Domain-Specific Acceleration via AndeStar™ V5 Processors

Charlie Su, Ph.D.
CTO and SVP
Talk Highlights

- Overview of Andes
- Update of Andes RISC-V Processors
- Andes Custom Extension™ for DSA
Andes Technology Corporation

- A 13-year-old public CPU IP company
- A founding member of the Foundation
- A major open source contributor
- Actively involved in standard extensions
  - Chair of P-extension (Packed DSP/SIMD) Task Group
  - Co-chair of Fast Interrupt Task Group
Baseline Processors for DSA

- Extensibility in RISC-V enables DSA
- Acceleration is the key, but 80-20 still applies
  - 80% of the time spent on 20% of the code/logic
  - For rest of the logic, its power & area matter
  - For rest of the code, its size matters. So is performance

- Baseline processors are important too

- AndeStar™ V5 architecture =
  RISC-V standard
  + Andes baseline extensions
  + Andes Custom Extension™ (ACE)
AndesCore™ 25-series at 28nm

► Smallest usable N25/NX25¹: ILM/DLM, no caches/BTB

- N25 @ 1 GHz: 37K, 0.033 mm², 4.1 uW/MHz
- NX25 @ 1 GHz: 56K, 0.044 mm², 6.0 uW/MHz

<table>
<thead>
<tr>
<th>Features</th>
<th>N*25</th>
<th>N*25F</th>
<th>A*25</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/D Local Memory</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>32KB I$/D$ + 256 BTB</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SP/DP FPU</td>
<td>NA</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MMU and S-Mode</td>
<td>NA</td>
<td>NA</td>
<td>Yes</td>
</tr>
<tr>
<td>Worst-Case Max. Freq. (GHz)¹</td>
<td>1.4</td>
<td>1.3</td>
<td>1.2</td>
</tr>
<tr>
<td>Coremark/MHz²</td>
<td>3.58 (rv32), 3.52 (rv64)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMIPS/MHz (ground rule)²</td>
<td>1.96 (rv32), 2.09 (rv64)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹: TSMC 28HPC+ RVT 9T library and high-speed memory. Frequency condition: 0.81v/-40°C.
²: BSP V5.0.0 toolchain; DMIPS/ground rule uses no-inline option.
AndesCore 22-Series

- AndeStar V5 or V5e ISA
  - Based on RV32-IMC or RV32-EMC

- 2-stage pipeline with AHB-lite main bus

- Rich baseline options:
  - I/D Local Memory, I Cache
  - Fast or small multiplier, branch predictions
  - Up to 16-entry PMP
  - M-mode, or M+U-mode
  - APB private peripheral port, Fast IO port
  - WFI, WFE, and PowerBrake
  - Vectored and preemptive PLIC

- Advanced options: ACE, DSP, FPU

- 28nm PPA: >750 MHz (worst case), 15K gates (minimal)

- Per-MHz performance: 1.59 DMIPS (no inline), 3.69 Coremark
ACE Framework

- C code
- Verilog
- Attributes
- scalar/vector
- background
- wide operands

COPILOT
Custom-OPtimized Instruction developmEnt Tools

Automated Env. For Cross Checking
Test Case Generator

Extended ISS

Extended Tools

Extended ISS

CPU ISS (near-cycle accurate)

Extended RTL

CPU RTL

Compiler Asm/Disasm

Debugger IDE

Extensible Baseline Components

Executable or library

Source file
madd32: A Half-Page ACE Design

**madd32.ace**

```ace
insn madd32 {  
operand= {io gpr acc,  
in gpr data, in gpr coef};  

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

latency= 1;  
};
```

**madd32.ace: ACE definition file**
- **insn**: define a scalar instruction, “madd32”
- **op(erand)**: operand names and attributes (in/out/io gpr, imm, etc.)
- **csim**: instruction semantics in C for ISS
- **latency**: estimated cycles spent on instruction execution; default=1

