
  - **All-In-One SDR:** do as much as possible on one core – *simpler programming, more flexibility*



Streamlined compute scenario

# Simple Code Example: 4x4 Complex Matrix Mul

Scalar C code (with DSP-extended scalar types – e.g. complex fractions):

```
static xb_cq15 a1[4][4][NSAMPLES];
static xb_cq15 b1[4][4][NSAMPLES];
static xb_cq15 c1[4][4][NSAMPLES];
void mm_auto_opt_4x4_stream_complex () { int i, j, h;
  for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j+=3) {
      for (h = 0; h < NSAMPLES; h++) {
        c1[i][j][h] = (xb_cq4_15)(a1[i][0][h] * b1[0][j][h]) + (xb_cq4_15)(a1[i][1][h] * b1[1][j][h]) +
          (xb_cq4_15)(a1[i][2][h] * b1[2][j][h]) + (xb_cq4_15)(a1[i][3][h] * b1[3][j][h]);
        c1[i][j+1][h] = (xb_cq4_15)(a1[i][0][h] * b1[0][j+1][h]) + (xb_cq4_15)(a1[i][1][h] * b1[1][j+1][h]) +
          (xb_cq4_15)(a1[i][2][h] * b1[2][j+1][h]) + (xb_cq4_15)(a1[i][3][h] * b1[2][j+1][h]);
        c1[i][j+2][h] = (xb_cq4_15)(a1[i][0][h] * b1[0][j+2][h]) + (xb_cq4_15)(a1[i][1][h] * b1[1][j+2][h]) +
          (xb_cq4_15)(a1[i][2][h] * b1[2][j+2][h]) + (xb_cq4_15)(a1[i][3][h] * b1[2][j+2][h]);
        c1[i][j+3][h] = (xb_cq4_15)(a1[i][0][h] * b1[0][j+2][h]) + (xb_cq4_15)(a1[i][1][h] * b1[1][j+2][h]) +
          (xb_cq4_15)(a1[i][2][h] * b1[2][j+2][h]) + (xb_cq4_15)(a1[i][3][h] * b1[2][j+2][h]);
      }
    }
  }
}
```

Inner loop compiler-generated code with code: vectorization, software pipelining and op-merging:

```
loopgtz a4,.LBB34_mm_auto_opt_4x4_stream_complex
{bbe_lv32x16s.ip v0,a2,512 nop bbe_mula32x18cpackq v5,v11,v0 bbe_mula32x18cpackq v6,v15,v0}
{bbe_lv32x16s.i v0,a2,1536 bbe_lv32x16s.i v3,a2,3584 bbe_mul32x18cpackq v1,v8,v0 bbe_mul32x18cpackq v2,v12,v0}
{bbe_lv32x16s.i v0,a2,5632 bbe_lv32x16s.ip v4,a2,512 bbe_mula32x18cpackq v1,v9,v0 bbe_mula32x18cpackq v2,v13,v0}
{bbe_lv32x16s.i v3,a2,1536 bbe_lv32x16s.i v7,a2,3584 bbe_mula32x18cpackq v1,v10,v3 bbe_mula32x18cpackq v2,v14,v3}
{bbe_sv32x16s.ip v5,a3,512 bbe_lv32x16s.i v0,a2,5632 bbe_mula32x18cpackq v1,v11,v0 bbe_mula32x18cpackq v2,v15,v0}
{nop bbe_sv32x16s.i v6,a3,1536 bbe_mul32x18cpackq v5,v8,v4 bbe_mul32x18cpackq v6,v12,v4}
{bbe_sv32x16s.ip v1,a3,512 nop bbe_mula32x18cpackq v5,v9,v3 bbe_mula32x18cpackq v6,v13,v3}
{bbe_sv32x16s.i v2,a3,1536 nop bbe_mula32x18cpackq v5,v10,v7 bbe_mula32x18cpackq v6,v14,v7}
```

