### Andes Corporate Overview

**Silicon Valley Tie**
- Core R&D from AMD, DEC, Intel, MIPS, nVidia, and Sun

**15-Year CPU IP Company**
- IPO in 2017; HQ in Taiwan
- AndeStar™ V1-V3, V5 (RISC-V)

**>1 Bn Annual Run Rate of Andes-Embedded SoC**
- ~300 customers in TW, CN, US, EU, JP, KR

**Founding Platinum Member and Major Contributor**
- Chairing Task Groups
- Contributing to GNU, LLVM, uBoot, glibc, Linux, etc.
Andes V5 Product Overview

**AndesCore™**
- High performance processors with leading PPA

**AndeStar™**
- Architecture V5

**AndeSight™**
- Professional IDE with high code quality

**AndeShape™**
- Handy peripheral IPs to speed up SoC construction

**AndeSoft™**
- Extensive SW stacks from bare metal, RTOS to Linux

---

2020 RISC-V CON Webinar
Agenda

- Computing Accelerations
- Andes Custom Extension™
- ACE Examples
- Summary
Standard IP Products?

Performance?  
Power?  
Area?  Cost?

Standard IP products cannot satisfy all requirements

⇒ Application Specific Accelerations ⇐
Evolution of Computing Acceleration

- **HW Acceleration Engine**
  - Crypto engines

- **Co-processors**
  - Graphic Processing Unit (GPU)

- **ISA expansion**
  - RISC-V M-, P-, V- extensions

- **User extensions**
  - RISC-V open ISA allows custom instruction
## Comparison of Acceleration Methods

<table>
<thead>
<tr>
<th></th>
<th>HW Engine</th>
<th>Co-Processor</th>
<th>ISA Expansion</th>
<th>User Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start-up latency</strong></td>
<td>Longest</td>
<td>Long</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td><strong>Resource sharing</strong></td>
<td>None</td>
<td>Decode/Control</td>
<td>Control &amp; RF</td>
<td>Control &amp; RF</td>
</tr>
<tr>
<td><strong>Implementation Freedom</strong></td>
<td>A TON</td>
<td>Lots</td>
<td>None</td>
<td>Restricted</td>
</tr>
<tr>
<td><strong>Proprietary advantage</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Best for</strong></td>
<td>Very heavy Semantics</td>
<td>Medium Semantics</td>
<td>Commoditized Computation</td>
<td>Low-Medium Semantics</td>
</tr>
</tbody>
</table>
Accelerating Data-Intensive Computation

- **Two parts:**
  - Data IO
  - Compute acceleration

- **Data IO: DMA with double buffers**
  - Switch buffers on phase changes

- **Compute Acceleration: ACE**
  1. ACE to perform Computation. E.g.
     - Dot products of 2 64x8 vectors
     - Matrix convolution
  2. ACE to control existing HW Engine. E.g.
     - Sending 90-bit commands in every cycle
Andes Custom Extension™

- scalar/vector
- wide operands
- direct IO

COPilot

Custom-OPtimized Instruction deveLOpment Tools

Extended Tools

Extended ISS

CPU ISS (near-cycle accurate)

Extended RTL

Compiler Asm/Disasm Debugger IDE

CPU RTL

Automated Env. For Cross Checking

Test Case Generator

Extended ISS

Extended RTL

Extensible Baseline Components

A new CPU and its tools

Taking RISC-V® Mainstream
COPilot w/ AndeSight IDE

Highly Integrated with AndeSight

Near Cycle Accurate Simulator Supported

Profiling Tool Provides Easy Spotted Result

Monitoring Customized Instructions

Hassle-Free Intrinsic Functions/API Replacement

Taking RISC-V® Mainstream
# ACE Features

<table>
<thead>
<tr>
<th>Items</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction</strong></td>
<td></td>
</tr>
<tr>
<td>scalar</td>
<td>single-cycle, or multi-cycle</td>
</tr>
<tr>
<td>vector</td>
<td><strong>for</strong> loop, or <strong>do-while</strong> loop</td>
</tr>
<tr>
<td>background option</td>
<td>retire immediately, and continue execution in the background. Applicable to scalar and vector.</td>
</tr>
<tr>
<td><strong>Operand</strong></td>
<td></td>
</tr>
<tr>
<td>standard</td>
<td>immediate, GPR, baseline memory (thru CPU)</td>
</tr>
<tr>
<td>custom</td>
<td>- <strong>ACR</strong> (ACE Register), <strong>ACM</strong> (ACE Memory), <strong>ACP</strong> (ACE Port)</td>
</tr>
<tr>
<td></td>
<td>- With arbitrary width and number</td>
</tr>
<tr>
<td>implied option</td>
<td>Implied operands don’t appear in mnemonic</td>
</tr>
<tr>
<td><strong>Auto Generation</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Opcode assignment: automatic by default</td>
</tr>
<tr>
<td></td>
<td>- All required tools, and simulator (C or SystemC)</td>
</tr>
<tr>
<td></td>
<td>- <strong>RTL code for instruction decoding, operand mapping, dependence checking, input accesses, output updates</strong></td>
</tr>
<tr>
<td></td>
<td>- Logic sharing</td>
</tr>
<tr>
<td></td>
<td>- Waveform control file</td>
</tr>
</tbody>
</table>
ACE Development Flow