The auto-generated intrinsic: `acc_madd32()`

**madd32.v**

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

**madd32.v: concise Verilog**

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

```
//ACE_BEGIN: madd32  
assign acc_out = acc_in  
  + data[15:0] * coef[15:0]  
  + data[31:16] * coef[31:16];  
//ACE_END
```
vmadd64: vectorizing madd32

```c
vec insn vmadd64 {
    operand= {io ACC acc, in XM data, in YM coef, in gpr cnt};
    loop_type= repeat(cnt);
    stride<data>= 1;
    csim= %{
        ...
    };
    latency= 1;
    latency_overhead= 2;
};

reg ACC {
    number= 4;
    width= 64;
};

ram XM {
    //same for YM
    width= 32;
    address_bits= 12;
    interface= SRAM;
};
```

- vec insn: define a vector instruction
- loop_type= repeat(n): . execute vector body n times
- csim: per-element operations
- latency: per element latency
- latency_overhead: extra one-time latency for vector instructions

RTL & csim: same as scalar version !!!
bvmadd64: backtorizing madd32

```c
vec bgInsn bvmadd64 {
  operand= {io ACC acc,
             in XM data, in YM coef,
             in gpr cnt};
  loop_type= repeat(cnt);
  stride<data>= 1;
  csim= %{
    ...
  };
  latency= 1;
  latency_overhead= 2;
};
```

```c
reg ACC {
  number= 4;
  width= 64;
};
```

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

//ACE_BEGIN: bvmadd64
  . . .
//ACE_END

- `vec bg_insn`: a background vector
- Options are available to control the parallelism of background execution
- Sync instructions are automatically generated for each background instruction. There is also an ACE global sync instruction.

Everything can be same as foreground version !!!
Driving Innovations™

**madd32 with Ring Buffers on XY Memory**

- **madd32rb.ace**

  ```plaintext
  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
      data addr_nx= (data addr + 1) & 0x7f;
      coef addr_nx= (coef addr + 1) & 0x7f; }
  }
  ```

- **madd32rb.ace**

  ```plaintext
  ram XM {
    address bits= 12; // 4K elements
    width= 32; // of 32 bits
    interface= SRAM;
  }
  reg xadr { // addr registers for XM
    number= 4;
    width= 12;
  }
  ram YM { ... }; // same as XM
  reg yadr { ... }; // same as xadr
  ```

- 4-element address registers **xadr** can point to 4 ring buffers in **XM** without reloading their initial values.
Logic Sharing

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

```
module sxttn_by_sxttn (  
    output [31:0] rslt,  
    input [15:0]  x,  
    input [15:0]  y  
  );  

  assign rslt= x * y;  
endmodule
```

```
//ACE_BEGIN: madd32  
...  
(*ace_shared*)  
sxttn_by_sxttn mult1( ... );  
(*ace_shared*)  
sxttn_by_sxttn mult2( ... );  
...  
//ACE_END
```

```
//ACE_BEGIN: msub32  
...  
(*ace_shared*)  
sxttn_by_sxttn mult1( ... );  
(*ace_shared*)  
sxttn_by_sxttn mult2( ... );  
...  
//ACE_END
```
Inner Product of Vectors with 64 8-bit Data

```c
reg CfReg {
    num= 4;
    width= 512;
};
rpm VMEM {
    interface= sram;
    address_bits= 3;
    width= 512;
};
insn ip64b {
    operand={
        out gpr IP,
        in CfReg C, in VMEM V};
    csim= {%
        //multi-precision lib. used
        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

Intrinsic: long ace_ip64b(CfReg_t, VMEM_t);
Operands and ACE interface signals will be added into waveform control file automatically.
Benefits of ACE

- Users focus instruction semantics, not CPU pipeline

- Housekeeping tasks are offloaded to COPILOT
  - Opcode selection and instruction decoding
  - Operand mapping/accesses/updates
  - Dependence checking

  ➔ Adding instructions is similar to ASIC design

- Comprehensive support:
  - Powerful instruction semantics: vector, background, wide operands
  - Auto-generation of verification environment, development tools and RTL code

- SW invokes the instructions by using intrinsic functions

- COPILOT unlocks RISC-V’s potential for DSA
Thank You!

Andes: Trusted Computing Expert and Your Best RISC-V Partner!