RISC-V CON

ONLINE WEBINAR

Andes Technology
Andes Custom Extension™

Accelerating Domain Specific Architecture

Marketing Division

YiChiang Chang

2020.04.09

Andes Corporate Overview

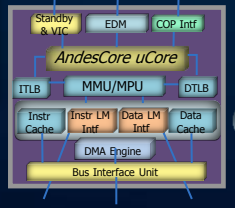
Silicon Valley Tie	• Core R&D from AMD, DEC, Intel, MIPS, nVidia, and Sun
15-Year CPU IP Company	• IPO in 2017; HQ in Taiwan • AndeStar™ V1-V3, V5 (RISC-V)
>1 Bn Annual Run Rate of Andes-Embedded SoC	• ~300 customers in TW, CN, US, EU, JP, KR
Founding Platinum Member and Major Contributor	• Chairing Task Groups • Contributing to GNU, LLVM, uBoot, glibc, Linux, etc.

Andes V5 Product Overview

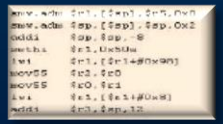
Best extensions to RISC-V

AndesCore™ Processors

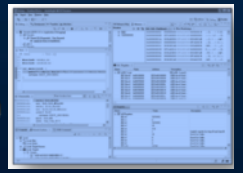
Highly optimized design with leading PPA



