Xtensa
A new ISA and Approach

Tensilica: www.tensilica.com
Earl Killian: www.killian.com/earl
Presentation Goals

- How Tensilica and Xtensa came to be
- What Xtensa is, with motivation for the decisions we made
  - Historical approach
- Get you thinking about a new paradigm
  - How do application-specific processors change the game?

- What are you interested in hearing about?
My Background

- **Major Projects**
  - 2 operating systems (not Unix)
  - 3 compilers (not gcc)
  - 1 satellite network
  - 4 processor instruction set designs
  - 6 processor micro-architectures

- **Places**
  - 1 University
  - 3 Start-ups (founder of one)
  - 1 Government lab
  - 2 Medium-sized companies
Outline

- About Tensilica
  - History, getting started, etc.
- Application-Specific Processors
  - What’s different
- Xtensa ISA
  - What we did and why
- Extensibility via the TIE (Tensilica Instruction Extension) Language
Tensilica Background

- Tensilica is the brainchild of Chris Rowen
  - founder and CEO
  - formerly Intel, Stanford, MIPS, sgi, and Synopsys
  - an idea that wouldn’t leave him alone: configurable processors

<table>
<thead>
<tr>
<th>1997</th>
<th>1998</th>
<th>1999</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>idea</td>
<td>exploration</td>
<td>open office</td>
<td>full selling</td>
</tr>
<tr>
<td>try snps</td>
<td></td>
<td>build team</td>
<td>2.0 development</td>
</tr>
<tr>
<td></td>
<td></td>
<td>plan</td>
<td>3.0 developer</td>
</tr>
<tr>
<td>Founded</td>
<td>Early Team</td>
<td>initial development</td>
<td>first customer</td>
</tr>
<tr>
<td>$2.3M A round</td>
<td>$10.6M B round</td>
<td>Xtensa 1.0</td>
<td>Xtensa 1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>trial selling</td>
<td>$20M C round</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Xtesna 2.0</td>
</tr>
</tbody>
</table>
Outline

- About Tensilica
  - History, getting started, etc.
- Application-Specific Processors
  - What’s different
- Xtensa ISA
  - What we did and why
- Extensibility via the TIE (Tensilica Instruction Extension) Language
Tensilica’s Mission

➢ From an early corporate overview:
   To be the leading provider of application-specific microprocessor solutions by delivering configurable, ASIC-based cores and matching software development tools.

➢ Therefore
   • Synthesizable, configurable, embedded processors
     – Application is known at ASIC-design time!
     – Key is to exploit application specificity
   • Compiler and OS are as important as the processor
   • Customers are system designers
     – Very cost conscious customers — will only pay for what they need
The Opportunity

- A choice between hard-wired, more optimized and softer, more flexible implementations
  - Intensive optimization is a bet on past knowledge, stable standards and predictable markets
  - Flexible design is a bet on future learning and unpredictable markets
- Sometimes, you can get ~best of both

Optimality/integration
(e.g. mW, $)

Optimality/integration
(e.g. mW, $)

Flexibility/modularity
(e.g. time-to-market)

special hardware

configurable processors + SW

FPGAs traditional processors + SW

Δ > 10^2

Δ > 10^2

24 February 2000
Not the Desktop Model

Intel Pentium III (~100mm² in 0.18µ)

Typical Xtensa processor (~2mm² in 0.18µ)

100x lower cost and power

20x lower parts count

10x lower system price

onto system-on-a-chip IC

into handheld appliance

onto system board

into PC box

processor IC
Technology Vision

Select processor options and describe new instructions in Web interface.

Using the Xtensa processor generator, create...

Tailored, HDL uP core

Customized Compiler, Assembler, Linker, Debugger, Simulator

Use standard library to target to the silicon
Types of Configurability

- **Quantity, size, etc.**
  - Often significant payback (e.g. cache size)

- **Options** (sort of quantity 0 or 1)
  - e.g. FP or not, MMU or not, DSP or not, ...

- **Parameters**
  - e.g. addresses of vectors, memories, ...

- **Target specifications**
  - e.g. synthesize for area at the cost of speed
  - Many applications don’t need the maximum processor performance
  - Process, standard cell library, etc.