Start

Profile application SW to identify time-critical code

Define ACE instructions with cycle estimation

Cycles met?

Yes

No

Implement ACE RTL

Evaluate PPA

Requirements met?

Yes

No

Room to Improve?

Yes

No

Done
ACE for Performance Efficiency

- **High speedup:**
  - Obvious results of defining instructions to match computation

- **Energy reduction:**
  - No ACE: N instructions going thru fetch, decode, execute, retire
    - $N \times (\text{fetch} + \text{decode} + \text{retire}) + N\times\text{execute}$
  - With ACE: only 1 instruction fetch, decode, retire
    - $1 \times (\text{fetch} + \text{decode} + \text{retire}) + \text{more efficient logic for } N\times\text{execute}$

<table>
<thead>
<tr>
<th>Andes’ Core + ACE Instructions</th>
<th>FIR Filter</th>
<th>CRC32</th>
<th>3DES</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speedup</strong></td>
<td>20x</td>
<td>&gt;140x</td>
<td>&gt;300x</td>
</tr>
<tr>
<td><strong>Power efficiency</strong></td>
<td>30x</td>
<td>&gt;240x</td>
<td>&gt;300x</td>
</tr>
</tbody>
</table>

* Estimated gains with and without corresponding ACE instructions
Madd32: A half-page ACE design

**madd32.ace**

```plaintext
code madd32 { 
  operand= {io gpr acc, 
              in gpr data, in gpr coef};
  csim= %{
    acc+= (data & 0xffff) * (coef & 0xffff)
      + (data >>16) * (coef >>16);
  };
  latency= 1;
};
```

**madd32.v**

```plaintext
//ACE_BEGIN: madd32
  assign acc_out = acc_in
    + data[15:0] * coef[15:0]
    + data[31:16] * coef[31:16];
//ACE_END
```

► **madd32.ace**: ACE definition file
- **insn**: define a scalar instruction, “madd32”
- **operand**: operand names/attributes (in/out/io gpr, imm, ...)
- **csim**: instruction semantics in C for ISS
- **latency**: estimated cycles spent on instruction execution; default=1
- ➔ auto-generated intrinsic: `acc_madd32()`

► **madd32.v**: concise Verilog

```plaintext
//ACE_BEGIN: madd32
  assign acc_out = acc_in
    + data[15:0] * coef[15:0]
    + data[31:16] * coef[31:16];
//ACE_END
```

... //custom logic
Madd32: Application C Code

```c
uint fir32(uint *C, uint *X, uint n) {
    uint rslt = 0;
    for (int i = 0; i < n; ++i)
        rslt += (C[i] & 0xffff) * (X[i] & 0xffff) + (C[i] >> 16) * (X[i] >> 16); // lower 16 bits
    return rslt;
}
```

```c
#include "ace_user.h" // prototypes for generated intrinsic
...
#endif USE_ACE
    rslt = ace_madd32(rslt, X[i], C[i]); // invoke intrinsic
#else ...
#endif
```

```c
op = { io acc, in dat, in coef }; // Auto-Generated Intrinsic Funct
```

Taking RISC-V® Mainstream
vmadd32: Vectorizing madd32

**Vector Instruction**

```c
vec insn vmadd32 {
  operand= {io ACC acc,
            in XMEM dat, in YMEM coef,
            imm5 cnt};
  loop_type= repeat(cnt);
  stride<dat>= 1;
  csim= %{
    ...
    %};
  latency= 1;
};
```

- for loop, repeat "cnt" times
- Memory address automatically increases 1 for next iteration
- per-iteration operation and latency

```c
ram XMEN { //same for YMEM
  interface = SRAM;
  width = 32;
  address_bits = 12;
};
```

```c
reg ACC {
  number = 4;
  width = 32;
};
```

- csim & RTL: same as scalar version !
- per-iteration logic

// ACE_BEGIN: vmadd32
...
// ACE_END
Background Vector Instruction

```c
vec bg_insn vmadd32 {
  operand= {io ACC acc, in XMEN dat, in YMEN coef, imm5 cnt};
  loop_type= repeat(cnt);
  stride<dat>= 1;
  csim= %{ ... %};
  latency= 1;
};
```

```c
ram XMEN { //same for YMEN
  interface= SRAM;
  width= 32;
  address_bits= 12;
};
```

```c
reg ACC {
  number = 4;
  width= 32;
};
```

Everything else:
same as foreground version !!!