AndeStar™ Architecture V5



AndeSight™ Tools



Professional IDE with high code quality

Andes Embedded™

Handy peripheral IPs to speed up SoC construction

AndeShape™ Platforms

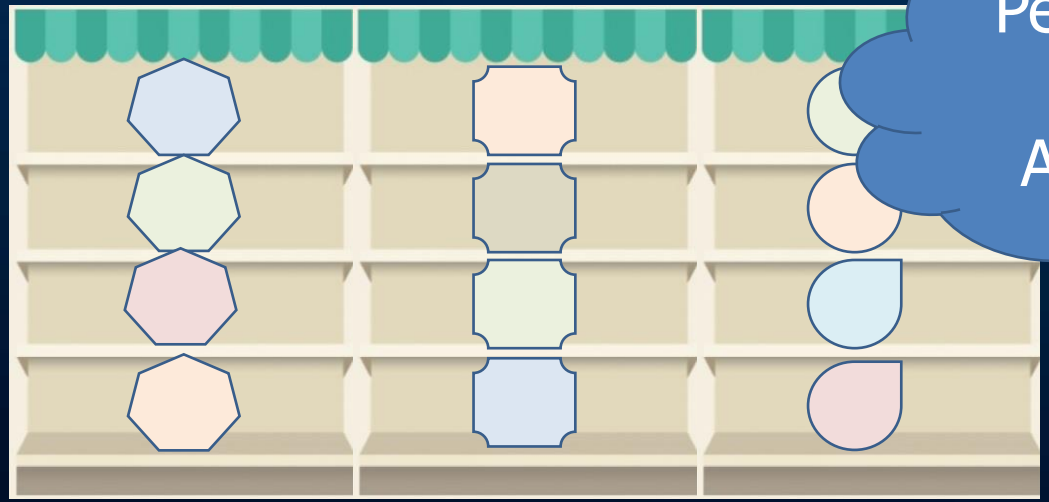
AndeSoft™ Stacks

Extensive SW stacks from bare metal, RTOS to Linux

Agenda

- Computing Accelerations
- Andes Custom Extension™
- ACE Examples
- Summary

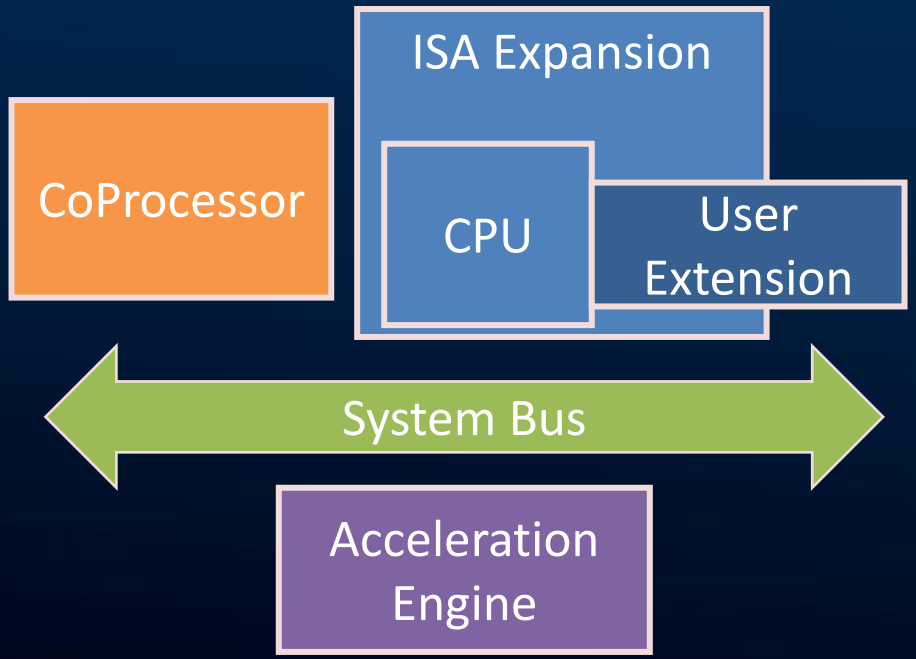
Standard IP Products?



Performance?
Power?
Area? Cost?

Standard IP products cannot satisfy all requirements
➔ Application Specific Accelerations ➔

Evolution of Computing Acceleration



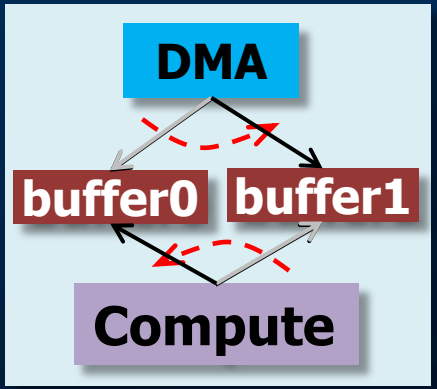
- **HW Acceleration Engine**
  - Cypto engines
- **Co-processors**
  - Graphic Processing Unit (GPU)
- **ISA expansion**
  - RISC-V M-,P-,V- extensions
- **User extensions**
  - RISC-V open ISA allows custom instruction

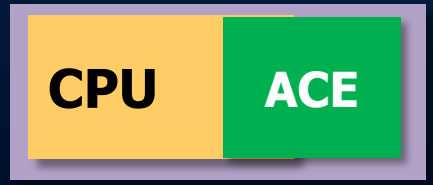
Comparison of Acceleration Methods

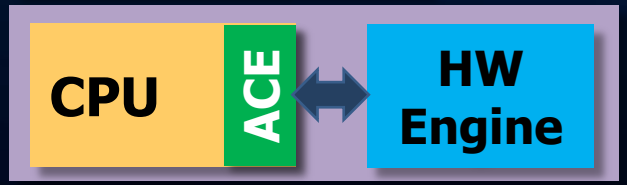
	HW Engine	Co-Processor	ISA Expansion	User Extension
Start-up latency	Longest	Long	None	None
Resource sharing	None	Decode/Control	Control & RF	Control & RF
Implementation Freedom	A TON	Lots	None	Restricted
Proprietary advantage	Yes	Yes	None	Yes
Best for	Very heavy Semantics	Medium Semantics	Commoditized Computation	Low-Medium Semantics

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™

- user.ace
- concise Verilog
- attributes

- scalar/vector
- wide operands
- direct IO

COPILOT

Custom-OPtimized Instruction deveLOpment Tools

Extended Tools + Compiler Asm/Disasm Debugger IDE

Extended ISS + CPU ISS (near-cycle accurate)

