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

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

22 May 2000
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

Logic Transistor / Chip (K)

Transistor/Staff-month

Source: NTRS’97
A Brief Tour of History

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

$\Delta > 10^2$

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

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

special hardware

configurable processors + SW

FPGAs traditional processors + SW
The Vision

Select processor options

Using the Xtensa processor generator, create...

Tailored, synthesizable HDL uP core

Customized Compiler, Assembler, Linker, Debugger, Simulator

Describe new instructions

In Minutes!
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
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 Results

- **.25μ**
  - 56 to 141MHz
  - 30 to 119K gates
  - 54 to 237mW power
  - 1.7mm² to 42.4mm² including cache RAMs

- **.18μ**
  - 93 to 200MHz
  - 30 to 91K gates
  - 36 to 129mW power
  - 0.9mm² to 17.3mm² including cache RAMs
Sample Extensibility

- **Instruction formats**
  - Instruction fields
  - Opcodes
  - Operands
- **Processor states**
  - Register files
  - Special states
- **Instruction semantics**
  - Computation
- **Micro-architecture guidelines**
  - Multi-cycle instructions
  - Instruction timing
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

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
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
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

22 May 2000
Pipeline Issues

- **Why not superscalar?**
  - Cost/benefit not right for this market
    - $2 \times$ register file read and write ports
    - Typical dual-issue adds 20-30% performance boost, not $2 \times$
  - 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 an 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>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E.g. AR[r] ← AR[s] + AR[t]</td>
</tr>
<tr>
<td>imm8</td>
<td>op1</td>
<td>s</td>
<td>t</td>
<td>op0</td>
<td>E.g. if AR[s] &lt; AR[t] goto PC+imm8</td>
</tr>
<tr>
<td>imm12</td>
<td>s</td>
<td>t</td>
<td>op0</td>
<td>E.g. if AR[s] = 0 goto PC+imm12</td>
<td></td>
</tr>
<tr>
<td>imm16</td>
<td>t</td>
<td>op0</td>
<td></td>
<td>E.g. AR[t] ← AR[t] + imm16</td>
<td></td>
</tr>
<tr>
<td>imm18</td>
<td>n</td>
<td>op0</td>
<td></td>
<td>E.g. CALL0 PC+imm18</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r</th>
<th>s</th>
<th>t</th>
<th>op0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E.g. AR[r] ← AR[s] + AR[t]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

22 May 2000
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 × 0.62 = 0.56**
## Code Size Comparison — ARM

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

### Xtensa code

<table>
<thead>
<tr>
<th>L16: addx4 a2, a3, a5</th>
</tr>
</thead>
<tbody>
<tr>
<td>132i a10, a2, 0</td>
</tr>
<tr>
<td>beqz a10, L15</td>
</tr>
<tr>
<td>add a11, a4, a7</td>
</tr>
<tr>
<td>call8 insert</td>
</tr>
<tr>
<td>L15: addi a3, a3, 1</td>
</tr>
<tr>
<td>bge a6, a3,L16</td>
</tr>
</tbody>
</table>

- 7 instructions
- 17 bytes

### ARM code

<table>
<thead>
<tr>
<th>L4:ADD a1,sp,#4</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR a1,[a1,a3,LSL#2]</td>
</tr>
<tr>
<td>CMP a1,#0</td>
</tr>
<tr>
<td>MOVNE a2,sp</td>
</tr>
<tr>
<td>BLNE insert</td>
</tr>
<tr>
<td>ADD a3,a3,#1</td>
</tr>
<tr>
<td>CMP a3,#&amp;3e8</td>
</tr>
<tr>
<td>BLT J4</td>
</tr>
</tbody>
</table>

- 8 instructions
- 36 bytes

### Thumb code

<table>
<thead>
<tr>
<th>L4: LSL r1,r7,#2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD r0,sp,#4</td>
</tr>
<tr>
<td>LDR r0,[r0,r1]</td>
</tr>
<tr>
<td>CMP r0,#0</td>
</tr>
<tr>
<td>BEQ L13</td>
</tr>
<tr>
<td>MOV r1,sp</td>
</tr>
<tr>
<td>BL insert</td>
</tr>
<tr>
<td>L13:ADD r7,#1</td>
</tr>
<tr>
<td>CMP r7,r4</td>
</tr>
<tr>
<td>BLT L4</td>
</tr>
</tbody>
</table>