- **Extensibility**
  - Adding things that the component supplier didn’t explicitly offer
Sample Xtensa Configurability

- **Cost, Power, Performance**
- **ISA**
  - Endianness
  - MUL16/MAC16
  - Various miscellaneous instructions
- **Interrupts**
  - Number of interrupts
  - Type of interrupts
  - Number of interrupt levels
  - Number of timers and their interrupt levels
  - more...
- **Memories**
  - 32 or 64 entry regfile
  - 32, 64, or 128b bus widths
  - Inst Cache
    - 1KB to 16KB
    - 16, 32, or 64B line size
  - Data Cache/RAM
    - ditto
  - 4-32-entry write buffer
- **Debugging**
  - No. inst addr breakpoints
  - No. data addr breakpoints
  - JTAG debugging
  - Trace port
Example .25μ Results

- 55 to 141MHz
- 28 to 84K gates
- 62 to 191mW power
- 2.0mm² to 8.3mm² including cache RAMs
Outline

- About Tensilica
  - History, getting started, etc.
- Application-Specific Processors
  - What’s different
- Xtensa ISA
  - What we did and why
- Extensibility via the TIE (Tensilica Instruction Extension) Language
Early Planning

- Product/ISA discussion started ≈3/1998
  - Do our own ISA or MIPS/ARM?
  - What do we optimize for (performance, cost, code size, etc.)?
  - How low-end do we go (e.g. 16-bit)?
  - If our own ISA, do we need an “on-ramp”? 
  - How much DSP?

- Issues
  - Only 8 months planned to do first product!
  - Legal issues using another’s ISA
  - Many standard processor tricks unavailable in synthesizable logic
Our Guess at Our Customers’ Priorities

- **Solution**
- **System (not processor) cost**
  - processor die area
  - code size
  - power
- **Time-to-market**
  - ease of use
  - verification
  - debugging
- **Energy efficiency**
- **Performance**
- **Compatibility**
Our Resulting ISA Priorities

- **Code size**
  - largest factor in system cost
- **Configurability, Extensibility**
  - provides best match to customer requirements, and so optimizes system cost
- **Processor cost**
  - a small factor in system cost
- **Energy efficiency**
  - minor influence on ISA, but listed for when it matters
- **Performance**
  - when all else is equal, this becomes important
- **Scalability**
- **Features**
The Importance of Code Size

Area vs. Program Instructions

- Based on base 0.18µm implementation plus code RAM or cache
- Xtensa code ~10% smaller than ARM9 Thumb, ~50% smaller than MIPS-Jade, ARM9 and ARC
- ARM9-Thumb has reduced performance
- RAM/cache density = 8KB/mm²

24 February 2000
ISA Process

- Micro-architecture was firmer than ISA
- Created/circulated ISA alternatives
- Lots of arguing over alternatives
- Some data collected (but not much time!)
  - code size
  - performance
- Generally converged on solutions by consensus
- Generally followed our priority list
ISA Influences

- Major ISAs that influenced Xtensa
  - MIPS (e.g. compare-and-branch, MDMX, MIPS V)
  - IBM Power (ISA aids for ifetch, address modes)
  - Sun SPARC (register windows)
  - ARM Thumb (code size)
  - HP Playdoh (speculative loads)
  - DSPs (loop instructions)

- Other ISAs that shaped my thinking
  - CDC 6600, Cray-1
  - DEC PDP10
  - DEC PDP11, Motorola 68000
  - Multics, LLNL S-1, S-2
  - Cydrome, Multiflow
Target Pipeline

- One clock, rising-edge triggered flip-flops
  - no time borrowing between stages
- Use RAM-compiler generated Instruction and Data RAMs
  - registered address input
Pipeline Issues

- **Why not superscalar?**
  - Cost/benefit not right for this market
    - 2× register file read and write ports
    - Typical dual-issue adds 20-30% performance boost, not 2×
  - Design/verification time
  - Balance
    - Should add branch prediction or branches cost too much

- **Why 5-stage (1980’s RISC in 2000)?**
  - Cycle time cost too high for < 5 stages
  - Energy and cost issues for > 5 stages
Pipeline Implications

- **Branches will be expensive**
  - lack of time borrowing, edge-triggered RAM
  - try to compensate in ISA with more powerful branches

- **Symmetry of I and M stages allows time for variable length instruction alignment**

- **Standard RISC principles:**
  - Instructions must be simple to decode, issue, bypass
  - Register file read addresses must from fixed instruction fields
Early Controversies

