Configurable and Extensible Processors Change System Design

Ricardo E. Gonzalez
Tensilica, Inc.
Yet Another Processor?
- No, a new way of building systems
- Puts system designers in the drivers seat

Configurable and Extensible
- Select and size only what you need
- Add system-specific instructions for 4-50× improvement

Portable
- Build your system ASICs in any foundry

Complete
- Hardware and software configured/extended together

Simple, Fast, and Robust
- Configure/extend in hours instead of months
Get the YAP stuff out of the way

<table>
<thead>
<tr>
<th>Instruction Set Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Instructions</td>
</tr>
<tr>
<td>Code size</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Registers</td>
</tr>
<tr>
<td>Address bits</td>
</tr>
<tr>
<td>Data bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Xtensa V1.5 Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline</td>
</tr>
<tr>
<td>Inst Cache</td>
</tr>
<tr>
<td>Data Cache</td>
</tr>
<tr>
<td>Write buffer</td>
</tr>
<tr>
<td>MMU</td>
</tr>
</tbody>
</table>
Xtensa’s unique 24/16 encoding

<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></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E.g. $AR[r] \leftarrow AR[s] + AR[t]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>imm8</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>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E.g. if $AR[s] &lt; AR[t]$ goto PC+imm8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>imm12</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></td>
<td></td>
<td></td>
<td>E.g. if $AR[s] = 0$ goto PC+imm12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>imm16</th>
<th>t</th>
<th>op0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>E.g. $AR[t] \leftarrow AR[t] + imm16$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>imm18</th>
<th>n</th>
<th>op0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>E.g. CALL0 PC+imm18</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></td>
<td></td>
<td></td>
<td>E.g. $AR[r] \leftarrow AR[s] + AR[t]$</td>
</tr>
</tbody>
</table>
Huffman encoding (EPIC image compression)
for (i=0; i< NUM; i++)
    if (histogram[i] != NULL)
        insert (histogram[i], &tree);

Xtensa’s new ISA takes performance and code size to a new level

Xtensa code
L16: addx4 a2, a3, a5
132i a10, a2, 0
beqz a10, L15
add a11, a4, a7
call8 insert
L15: addi a3, a3, 1
bge a6, a3,L16

ARM code
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,#3e8
BLT J4

Thumb code
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

7 instructions 17 bytes  8 instructions 36 bytes  10 instructions 20 bytes
Is it real?

YES!

V1.5 released June
Customer designs underway

First silicon fully functional using
TSMC 0.25u CMOS process
  – > 200 MHz typical
  – 140 MHz worst case (T,V,P)

<table>
<thead>
<tr>
<th></th>
<th>Core Area</th>
<th>Gates</th>
<th>Core Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area Optimized</td>
<td>1.1mm(^2)</td>
<td>27K</td>
<td>0.8 mW/MHz</td>
</tr>
<tr>
<td>Speed Optimized</td>
<td>1.5mm(^2)</td>
<td>35K</td>
<td>1.0 mW/MHz</td>
</tr>
</tbody>
</table>
But that’s not as neat as...

Describe the processor attributes from a browser

Using the **Xtensa** processor generator, create...

Tailored, HDL μP core

Customized Compiler, Assembler, Linker, Debugger, Simulator, Eval Board

Use a standard cell library to target to the silicon process
Xtensa is designed for configurability and extensibility.
SW development tools are ‘in-sync’

Built on top of industry standard GNU tools

All tools automatically tuned and extended for application-specific instructions and configuration

✔ ANSI C/C++ Compiler
✔ Assembler
✔ Debugger
✔ Linker
✔ Code Profiler
✔ Instruction Set Simulator
✔ Function Libraries
Extensibility via TIE Language

TIE (Tensilica Instruction Extension)
- Basis for Designer-Defined Instructions
- Describes instruction encoding and semantics
  - Semantics in Verilog subset
- Is independent of pipeline
  - Easy to write
  - Same description will work with future Tensilica designs
- Translated to Verilog, VHDL, and C
  - RTL, Compiler, Assembler, Simulator, Performance model, and Debugger support is automatic
  - Decode, interlock, bypass, and immediate logic generated from encoding descriptions
// define a new opcode for byteswap
opcode BYTEWSAP op2=4'b0000 CUST0

// define a new instruction class
iclass bs {BYTESWAP} {out arr, in ars}

// semantic definition of byteswap
semantic bs {BYTESWAP} {
    assign arr =
        {ars[7:0],ars[15:8],ars[23:16],ars[31:24]};
}
Slightly More Complicated TIE

opcode BYTESWAP op2=4'b0000 CUST0

// declare state SWAP and ACCUM
state SWAP 1
state ACCUM 40

// map ACCUM and SWAP to user register file entries
user_register 0 ACCUM[31:0]
user_register 1 {SWAP, ACCUM[39:32]}

iclass bs {BYTESWAP} {out arr, in ars} {in SWAP, inout ACCUM}

semantic bs {BYTESWAP} {
  wire [31:0] ars_swapped =
    {ars[7:0], ars[15:8], ars[23:16], ars[31:24]};
  assign arr = SWAP ? ars_swapped : ars;
  assign COUNT = COUNT + SWAP;
  assign ACCUM = {ACCUM[39:30] + ars_swapped[31:24],
  ACCUM[29:20] + ars_swapped[23:16],
  ACCUM[19:10] + ars_swapped[15:8],
  ACCUM[ 9: 0] + ars_swapped[7:0]};
}
TIE Hardware Generation