Extended RTL + CPU RTL

Extensible Baseline Components

A new CPU and its tools

Automated Env. For Cross Checking

Test Case Generator

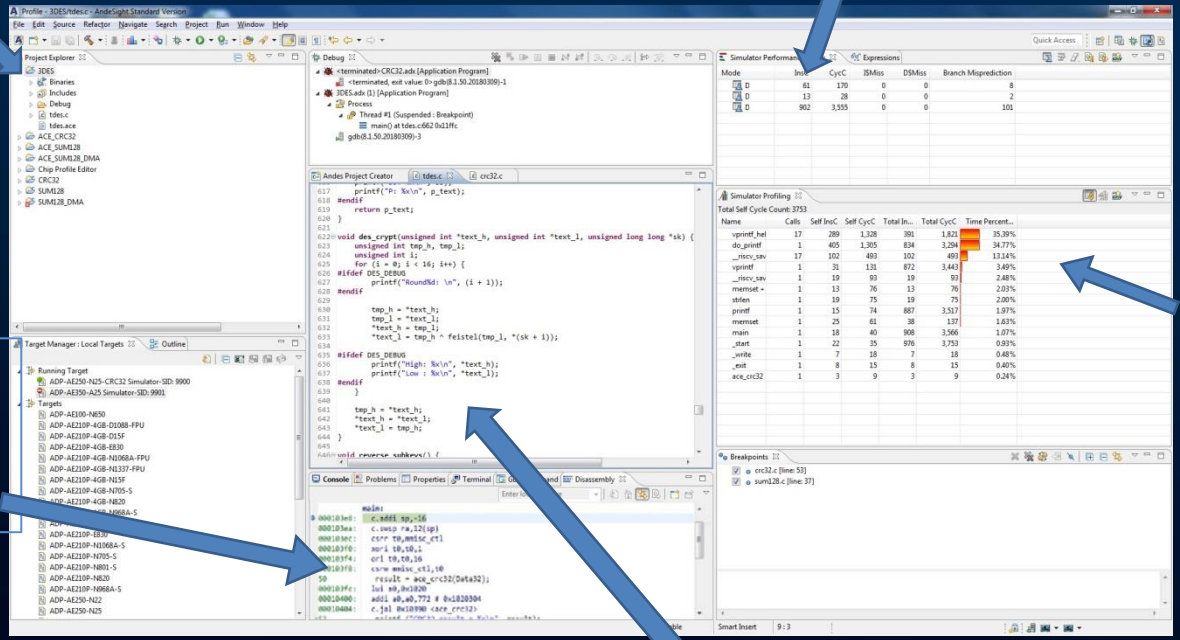
Extended ISS

Extended RTL

COPILOT w/ AndeSight IDE

Highly Integrated with AndeSight

Near Cycle Accurate Simulator Supported



Monitoring Customized Instructions

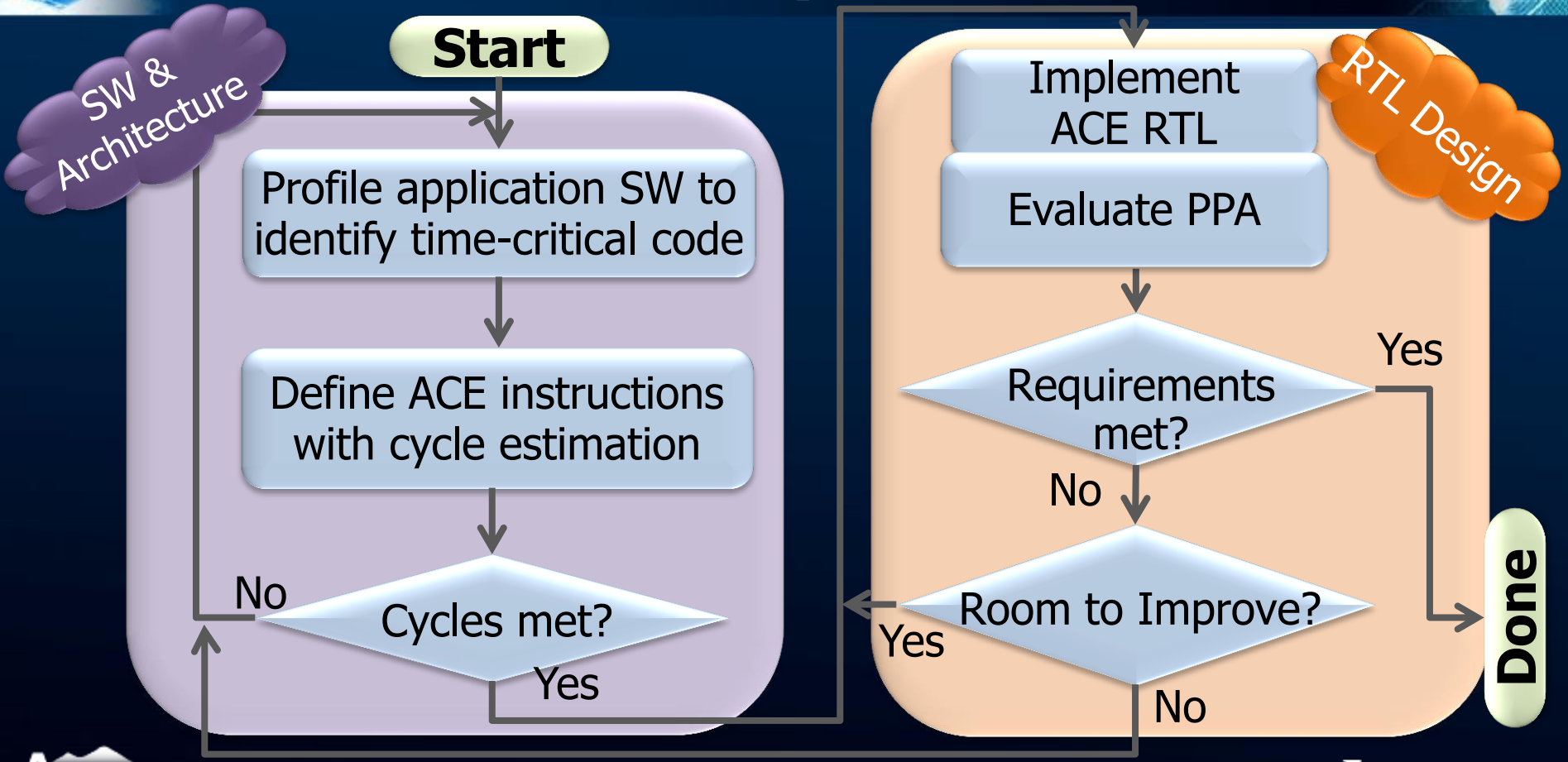
Profiling Tool Provides Easy Spotted Result

Hassle-Free Intrinsic Functions/API Replacement

ACE Features

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

ACE Development Flow



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 * (fetch + decode + retire) + N*execute
  - With ACE: only 1 instruction fetch, decode, retire
    - 1 * (fetch + decode + retire) + more efficient logic for N* execute

Andes' Core + ACE Instructions	FIR Filter	CRC32	3DES
Speedup*	20x	>140x	>300x
Power efficiency*	30x	>240x	>300x

* Estimated gains with and without corresponding ACE instructions

Madd32: A half-page ACE design

madd32.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.v

```
//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
   . . .          //custom logic
//ACE_END
```

- **madd32.ace: ACE definition file**
  - **insn**: define a scalar instruction, "madd32"