- Performance/scalability vs. code size
- Multiple instruction sizes and instruction ≠ 32b
- Register windows
- How to handle the small size of immediate operands
- Instruction mnemonics
- DSP
Performance vs. Code Size

- **Traditional performance-oriented ISA**
  - Fixed 32b instruction word
    - supports 3 or 4 5-6b register fields
    - supports easy superscalar growth path

- **Code-size oriented ISA**
  - Most instructions < 32b (usually 16b)
    - 2 or 3 3-4b register fields (extra spills or moves)
  - Multiple instruction sizes
    - superscalar more difficult

- **Considered 32/16, 24/12, and 24/16**
  - Two sizes differentiated by a single bit

- **Tensilica chose 24/16 in line with our priorities**
  - best code size of the choices
  - good performance from 3 4b register fields
Register Windows

- Code size savings from elimination of save/restore
  - savings very application dependent
  - our estimate was 6-10%

- Issues
  - larger register file (adds to processor area)
    - especially with standard cell implementation
  - may impact real-time applications
  - windows not well-liked (colored by SPARC)

- Tensilica chose windows as per our priorities
  - fixed SPARC problems
# Xtensa Instruction Formats

<table>
<thead>
<tr>
<th>op2</th>
<th>op1</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>op0</th>
</tr>
</thead>
</table>

- **E.g.** \( \text{AR}[r] \leftarrow \text{AR}[s] + \text{AR}[t] \)

<table>
<thead>
<tr>
<th>imm8</th>
<th>op1</th>
<th>s</th>
<th>t</th>
<th>op0</th>
</tr>
</thead>
</table>

- **E.g.** if \( \text{AR}[s] < \text{AR}[t] \) goto PC+imm8

<table>
<thead>
<tr>
<th>imm12</th>
<th>s</th>
<th>t</th>
<th>op0</th>
</tr>
</thead>
</table>

- **E.g.** if \( \text{AR}[s] = 0 \) goto PC+imm12

<table>
<thead>
<tr>
<th>imm16</th>
<th>t</th>
<th>op0</th>
</tr>
</thead>
</table>

- **E.g.** \( \text{AR}[t] \leftarrow \text{AR}[t] + \text{imm16} \)

<table>
<thead>
<tr>
<th>imm18</th>
<th>n</th>
<th>op0</th>
</tr>
</thead>
</table>

- **E.g.** CALL0 PC+imm18

<table>
<thead>
<tr>
<th>r</th>
<th>s</th>
<th>t</th>
<th>op0</th>
</tr>
</thead>
</table>

- **E.g.** \( \text{AR}[r] \leftarrow \text{AR}[s] + \text{AR}[t] \)
Code Size

- **Bits per instruction reduction (0.62)**
  - 24-bit encoding (25%)
  - 16-bit optional encodings (12%)

- **Instruction count (0.91)**
  - Compound instructions
    - 15% from compare-and-branch
    - 2% from shift add/subtract
    - 2% from shift mask (extract)
    - 2% from L32R vs. 2-instruction 32-bit immediate synthesis
  - Register windows
    - 6% from elimination of functional call overhead (save/restore)
  - 24-bit encoding
    - 10% from register spill
    - 8% from small immediates

- **Combined 0.91 \times 0.62 = 0.56**
## Code Size Comparison — ARM

```c
for (i=0; i < NUM; i++)
    if (histogram[i] != NULL)
        insert (histogram[i], &tree);
```

<table>
<thead>
<tr>
<th>Xtensa code</th>
<th>ARM code</th>
<th>Thumb code</th>
</tr>
</thead>
<tbody>
<tr>
<td>L16: addx4 a2, a3, a5 132i a10, a2, 0 beqz a10, L15 add a11, a4, a7 cal18 insert</td>
<td>J4:ADD a1,sp,#4 LDR a1,[a1,a3,LSL#2] CMP a1,#0 MOVNE a2,sp BLNE insert ADD a3,a3,#1 CMP a3,#&amp;3e8 BLT J4</td>
<td>L4: LSL r1,r7,#2 ADD r0,sp,#4 LDR r0,[r0,r1] CMP r0,#0 BEQ L13 MOV r1,sp BL insert L13:ADD r7,#1 CMP r7,r4 BLT L4</td>
</tr>
<tr>
<td>L15: addi a3, a3, 1 bge a6, a3,L16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **7 instructions** 17 bytes
- **8 instructions** 36 bytes
- **10 instructions** 20 bytes
Xtensa ISA Summary