- Instruction
  - Instruction Decode
  - Pipe Control
  - Register File
  - Bypass

R stage

- State
- Decode
- Data
  - TIE
  - Branch
  - Shifter
  - Adder
  - AGen

E stage

- result
- vAddr
TIE Software Support

- Translate from TIE to C for native development
- Add instructions to assembler, debugger, simulator, performance model
- Add intrinsics to the compiler
  - developer can code in C/C++
    e.g. write
    
    ```
    out[I] = byteswap(in[I]);
    ```
Typical TIE Development Cycle

1. Develop Application in C/C++
2. Profile and Analyze
3. ID Potential New Instructions
4. Implement in TIE
5. Translate TIE to C/C++
6. Profile and Analyze
7. Re-compile Source with new instructions instead of function calls
8. Run Cycle Accurate ISS
9. Synthesize and Build Processor

Acceptable?

Acceptable?
Triple DES Example

Triple DES used for

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

Add 4 TIE instructions:

- Speed increased by 43X to 72X
- Code and data storage requirement reduced by 36X
- ~4500 additional gates
- No cycle time impact

DES Performance

<table>
<thead>
<tr>
<th>Block Size (Bytes)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>43</td>
</tr>
<tr>
<td>64</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>72</td>
</tr>
<tr>
<td>Mean</td>
<td>53</td>
</tr>
</tbody>
</table>
Application-specific processors make a huge difference in performance.

- **FIR filter** (signal processing) + 6500 gates
- **JPEG** (image compression) + 7500 gates
- **Viterbi Decoding** (wireless communication) + 900 gates
- **Motion Estimation** (video conferencing) + 1000 gates
- **DES** (content encryption) + 4500 gates

Improvement in MIPS, MIPS/Watt over general-purpose 32b RISC.
# Extension via TIE vs. special logic outside the CPU

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>No latency introduced by communication between processor and special hardware</td>
<td>Some Verilog constructs not supported</td>
</tr>
<tr>
<td>Easier to accomplish complicated control using the instruction stream</td>
<td></td>
</tr>
<tr>
<td>Field upgrade or fix bugs by changing code not hardware</td>
<td></td>
</tr>
<tr>
<td>Easier prototyping, verification and debug</td>
<td></td>
</tr>
</tbody>
</table>
Extension via TIE vs. RTL Modification

TIE is a high-level specification:
- Software generated to match hardware
- Easier to write
  - don’t need to understand pipeline
  - mixed native/cross development methodology
- Easier to verify instruction definition
  - verify in C, not by simulating RTL
  - correct by construction pipeline etc. logic
- Faster to market
- Faster design iteration, better feedback produces better final architecture
The tail shouldn’t wag the dog

Custom processors tie the system to a foundry

But the processor may be a small part of the total design – why is it forcing the foundry choice?

Xtensa uses virtually any standard cell library with commercial memory generators

to scale on a typical $10 IC (3-6% of 60mm²)
Synthesized vs. custom (1)

**Custom Pros:**
- Potentially highest MHz
  - dynamic circuits
  - controlled layout
- Good micro-architecture support
  - CAM structures for TLBs, address buffers, etc.
  - specialized RAMs (e.g. pipelined)

**Custom Cons:**
- Compromised by Porting
  - Designs rarely see volume in target process
  - Old designs fail to exploit new processes
- Hard to integrate
  - CAD tool differences
  - foreign layout
- Design+Debug longer and costlier
- Hard to modify
  - cannot configure
  - cannot extend

• Design+Debug longer
  – dynamic circuits
  – controlled layout
• Good micro-architecture support
  – CAM structures for TLBs, address buffers, etc.
  – specialized RAMs (e.g. pipelined)
Synthesized vs. custom (2)

**Synthesized Cons:**
- Standard cell libraries
  - limited functionality
  - not designed for high-speed
- Portability implications
  - avoid tri-state
  - avoid 2-phase transparent latch clocking
- Large clocking overhead

**Synthesized Pros:**
- Portable
  - Moves to new foundries easily
  - Takes advantage of new process technology as it becomes available
- Allows configurability
  - Need only change netlist; layout is automatic
  - Extensibility delivers more than a few extra MHz
- Map RTL multiple ways
  - use low-power library
  - change synthesis goals
Conclusions

Configurable/extensible processors are here
  – Key is integrated hardware/software solution
  – Boon for system designers

Synthesizable processors are effective solutions
  – Competitive with traditional offerings in MHz, mm², mW
  – Portable and easier to work with

Advantage of extensibility overwhelms the limited (and often theoretical) advantages of custom design