# Simple Code Example: 2x2 Real Matrix Mul

## 16 packed matrices @ 8 multiplies every 3 cycles

### Length-independent Vector C code

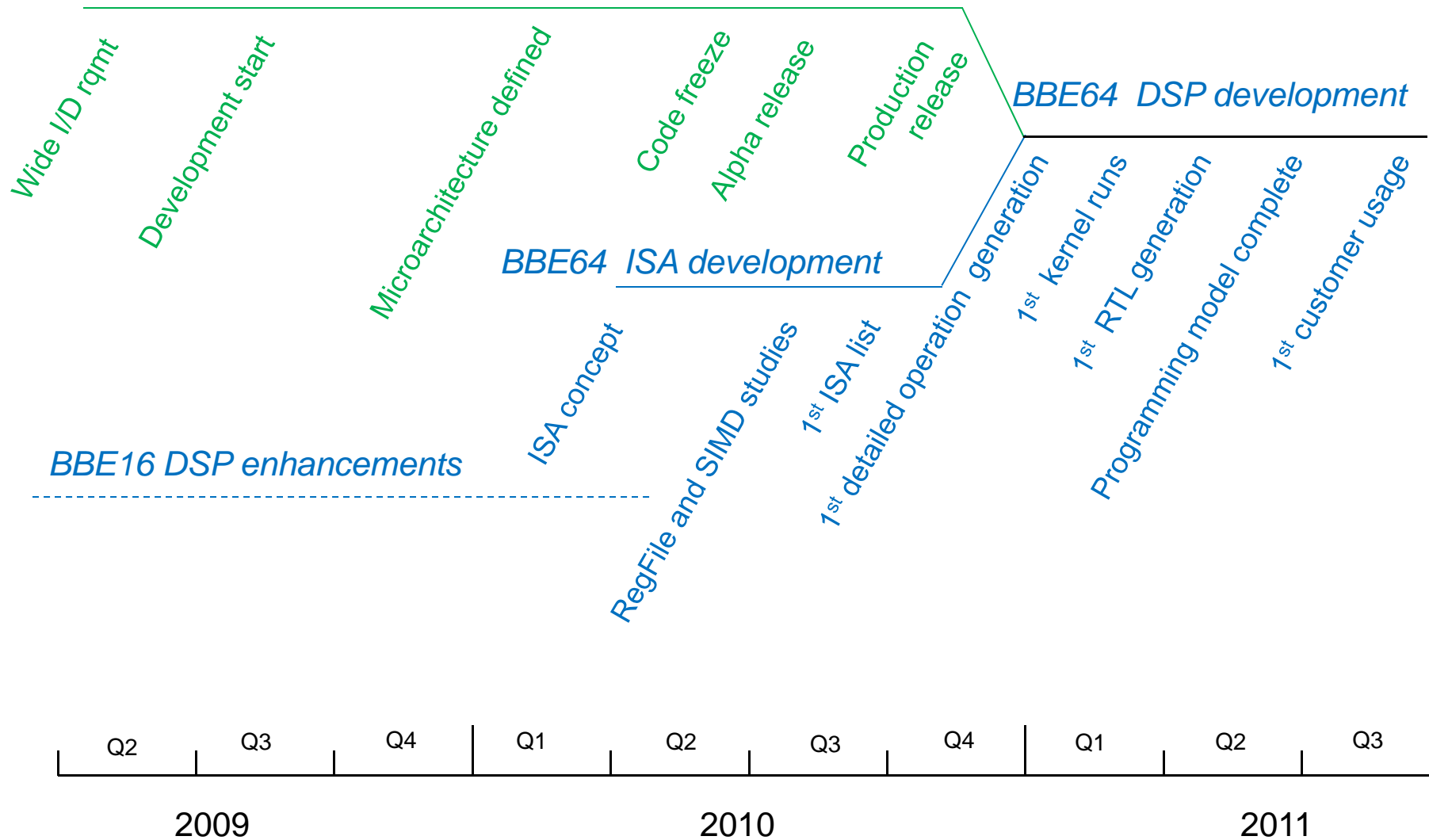
```
for (i = 0; i < n*4/BBE_SIMD; ++i) {  
    BBE_SELMMNXQ4_15(select_vec1,select_vec2,coeff[i],data[i],data[i],BBE_SELIMMR_2X2_X_2X2_STEP_1);  
    vout_temp = BBE_MULNXQ4_15PM(select_vec1, select_vec2, MATMUL_REAL_REPLICATION_INDEX_2);  
    out_p[i] = vout = BBE_PACKNXQ9_30Q(vout_temp);  
}
```

### Inner loop compiler-generated code with code: vectorization, software pipelining and op-merging:

```
loopgtz a4,.LBBx  
{bbe_lv32x16s.ip v0,a2,64          bbe_lv32x16s.ip v1,a3,64  nop          nop}  
{bbe_sv32x16packq.ip wv0,wv1,a9,64 bbe_lv32x16s.ip v4,a2,64 bbe_selmm32x20r v5,v2,v5,v4,v4,0 bbe_mul32x18pm wv0,wv1,v2,v3,2}  
{bbe_sv32x16packq.ip wv2,wv3,a9,64 bbe_lv32x16s.ip v5,a3,64 bbe_selmm32x20r v2,v3,v1,v0,v0,0 bbe_mul32x18pm wv2,wv3,v5,v2,2}
```