- 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)
**Compare and Branch**

C

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

**SPARC**

```c
cmp %o0, %o1
bge L1
<<delayslot>>
or %g0, 0, %o2
```

**Xtensa**

```c
bge a2, a3, L1
movi a4, 0
```

**L1:**

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

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
Xtensa MAC16 DSP Unit

Instruction Format:
MUL.xy.qq

40-bit register as the accumulation destination
# MAC16 Instruction Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDINC</td>
<td>RRR</td>
<td>Load MAC16 register rr, autoincrement</td>
</tr>
<tr>
<td>LDDEC</td>
<td>RRR</td>
<td>Load MAC16 register rr, autodecrement</td>
</tr>
<tr>
<td>UMUL.AA.qq</td>
<td>RRR</td>
<td>Unsigned multiply of AR[s] and AR[t]</td>
</tr>
<tr>
<td>MUL.AA.qq</td>
<td>RRR</td>
<td>Signed multiply of AR[s] and AR[t]</td>
</tr>
<tr>
<td>MUL.AD.qq</td>
<td>RRR</td>
<td>Signed multiply of AR[s] and MR[1][t2]</td>
</tr>
<tr>
<td>MUL.DA.qq</td>
<td>RRR</td>
<td>Signed multiply of MR[0][r2] and AR[t]</td>
</tr>
<tr>
<td>MULA.DA.qq</td>
<td>RRR</td>
<td>Signed multiply/accumulate of MR[0][r2] and AR[t]</td>
</tr>
<tr>
<td>MULA.DD.qq</td>
<td>RRR</td>
<td>Signed multiply/accumulate of MR[0][r2] and MR[1][t2]</td>
</tr>
<tr>
<td>MULS.AA.qq</td>
<td>RRR</td>
<td>Signed multiply/subtract of AR[s] and AR[t]</td>
</tr>
<tr>
<td>MULS.AD.qq</td>
<td>RRR</td>
<td>Signed multiply/subtract of AR[s] and MR[1][t2]</td>
</tr>
<tr>
<td>MULS.DA.qq</td>
<td>RRR</td>
<td>Signed multiply/subtract of MR[0][r2] and AR[t]</td>
</tr>
<tr>
<td>MULS.DD.qq</td>
<td>RRR</td>
<td>Signed multiply/subtract of MR[0][r2] and MR[1][t2]</td>
</tr>
<tr>
<td>MULA.DA.qq.LDINC</td>
<td>RRR</td>
<td>Signed multiply/accumulate of MR[0][r2] and AR[t], load MR[r1]</td>
</tr>
<tr>
<td>MULA.DD.qq.LDINC</td>
<td>RRR</td>
<td>Signed multiply/accumulate of MR[0][r2] and MR[1][t2], load MR[r1]</td>
</tr>
<tr>
<td>MULA.DA.qq.LDDEC</td>
<td>RRR</td>
<td>Signed multiply/accumulate of MR[0][r2] and AR[t], load MR[r1]</td>
</tr>
<tr>
<td>MULA.DD.qq.LDDEC</td>
<td>RRR</td>
<td>Signed multiply/accumulate of MR[0][r2] and MR[1][t2], load MR[r1]</td>
</tr>
</tbody>
</table>
FIR Filter with MAC16

Single sample FIR filter inner loop using MAC16

\[
\text{mula.dd.ll.ldinc } m0,a3,m0,m2 \quad \text{// } m0 = a[i+1]:a[i+0]; acc += a[i-4+1]*b[i-4+1]
\]

\[
\text{mula.dd.hh.ldinc } m2,a4,m1,m3 \quad \text{// } m2 = b[i+1]:b[i+0]; acc += a[i-4+2]*b[i-4+2]
\]

\[
\text{mula.dd.ll.ldinc } m1,a3,m1,m3 \quad \text{// } m1 = a[i+3]:a[i+2]; acc += a[i-4+3]*b[i-4+3]
\]

\[
\text{mula.dd.hh.ldinc } m3,a4,m0,m2 \quad \text{// } m3 = b[i+3]:b[i+2]; acc += a[i+0]*b[i+0]
\]

- 1 32-bit load per cycle instead of 2 16-bit loads per cycle
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
TIE Overview

Configure Base uP

*******
****
***********
***

Describe new inst in TIE

*******
****
*** ****
***

Application

Processor Generator

Processor Verilog RTL

Software Tools

ASIC flow

Software compile

uP

Mem
TIE Design Cycle

1. Develop application in C/C++
2. Profile and analyze
3. Identify potential new instructions
4. Describe new instructions
5. Generate new software tools
6. Compile and run application

Correct?