  - **op(erand)**: 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: Application C Code

Pure C Code

```
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);     //lower 16 bits
            + (C[i] >> 16)   *(X[i] >> 16);       //upper 16 bits
   return rslt;
}
```

With ACE

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

out

in

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

vmadd32: Vectorizing madd32

Vector Instruction

```
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;
};
```

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

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

for loop, repeat "cnt" times

Memory address automatically increases 1 for next iteration

per-iteration operation and latency

csim & RTL: same as scalar version !

```
// ACE_BEGIN: vmadd32
...
// ACE_END
```

per-iteration logic

bvmadd32: Backtorizing madd32

Background Vector Instruction

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

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

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

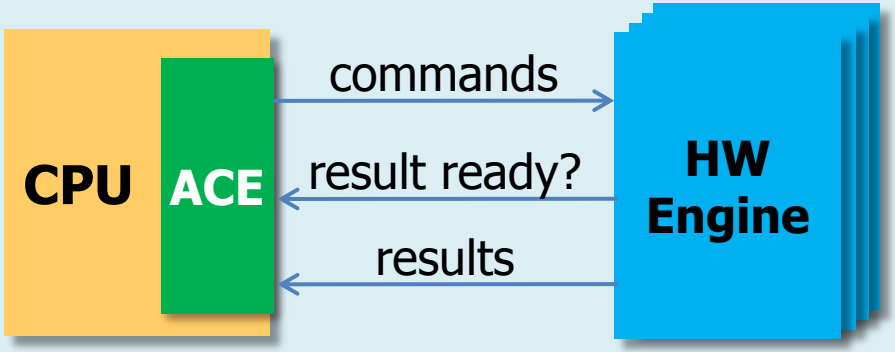
Everything else:
same as foreground version !!!

```
// ACE_BEGIN: vmadd32
...
// ACE_END
```

