Configurable Cores: XTensa

CSE 291E / EE260C
Spring 2002
Overview

• Overview of Tensilica
• Overview of XTensa
  – Design Flow
  – Base ISA
  – Optional Components
  – TIE
• Challenges for Customized Processors
  – What are the problems?
  – How does XTensa address them?
  – What could be done better?
Tensilica: The Company

- Founder/CEO/Mastermind - Chris Rowen
  - from Intel, Stanford, MIPS, sgi, Synopsys
- Idea: Build customizable processor design chain
- Strategic Investors
  - Cisco, Matsushita, Altera, Conexent, NEC
Motivation: The “Design Gap”

Source: NTRS’97
Motivation: Before
Motivation: With XTensa
Design Flow - high level

1. **Choose Processor Options**
2. **Build Configuration**
3. **Download Software Tools**
4. **Download Hardware Tree**
5. **Profile Application Code**
6. **Synthesize and Check Timing**
   - **Analyze Performance**
7. **Integrate with System Software**
8. **Integrate with Rest of ASIC**
Design Flow – detailed

1. Processor Configuration Inputs
2. Designer-Defined Instruction Descriptions
3. Configuration File
4. Configured GNU C/C++ Compiler
5. Configured GNU Assembler/Disassembler
6. Configured Instruction Set Simulator/Emulator
7. Configured Processor HDL
8. Area, Power and Timing Estimation
9. Logic Synthesis (Synopsys or Ambit)
10. Block Place/Route (Avant! or Cadence)
11. Timing Verification
12. Hardware Profile
13. OPTIMIZED HARDWARE
14. Software Debugging/Profiling
15. Application Simulation with ISS and/or Emulator
16. Application Specific Compile, Assemble, Link
17. Sample Application Data
18. Application Source Code
19. OPTIMIZED SOFTWARE
Xtensa ISA Priorities

- Code Size
  - Large factor in system cost
- Configurability, Extensability
  - Provide match to requirements
- Processor Cost
  - More than just area?
- Energy Efficiency
- Performance
- Scalability
- Features
Xtensa ISA

- RISC architecture
- 5-stage Pipeline
  - I R E M W
- 24/16 bit instructions

  \[0.25\mu\]
  - 56 to 141MHz
  - 30 to 119K gates
  - 54 to 237mW power
  - 1.7mm² to 42.4mm² including cache RAMs

  \[0.18\mu\]
  - 93 to 200MHz
  - 30 to 91K gates
  - 36 to 129mW power
  - 0.9mm² to 17.3mm² including cache RAMs
Xtensa Pipeline

I

Instruction RAM
Instruction Cache
Instruction ROM

R

Decode
General Registers
CoProcessor Registers

E

Address Generation
ALU
CoProcessor ALU

M

XLMI
Data ROM
Data Cache
Data RAM

W

Instruction ROM
XTensa Architecture

- Align and Decode
  - Register File
  - ALU
  - Mul16
  - Mac16
  - Mul32
  - FPU
  - Vectra DSP

- Instruction Fetch Unit
  - Instr TLB
  - Instruction Cache
  - Instruction ROM
  - Instruction RAM

- Data Load/Store Unit
  - Data TLB
  - Data Cache
  - Data ROM
  - Data RAM
  - XLMI

- Processor Controls
  - Interrupt Control
  - Timers 0-n
  - Exception Control

- Optional & Configurable
  - Data Address Watch 0-n
  - Instr Address Watch 0-n

- Base ISA
- Optional
- Configurable
- Advanced
TIE

• Major parts of TIE
  – Instruction Fields
  – Opcodes
  – Operands
  – State and Register
  – Instruction Semantics
  – Compiler Prototype
  – Pipelining/Scheduling

• What do we need to worry about?
TIE Overview

• No micro-architecture details
  – Same TIE will work with new base
  – Decode, interlock, bypass, and pipelining, OS support of context switch automatic

• Automatic configuration of software tools
  – Compiler
  – Instruction-set simulator
  – Debugger

• Automatic Synthesis
TIE Example: ADD4

```plaintext
regfile datareg 64 16
c-type 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 Example: Accum

```plaintext
// define a new opcode for byteswap
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]}

// define a new instruction class
iclass bs {BYTESWAP} (out arr, in ars) { in SWAP,
inout ACCUM}

// semantic definition of byteswap
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 ACCUM = {ACCUM[39:30] + arr[31:24],
                    ACCUM[29:20] + arr[23:16],
                    ACCUM[19:10] + arr[15:8],
                    ACCUM[ 9: 0] + arr[7:0]};
}
```
Code Size

• Small Encoding Size
  – 24/16 bits
  – Mode-less encoding

• Code savings from elimination of save/restore
  – Special instruction now handles this
  – Estimate 6-10% Reduction in code size

• Compound Instructions
  – Loop instructions / Compare-and-Branch
  – Shift-Add/Subtract
  – Shift-Mask
## Instruction 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>E.g. $AR[r] \leftarrow AR[s] + AR[t]$</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>E.g. $AR[t] \leftarrow AR[t] + \text{imm16}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm18</td>
<td>n</td>
<td>op0</td>
<td>E.g. CALL0 PC+imm18</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>op0</td>
<td>E.g. $AR[r] \leftarrow AR[s] + AR[t]$</td>
</tr>
</tbody>
</table>
Example Code

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

**Xtensa code**

L16:  addx4 a2, a3, a5  
L132i 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  
BBQ 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
What are the challenges?

• Layout / Synthesis
• Code Size
• Verification
• Ease/Speed of use
Layout / Synthesis

• Not Full Custom
• One Clock
  – Rising Edge Triggered
  – No standard processor tricks
    • No time borrowing
• Caches
  – Generated with memory compiler
  – Registered address input
• Use hints to layout tools to make sure there is a sane placement
Code Size

- Small instruction encoding
- Compound instructions
- Register Windows
- Section/Pooling Literals
- What else could we do?
Verification

• Directed Diagnostics
• Pseudo-Random program generator
• Coverage Analysis
  – Architecture level (AVP)
  – Micro-architecture level (MVP)
  – Random generator
  – Cycle accurate simulator -> Co-simulation
• What else?
Ease/Speed of use
Conclusions and Discussion

• What is next for Tensilica and Customized processors in the future?
  – CMP?
  – Vector?
  – FPGA based?
  – Tile based?
  – What about further out?