# BBE64 Development Timeline

*Xtensa LX4 base technology development*

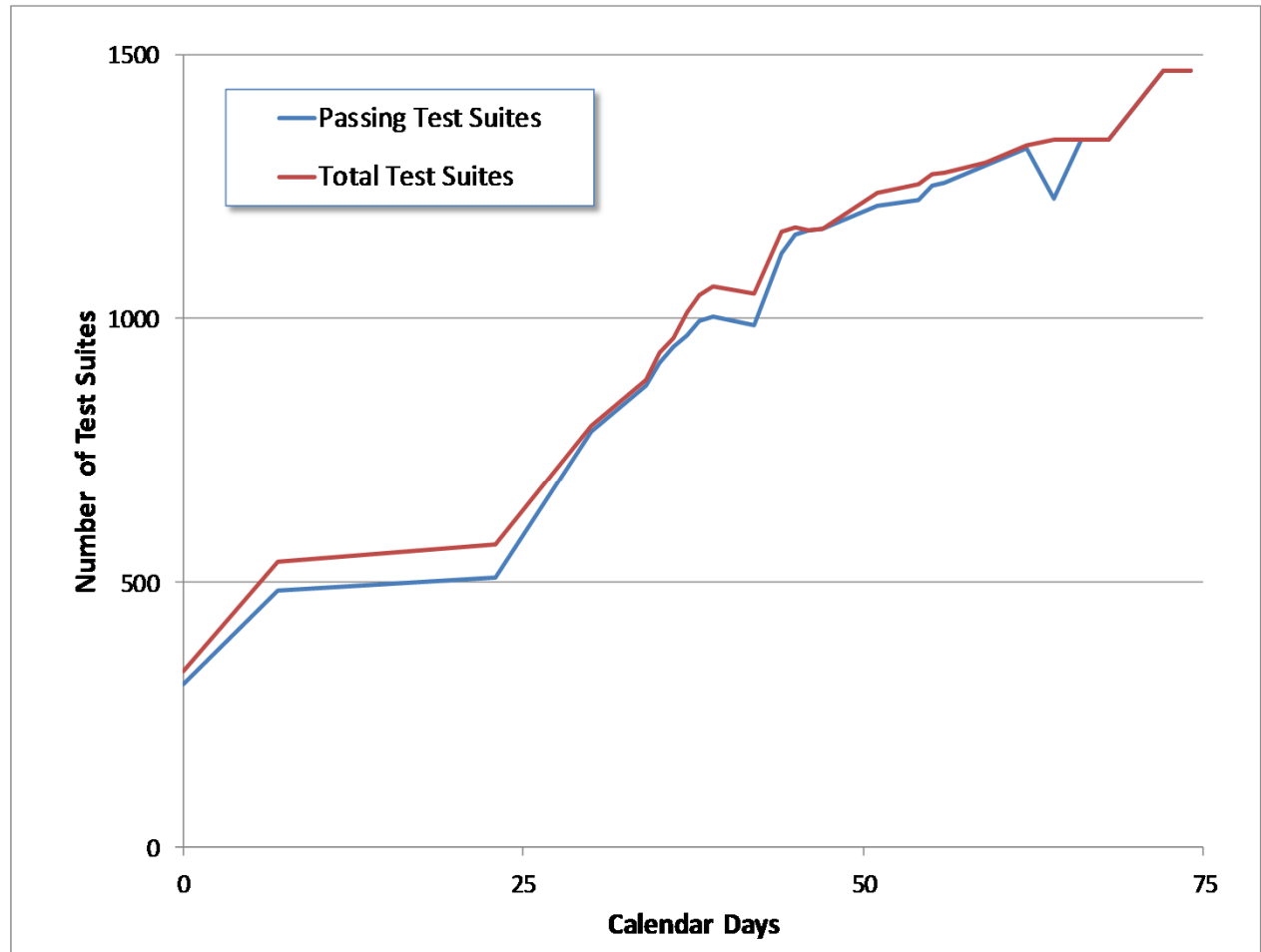


# BBE64 Verification Convergence

1400 test suites @ 3000 tests per suite



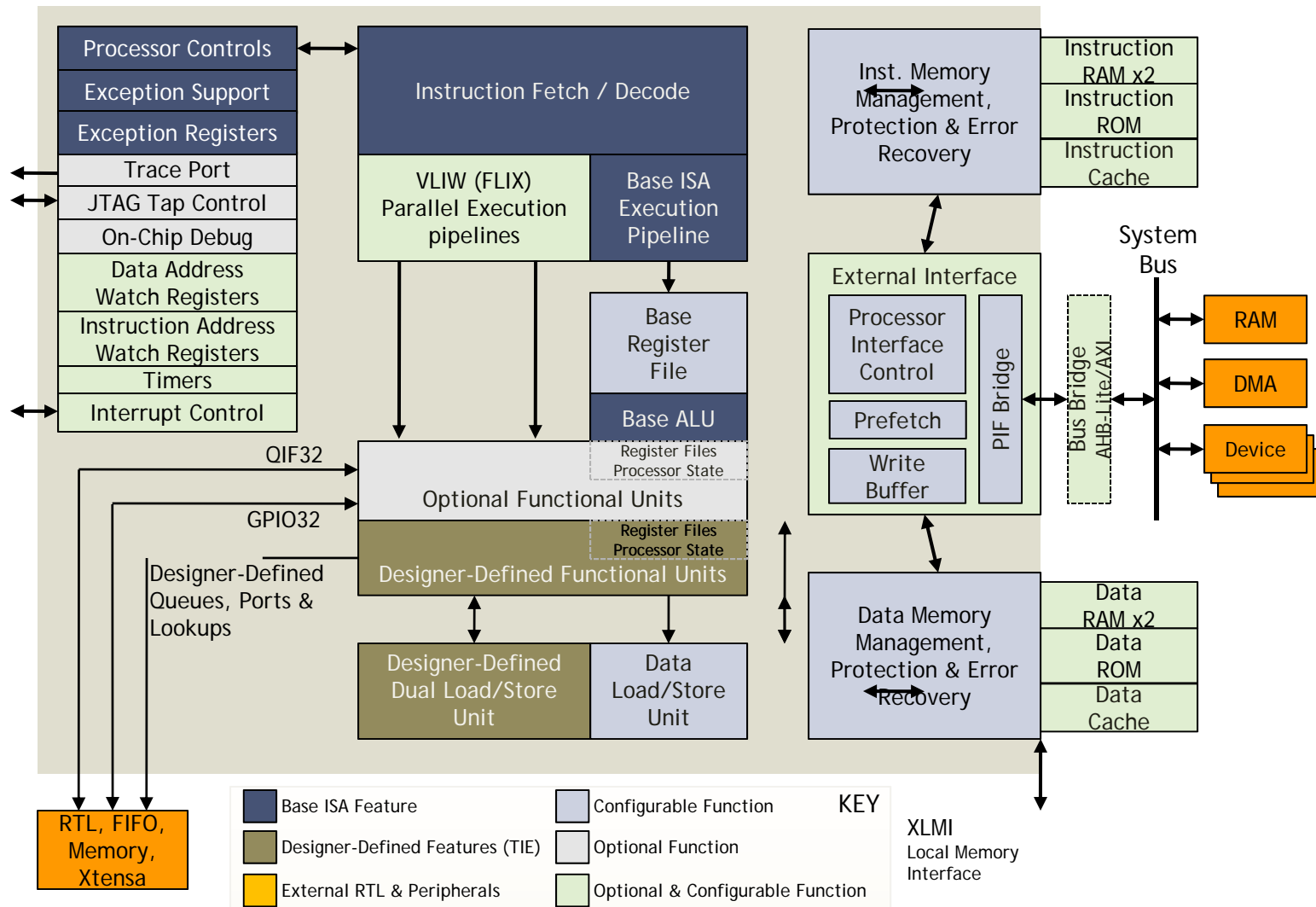
- Average of 3K directed and random data sets per test suite.
- Several test suites per instruction
- Tests and C reference developed independent of implementation
- Tools guarantee orthogonality of instruction implementations to reduce instruction interaction testing
- Many tests adapted from BBE16