Run cycle-accurate ISS

Acceptable? [N]

Measure hardware impact

Acceptable? [Y]

Build the entire processor
Adding Instructions in Minutes!

- 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
- State and Register files
- Instruction semantics
- Compiler prototypes
- Pipelining
- Documentation
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]
  ```

  ![Diagram](image)
Opcode Definition

- **TIE code:**
  
  - opcode QRST op0=4'b0000
  - opcode CUST0 op1=4'b1100 QRST
  - opcode ADD4 op2=4'b0000 CUST0

- **TIE compiler generates decode logic**

<table>
<thead>
<tr>
<th>23</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1101</td>
</tr>
</tbody>
</table>

ADD4 Instruction
State Definition

- Storage used as implicit instruction operands
- TIE code:
  ```
  state    carry 1
  state    overflow 1
  state    roundmode 2
  state    acc 40
  user_register 12 {carry, overflow, roundmode}
  user_register 13 {acc[31:0]}
  user_register 14 {acc[39:32]}
  ```

- Assembly example:
  ```
  WUR      a2, ACCL
  WUR      a3, ACCH
  ```

- C example:
  ```
  WUR13(a[i]);
  ```

- TIE compiler generates RTOS context switch code
Register File Definition

- Storage used as explicit instruction operands
- TIE code:
  ```
  regfile datapreg 64 16
  ctype vec4x16 64 64 d
  ```
- Assembly example:
  ```
  ADD4 d2, d5, d10
  ```
- C example:
  ```
  vec4x16 *p, *q, scale;
  for (i = 0; i < N; i += 1) {
    p[i] = ADD4(q[i], scale);
  }
  ```
- TIE compiler generates RTOS context switch code
Operand Definition

- **TIE code:**
  
  ```
  operand ds s  {datareg[s]}
  operand dt t  {datareg[t]}
  operand dr r  {datareg[r]}
  iclass ddd  {ADD4}  {out dr, in ds, in dt}
  ```

- **Assembly example:**
  ```
  ADD4 d2, d3, d5
  ```

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

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

- **TIE code:**

```plaintext
semantic add4_semantic {ADD4} {
    wire [15:0] r0 = ds[15: 0] + dt[15: 0];
    wire [15:0] r1 = ds[31:16] + dt[31:16];
    assign dr = {r3, r2, r1, r0};
}
```

0000 1101  r  s  t  0000  ADD4 Instruction

```
```

- **ADD4 Instruction**

22 May 2000
Complete Example

regfile datareg 64 16
ctype vec4x16 64 64 d
operand ds s {datareg[s]}
operand dt t {datareg[t]}
operand dr r {datareg[r]}
opcode ADD4 op2=4'b0000 CUST0
iclass ddd {ADD4} {out dr, in ds, in dt}
semantic add4_semantic {ADD4} {
    wire r0 = ds[ 7: 0] + dt[ 7: 0];
    wire r1 = ds[15: 8] + dt[15: 8];
    wire r2 = ds[23:16] + dt[23:16];
    wire r3 = ds[31:24] + dt[31:24];
    assign dr = {r3, r2, r1, r0};
}
TIE Development Process

- TIE Description
- TIE Compiler
  - Native C stubs
  - cc.so
  - ISS.so
  - TIE.v
  - Software tools
  - ISS
  - Xtensa RTL
  - TIE Development Kits

22 May 2000
Using TIE Instruction in C