- **80 base instructions**
  - Load and Store (8 instructions)
  - Move (5 instructions)
  - Shift (13 instructions)
  - Arithmetic Operations (12 instructions)
  - Logical Operations (AND, OR, XOR)
  - Jump and Branch (29 instructions)
  - Zero Overhead Loops (3 instructions)
  - Pipeline Control (7 instructions)
## Xtensa ISA Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Code size</th>
<th>Energy efficiency</th>
<th>Performance</th>
<th>Extensibility</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>24-bit encoding</td>
<td>3</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>16-bit encoding</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register windows</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compare and branch</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit test/mask and branch</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No branch delay</td>
<td>3</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Funnel shifts</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Right shift and mask</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional moves</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speculative loads</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-overhead loop</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIE</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Multiprocessor</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSP option</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP option</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

24 February 2000
Compare and Branch

```c
if (a < b) {
    c = 0;
}
```

**SPARC**
```
cmp   %o0, %o1
bge   L1
<<delayslot>>
or    %g0, 0, %o2
L1:
```

2 cycle branch untaken or taken
(3 if nop in delay slot)

**Xtensa**
```
bge   a2, a3, L1
movi  a4, 0
L1:
```

1 cycle branch if untaken,
3 cycle branch if taken
Zero-Overhead Loops

```assembly
loopgtz a0, endloop
loop:
  body0
  ...
  ...
  bodyN
endloop:
```

- Processor automatically branches to body0 after executing bodyN the number of times in a0
- No branch penalty in most cases
- Implemented with the LBEG, LEND, and LCOUNT special registers
Overlapped Register Windows

- Routine F calls routine G incrementing register file pointer by 4, 8, or 12
- F and G’s windows into the physical register file overlap
- F can pass register parameters to G by writing its high registers
- The register file pointer increment hides 4-12 of F’s registers
- No save or restores required unless pointer wraps
Window Code Example

Foo:
entry sp, 16
movi a6, 1 // a6 will become a2 in Bar after entry
l32i a7, a2, 4 // a7 will become a3 in Bar after entry
call4 Bar // call Bar, request increment of 4
addi a2, a6, 1 // a6 is Bar’s a2 before the retw
retw

Bar:
entry sp, 16 // move window by caller’s increment
add a2, a2, a3 // add our arguments, with result
              // to return value register
retw // move window back (decrement)
Window Code Comparison

**Traditional**

\[
\begin{align*}
&f: \text{addi } sp, sp, -\text{framesize} \\
&s32i \ a0, \ framesize-12(sp) \\
&s32i \ a12, \ framesize-8(sp) \\
&s32i \ a13, \ framesize-4(sp) \\
&\ldots \\
&132i \ a0, \ framesize-12(sp) \\
&132i \ a12, \ framesize-8(sp) \\
&132i \ a13, \ framesize-4(sp) \\
&\text{addi } sp, sp, \ framesize \\
&\text{ret}
\end{align*}
\]

**With Windows**

\[
\begin{align*}
&f: \text{entry } sp, \ framesize \\
&\ldots \\
&\text{retw} \\
&\quad \text{Smaller} \\
&\quad \text{Faster}
\end{align*}
\]
Outline

- About Tensilica
  - History, getting started, etc.
- Application-Specific Processors
  - What’s different
- Xtensa ISA
  - What we did and why
- Extensibility via the TIE (Tensilica Instruction Extension) Language
Productivity Gap

58%/Yr. complexity growth rate

21%/Yr. Productivity growth rate

Logic Transistor / Chip (K)
Transistor/Staff-month

Source: NTRS'97
TIE Overview

Configure Base uP

Processor Generator

Software Generator

Processor Verilog RTL

Software Tools

ASIC flow

uP

Mem

Software compile

Describe new inst in TIE

Application

24 February 2000
TIE Design Cycle

Develop application in C/C++

Profile and analyze

Id potential new instructions

Describe new instructions

Generate new software tools

Compile and run application

Correct ?

Run cycle-accurate ISS

Acceptable ?

Measure hardware impact

Acceptable ?

Build the entire processor

24 February 2000
Tensilica Instruction Extension

- **No micro-architecture (implementation) details**
  - same TIE will work with new base
  - decode, interlock, bypass, and pipelining automatic

- **Automatic configuration of software tools**
  - compiler
  - instruction-set simulator
  - debugger
  - etc.