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)

```
port command {      //a 90-bit output port to
                    // all 4 HW engines
    width= 90;      //including a valid bit and
                    // a HW engine ID
    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

```
//ACE_BEGIN: madd32
   ...
  (*ace_shared*)
   sxtn_by_sxtn mult1( … );

  (*ace_shared*)
   sxtn_by_sxtn mult2( … );
   ...
//ACE_END
```

different

same

```
//ACE_BEGIN: msub32
  ...
  (*ace_shared*)
   sxtn_by_sxtn mult1( … );

  (*ace_shared*)
   sxtn_by_sxtn mult2( … );
  ...
//ACE_END
```

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

16x16.v

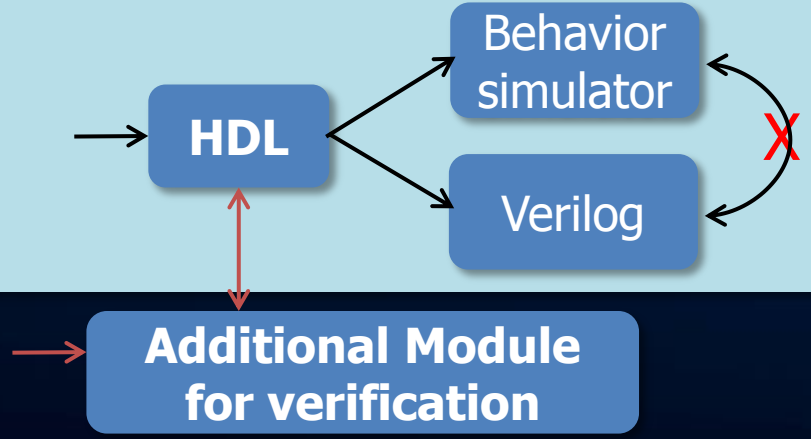
```
module sxtn_by_sxtn (
 output [31:0] rslt,
 input [15:0]  x,
 input [15:0]  y
);