```c
#ifdef NATIVE
#include ADD4_cstub.c
#endif

vec4x16 a[ ], b[ ], c[ ];
...
for (i = 0; i < n; i++) {
    c[i] = ADD4(a[i], b[i]);
}
...
```
Testing New Instructions on the Host

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

- Objectives
  - Verify TIE description
  - Verify application code

- Advantage
  - Short iteration cycle
Testing New Instructions on the Xtensa Simulator

\texttt{shell> xt-gcc -o app app.c}
\texttt{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
Hardware Design Made Simple

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

Δ > 10^2

Δ > 10^2

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

special hardware

Application-specific processors + SW

FPGAs traditional processors + SW

DES
Data Encryption Standard

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

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

- **Final step**
  \(D_{out\ 64} = \text{Final}_\text{permutation}(L, R)\)
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;
}
DES Hardware Implementation

- Initial Permutation
- Expansion Permutation
- S Boxes
- P Permutation
- Final Permutation

Key Generation

State Machine

Complicated control logic! Too hard!
DES Implemented in TIE

SETDATA ars, art

Initial Permutation

Expansion Permutation

S Boxes

P Permutation

Final Permutation

SETKEY ars, art

Key Generation

State Machine

GETDATA ars, hilo

DES immediate

22 May 2000
DES Program

Encryption

SETKEY(K_hi, K_lo);
for (;;) {
    ... /* read data */
    SETDATA(D_hi, D_lo);
    DES(ENCRYPT1);
    DES(ENCRYPT1);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT2);
    DES(ENCRYPT1);
    DES(ENCRYPT1);
    E_hi = GETDATA(hi);
    E_lo = GETDATA(lo);
    ... /* write encrypted data */
}

Decryption

SETKEY(K_hi, K_lo);
for (;;) {
    ... /* read encrypted data */
    SETDATA(D_hi, D_lo);
    DES(DECRYPT1);
    DES(DECRYPT1);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT2);
    DES(DECRYPT1);
    DES(DECRYPT1);
    E_hi = GETDATA(hi);
    E_lo = GETDATA(lo);
    ... /* write data */
}
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**

![Graph showing DES performance with block sizes and speedup values.](image-url)
Software speedup made easy

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

Δ > 10^2

special hardware

Δ > 10^2

FFT

Application-specific processors + SW

FPGAs

traditional processors + SW

Flexibility/modularity
(e.g. time-to-market)
Inner Loop of FFT

- Complex input numbers: A, B
- Complex output numbers: R, S
- The bufferfly computation
  \((A, B) \Rightarrow (R, S)\)
- Detailed bufferfly computation
  \[
  \begin{align*}
  R_r &= A_r + B_r \\
  R_i &= A_i + B_i \\
  S_r &= (A_r - B_r) \cdot C_r - (A_i - B_i) \cdot C_i \\
  S_i &= (A_r - B_r) \cdot C_i + (A_i - B_i) \cdot C_r
  \end{align*}
  \]
Speedup FFT

- **Using RISC instructions would require**
  - 4 loads, 4 stores, ~12 ops
  - ~20 cycles per bufferfly

- **Room for speedup**
  - Use 128-bit load/store
  - 2 bufferflies in parallel
    \[(A_0,A_1,B_0,B_1) \Rightarrow (R_0,R_1,S_0,S_1)\]
  - Use buffer \((A_0',A_1',B_0',B_1')\) for parallel load and butterfly
  - Use buffer \((R_0',R_1',S_0',S_1')\) for parallel store and butterfly
FFT Implementation in TIE

LDBF1:
Load A0, A1
Compute R0r, S0r
Move R0r, R1r, S0r, S1r

LDBF2:
Load B0, B1
Compute R0i, S0i
Move R0i, R1i, S0i, S1i

STBF1:
Store R0, R1
Compute R1r, S1r

LDBF2:
Store S0, S1
Compute R1i, S1i
Move A0, A1, B0, B1
FFT-specific Instructions

- **Inner Loop:**
  - LDBF1
  - LDBF2
  - STBF1
  - STBF2

- **Speedup: 10x**
  - 20 cycles => 2 cycles

- **Hardware efficiency:**
  - 100% utilization of load/store unit
  - 100% utilization of 2 multipliers
Summary of Examples

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

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

Δ > 10^2

special hardware

Application-specific processors + SW

DES

Δ > 10^2

FPGAs

traditional processors + SW

FFT

Δ > 10^2

Application-specific processors + SW

FFT
Result: Flexibility + Efficiency

- **CDMA (wireless)**
  - Improvement in MIPS: 2x, 4x, 6x, 8x, 10x, 50x
  - +9000 gates
- **JPEG (cameras)**
  - +7500 gates
- **IP Routing**
  - +8000 gates
- **FIR Filter (telecom)**
  - +6500 gates
- **CDMA (wireless)**
  - +4000 gates
- **DES Encryption (IPSEC, SSH)**
  - +4500 gates
- **Viterbi Decoding (wireless)**
  - +9000 gates
- **Motion Estimation (video)**
  - +30000 gates

Improvement in MIPS over general-purpose 32b RISC
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,
TIE Summary

Application-specific instructions
Conclusion

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

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