- **Automatic synthesis of efficient hardware compatible with the base processor**

- **Extension language, not a language to describe a complete CPU**
Major sections in TIE

- Instruction fields
- Opcode
- Operands
- Instruction semantics
Instruction Field Definition

- **TIE code:**
  - field op0 Inst[3:0]
  - field op1 Inst[19:16]
  - field op2 Inst[23:20]
  - field r Inst[15:12]
  - field s Inst[11:8]
  - field t Inst[7:4]

```plaintext
+---+---+---+---+---+---+
<table>
<thead>
<tr>
<th>op2</th>
<th>op1</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>op0</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>Inst</td>
</tr>
</tbody>
</table>
```
 Opcode Definition

- **TIE code:**
  
  - Opcode `QRST` \( \text{op0}=4'b0000 \)
  
  - Opcode `CUST0` \( \text{op1}=4'b1100 \) \( \text{QRST} \)
  
  - Opcode `ADD4` \( \text{op2}=4'b0000 \) \( \text{CUST0} \)

- **TIE compiler generates decode logic**
Operand Definition

- **TIE code:**
  ```
  operand ars s {AR[s]}
  operand art t {AR[t]}
  operand arr r {AR[r]}
  iclass rrr {ADD4}{out arr, in ars, in art}
  ```

- **Assembly example:**
  ```
  ADD4 a2, a3, a5
  ```

- **C example:**
  ```
  X = ADD4(y, z);
  ```

- **TIE compiler generates interlock and bypass logic**

---

<table>
<thead>
<tr>
<th>0000</th>
<th>1101</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>0000</th>
</tr>
</thead>
</table>

ADD4 Instruction

![Diagram showing the ADD4 Instruction](image)
Semantic Description

➢ TIE code:

```plaintext
semantic add4_semantic {ADD4} {
    wire [7:0] arr0 = ars[7:0] + art[7:0];
    wire [7:0] arr1 = ars[15:8] + art[15:8];
    wire [7:0] arr3 = ars[31:24] + art[31:24];
    assign arr = {arr3, arr2, arr1, arr0};
}
```

ADD4 Instruction

```
0000 1101 r s t 0000
```

```
wa ra0 ra1
```

```
wdrd0 rd1
```

```
ars art
```

```
+
```

```
arr
```

24 February 2000
Complete Example

opcode ADD4  op2=4'b0000  CUST0
iclass rrr  {ADD4}  {out arr, in ars, in art}
semantic add4_semantic {ADD4} {
    wire arr0 = ars[ 7: 0] + art[ 7: 0];
    wire arr1 = ars[15: 8] + art[15: 8];
    wire arr2 = ars[23:16] + art[23:16];
    wire arr3 = ars[31:24] + art[31:24];
    assign arr = {arr3, arr2, arr1, arr0};
}
TIE Development Process

TIE Description

Native C stubs

cc.so

ISS.so

TIE.v

Software tools

ISS

Xtensa RTL

TIE Development Kits
#ifdef NATIVE
#include ADD4_cstub.c
#endif

int a[], b[], c[];
char *x=a, *y=b, *z=c;
...
read(x);
read(y);
for (i = 0; i < n; i++) {
    c[i] = ADD4(a[i], b[i]);
}
write(z);
...
Testing new instructions on the host

```plaintext
shell> gcc -o app -DNATIVE app.c
shell> app
```

- **Objectives**
  - Verify TIE description
  - Verify application code
- **Advantage**
  - Short iteration cycle
Testing new instructions on Xtensa simulator

shell> xt-gcc -o app app.c
shell> iss app

- Objectives
  - Testing TIE description
  - Testing application
  - Measuring performance

- Advantage
  - Cycle-accurate
Checking the Hardware

shell> vi app.dcsh
shell> dc_shell -f app.dcsh
shell> vi app.report

- Objectives
  - Measuring cycle-time impact
  - Measuring area impact
- Advantage
  - Time-accurate
  - Cost-accurate
Data Encryption Standard

- **Initial step**
  \((R, L) = \text{Initial}_\text{permutation}(Din_{64})\)