assign rslt= x * y;
endmodule
```

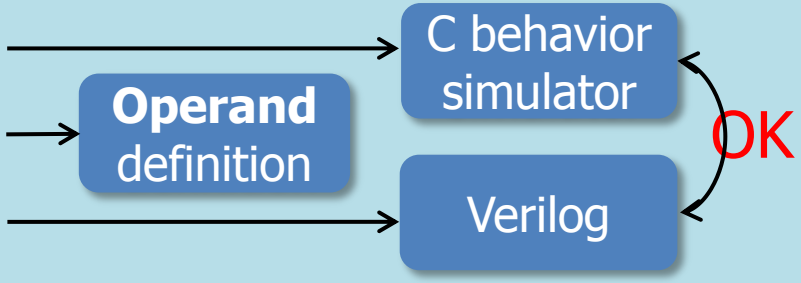
Uniqueness in Verification

■ Incorporate auto-verification in the flow

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



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

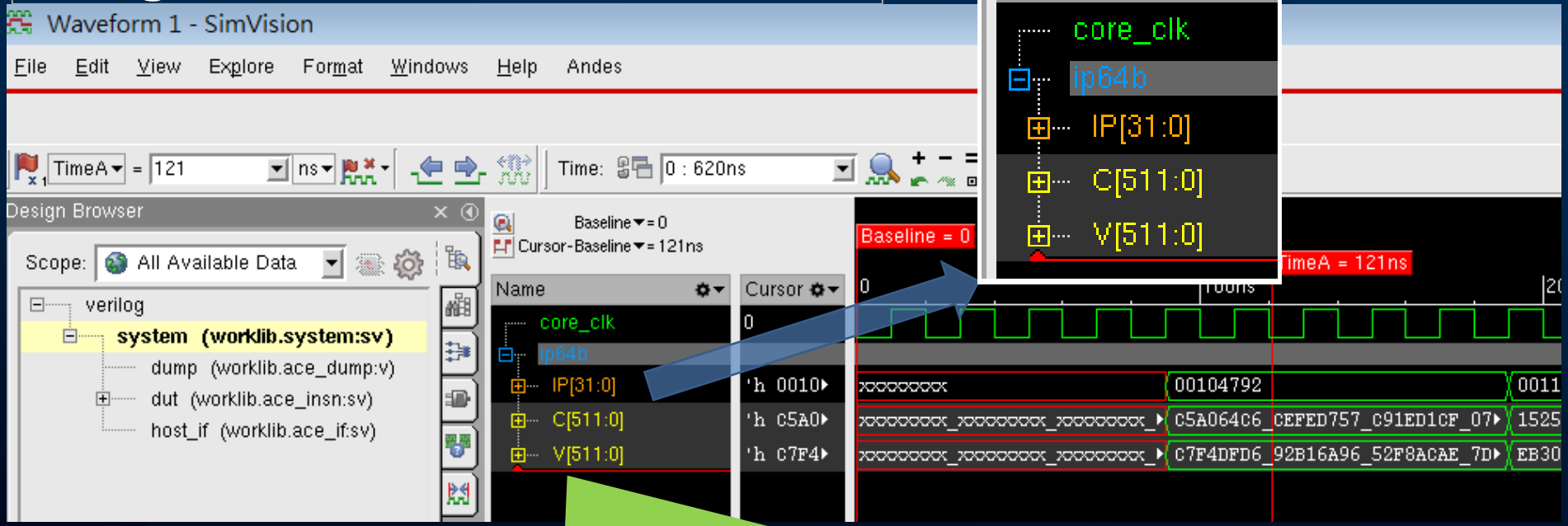


■ Use standard languages you are already familiar with

- Plus **ACE script** to guide the tool to do housekeeping work for you

RTL Debugging

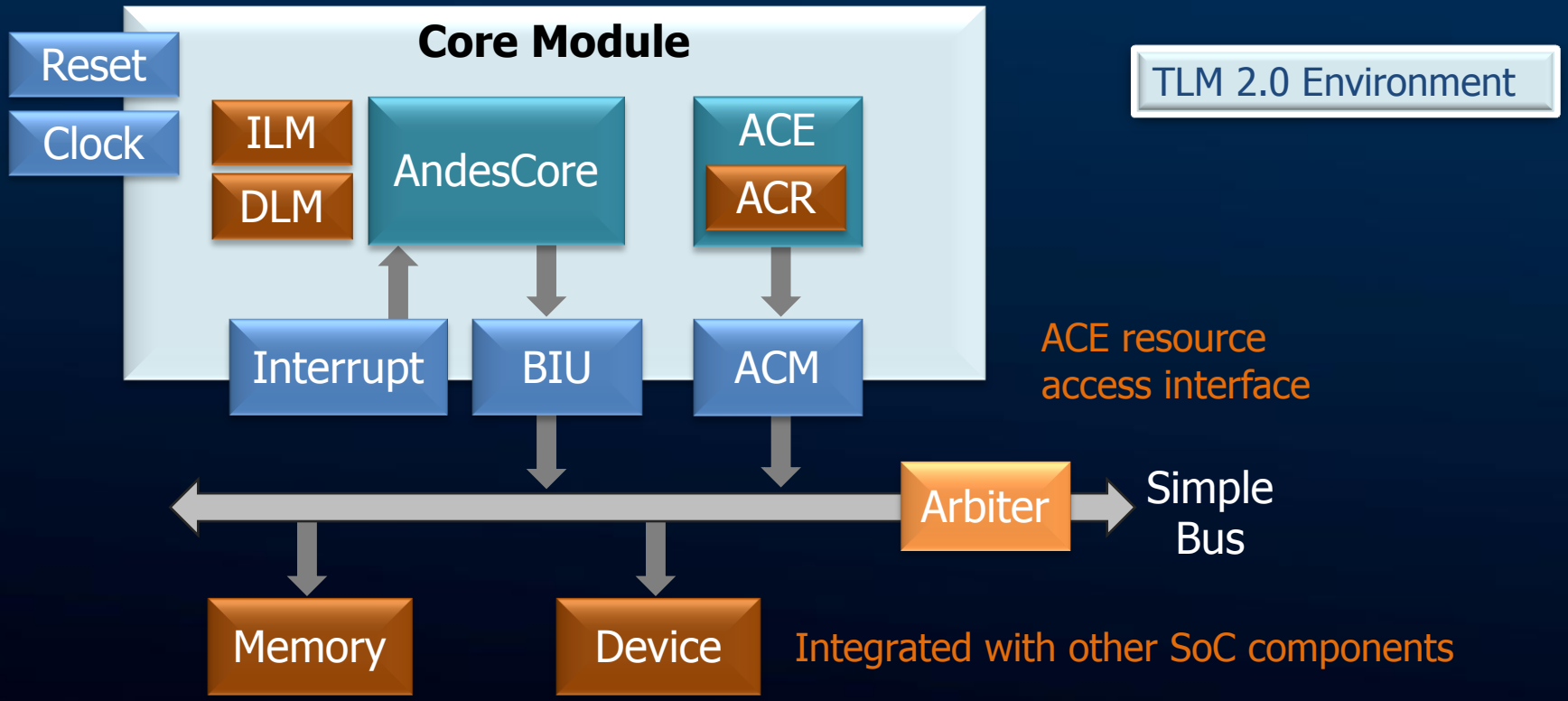
Auto-generate waveform control file



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, …)



Inner Product of Vectors with 64 8-bit Data

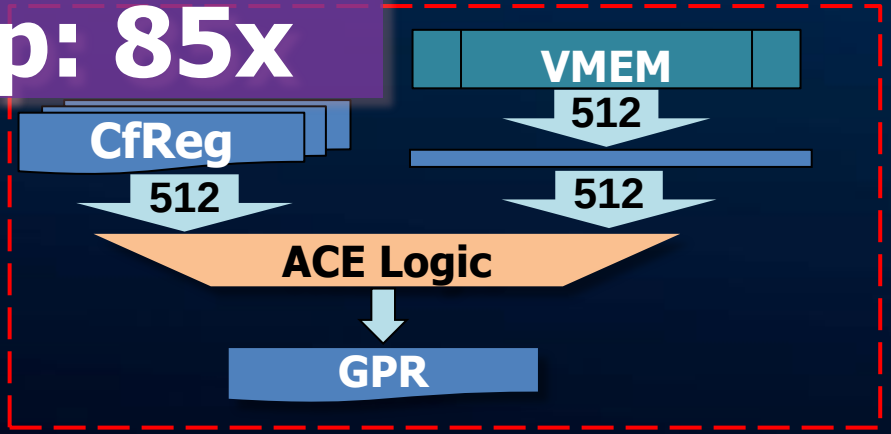
```
reg CfReg {                 //coef. Custom Register
    num= 4;
    width= 512;
}
ram  VMEM {                 //vector Custom Memory
     interface= sram;
     address_bits= 3;              //8 elements