Background instruction is executed in parallel with CPU baseline instructions
ACP for Direct HW Engine Control

**CPU controls 4 HW engines.**

---

**App. code sequence:**
- prepare command (say, thru ACR);
- send command;
- do other useful work;
- wait for results to be ready;
- get results;

---

**ace script (Andes Custom Port)**

```plaintext
port command { //a 90-bit output port to all 4 HW engines
    width= 90;
    io_type= out;
}
//4 HW engines
port ready { //4 ready signals
    num=4;
    io_type= in;
}
port results { //4 256-bit input ports
    num= 4;
    width= 256;
    io_type= in;
}
```
Logic Sharing

- Same instance name: same HW
- Different instance names: different HW
- All control code is auto-generated to enable the sharing

```verilog
module sxtxn_by_sxtxn (  
  output [31:0] rslt,  
  input [15:0]  x,  
  input [15:0]  y  
);  
assign rslt= x * y;  
endmodule
```
Uniqueness in Verification

- Incorporate auto-verification in the flow

Some: auto-generated Verilog & Csim - same source → can’t cross-check

- Use standard languages you are already familiar with
  - Plus ACE script to guide the tool to do housekeeping work for you

Andes: common verification approach - C and Verilog → auto cross-checking
RTL Debugging

Auto-generate waveform control file

Operands and ACE interface signals will be added into waveform control file automatically.
Virtual Prototyping Support

SystemC CPU model library (baseline CPU, GDB server, local memory, interface, ...)

Core Module

- Reset
- Clock
- ILM
- DLM
- AndesCore
- ACE
- ACR
- BIU
- Interrupt
- ACM
- Arbiter
- Simple Bus
- Memory
- Device

ACE resource access interface

TLM 2.0 Environment

Integrated with other SoC components

Taking RISC-V® Mainstream
Inner Product of Vectors with 64 8-bit Data

reg CfReg {
    num = 4;
    width = 512;
}

ram VMEM {
    //vector Custom Memory
    interface = sram;
    address_bits = 3;
    //8 elements
    width = 512;
}

insn ip64B {
    operand = {out gpr IP, in CfReg C, in VMEM V};
    csim = 
    IP = 0;
    for (uint i = 0; i < 64; ++i)
        IP += ((C >> (i*8)) & 0xff) * ((V >> (i*8)) & 0xff);
    %};
    latency = 3;  //enable multi-cycle ctrl
};

//ACE_BEGIN: ip64B
assign IP = C[7:0] * V[7:0]
    + C[15:8] * V[15:8]
    ...
    + C[511:504] * V[511:504];
//ACE_END

Speedup: 85x
Madd32 w/ Ring Buffers on XY Memory

- 2-entry address registers xadr point to 2 ring buffers in XM without keeping reloading their initial values.
- Similar mechanism in Verilog side

```vhdl
madd32rb.ace

insn madd32rb { //with ring buffer
          op = {io gpr acc,
                in XM@xadr:u data, in YM@yadr:u coef};

          csim= %{
                acc+=(data & 0xffff) * (coef & 0xffff)
                 +(data >> 16) * (coef >> 16);

                //update XM/YM pointers in any desired form
                data_addr_nx = (data_addr + 1) & 0x7f;
                coef_addr_nx = (coef_addr + 1) & 0x7f;
          %};
}

ram XM {
          address_bits= 12;  //4K elements
          width= 32;        //of 32 bits
          interface= SRAM;
};

reg xadr { //addr registers for XM
          number= 2;
          width= 12;
};

ram YM { ... }; //same as XM
reg yadr { ... }; //same as xadr
```

- Similar mechanism in Verilog side
Summary of ACE Advantages

- Designers focus on instruction semantics, not CPU pipeline
- Housekeeping tasks are offloaded to COPILOT
  - Opcode selection and instruction decoding
  - Operand mapping/accesses Updates
  - Dependence checking
- Comprehensive support
  - Powerful instruction semantics: vector, background, wide operands
  - Custom architecture data types: ACR, ACM, ACP
  - Auto-generation of RTL code, verification env., development tools
- ACE unlocks RISC-V’s potential for Domain Specific Accelerator
Thank you!
See you next Webinar