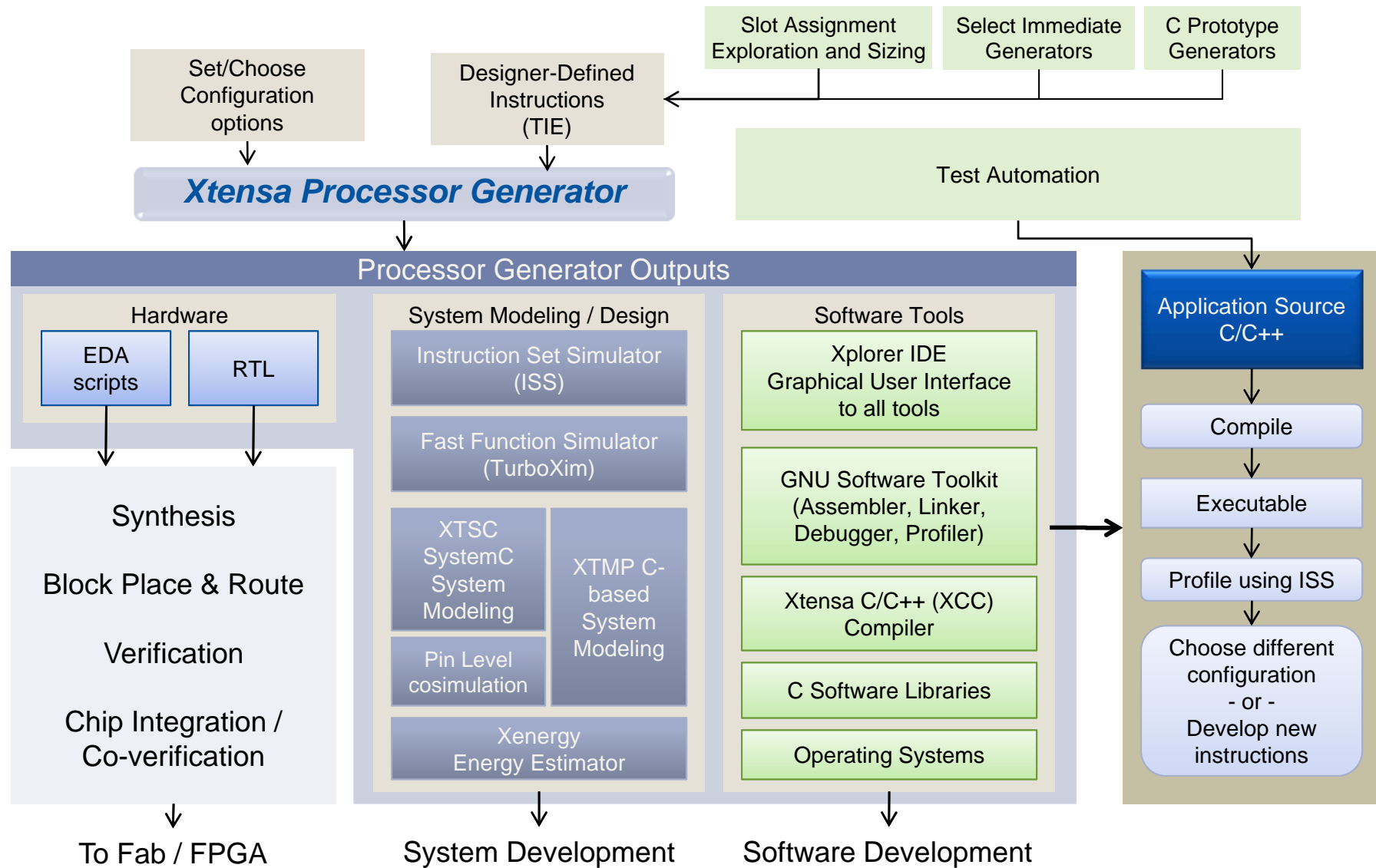
# Configurability and Extensibility



## LX4 Block Diagram



# Tools for DSP Automation



## Wrap-up

---

- The BBE64 kit is in the hands of lead customers— expect silicon in 2012
- Processor automation tools reduced cost of DSP development, especially in enabling rapid exploration of HW cost vs. SW benefit for many options
- Achieving a balanced design still required
  1. Broad algorithm analysis - BBE16 was good foundation
  2. Dramatic upgrades in infrastructure (wider instruction and data, regfile optimization)
- Most consciously scalable processor design in Tensilica history – enables easy generation of HW, testing, software development tools, DSP software across wide range of configurations and vector lengths
- Pushes the envelope on VLSI design – significant impact and accommodation of
  - High instruction set complexity
  - High fan-out in large data-path
  - Proving ground for exploitation of 28nm