     width= 512;
}
insn ip64B {
  operand= {out gpr IP,
                in CfReg C, in VMEM V};
  csim= %{                  //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
};
```

ip64B.ace

```
//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
```

ip64B.v

Speedup: 85x



Intrinsic: long **ace_ip64B**(CfReg_t, VMEM_t);

Madd32 w/ Ring Buffers on XY Memory

madd32rb.ace

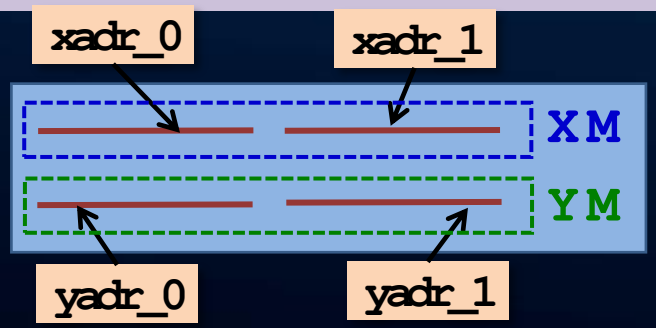
```
insn madd32rb {                    //with ring buffer
  op= {io gpr acc,
       in XM@xadr:u data, in YM@yadr:u coef};

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

madd32rb.ace

```
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
```



- 2-entry address registers **xadr** point to 2 ring buffers in **XM** without keeping reloading their initial values.
- 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