- **Iterate 16 times**
  - **Key generation**
    \((C, D) = PC1(k)\)
    \(n = \text{rotate}_\text{amount} \text{ (function of iteration count)}\)
    \(C = \text{rotate}_\text{right}(C, n)\)
    \(D = \text{rotate}_\text{right}(D, n)\)
    \(K = PC2(D, C)\)
  - **Encryption**
    \(R_{i+1} = L_i \oplus \text{Permutation ( S_Box ( K \oplus \text{Expansion ( R )} ))} \)
    \(L_{i+1} = R_i\)

- **Final step**
  \(Dout_{64} = \text{Final}_\text{permutation}(L, R)\)
static unsigned permute(unsigned char *table, int n, unsigned hi, unsigned lo)
{
    int ib, ob;
    unsigned out = 0;
    for (ob = 0; ob < n; ob++) {
        ib = table[ob] - 1;
        if (ib >= 32) {
            if (hi & (1 << (ib-32))) out |= 1 << ob;
        } else {
            if (lo & (1 << ib)) out |= 1 << ob;
        }
    }
    return out;
}

Too much computation!
Too slow!
DES Hardware Implementation

Initial Permutation → Expansion Permutation → S Boxes → P Permutation → Final Permutation → Key Generation

Complicated control logic! Too hard!
DES Implemented in TIE

Initial Permutation

Expansion Permutation

S Boxes

P Permutation

Final Permutation

Key Generation

State Machine

SETDATA  ars, art

GETDATA  ars, hilo

SETKEY  ars, art

DES  immediate
SETKEY(K_hi, K_lo);

for (;;) {
  ...
  SETDATA(D_hi, D_lo);
  DES(ENCRYPT1);
  DES(ENCRYPT1);
  DES(ENCRYPT2);
  DES(ENCRYPT2);
  DES(ENCRYPT2);
  DES(ENCRYPT2);
  DES(ENCRYPT2);
  DES(ENCRYPT2);
  DES(ENCRYPT1);
  DES(ENCRYPT2);
  E_hi = GETDATA(hi);
  E_lo = GETDATA(lo);
  ...
}

SETKEY(K_hi, K_lo);

for (;;) {
  ...
  SETDATA(D_hi, D_lo);
  DES(DECRYPT1);
  DES(DECRYPT1);
  DES(DECRYPT2);
  DES(DECRYPT2);
  DES(DECRYPT2);
  DES(DECRYPT2);
  DES(DECRYPT2);
  DES(DECRYPT2);
  DES(DECRYPT2);
  DES(DECRYPT1);
  E_hi = GETDATA(hi);
  E_lo = GETDATA(lo);
  ...
}
Triple DES Example

- **Application:**
  - Secure Shell Tools (SSH)
  - Internet Protocol for Security (IPSEC)

- **Add 4 TIE instructions:**
  - 80 lines of TIE description
  - No cycle time impact
  - ~1700 additional gates
  - Code-size reduced

![DES Performance Chart]
Result: Flexibility + Efficiency

Improvement in MIPS over general-purpose 32b RISC

- IP Routing: +8000 gates
- FIR Filter (telecom): +6500 gates
- JPEG (cameras): +7500 gates
- CDMA (wireless): +4000 gates
- DES Encryption (IPSEC, SSH): +4500 gates
- Viterbi Decoding (wireless): +9000 gates
- Motion Estimation (video): +30000 gates
Cost $<1,5 -100x speed-up

Application Speed-up over 32b RISC (18 examples)

- Cost = marginal cost for core+memory in 0.25µm foundry in volume
- Data from communication and consumer applications: FIR filter, Viterbi, DES, JPEG, Motion Estimation, W-CDMA, Packet Flow, RGB2CYMK, RGB2CYMK, RGB2YIQ, Grayscale Filter, Auto-Correlation,
A Common TIE Paradigm

Software: Control

Hardware: Computation

ALU

Initial Permutation

Expansion Permutation

S Boxes

P Permutation

⊕

Final Permutation

Key Generation

State Machine

24 February 2000
Summary continued

<table>
<thead>
<tr>
<th></th>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>hard</td>
<td>easy</td>
</tr>
<tr>
<td>Computation</td>
<td>easy</td>
<td>hard</td>
</tr>
</tbody>
</table>

Application-specific instructions
Conclusion

- Presentation
  - About Tensilica
  - Application-Specific Processors
  - Xtensa ISA
  - TIE

- Is there anything else you would like me to cover?