Tutorial

New tool innovations

SYNOPSYS®
Silicon to Software™

Under the Hood of ASIP Designer

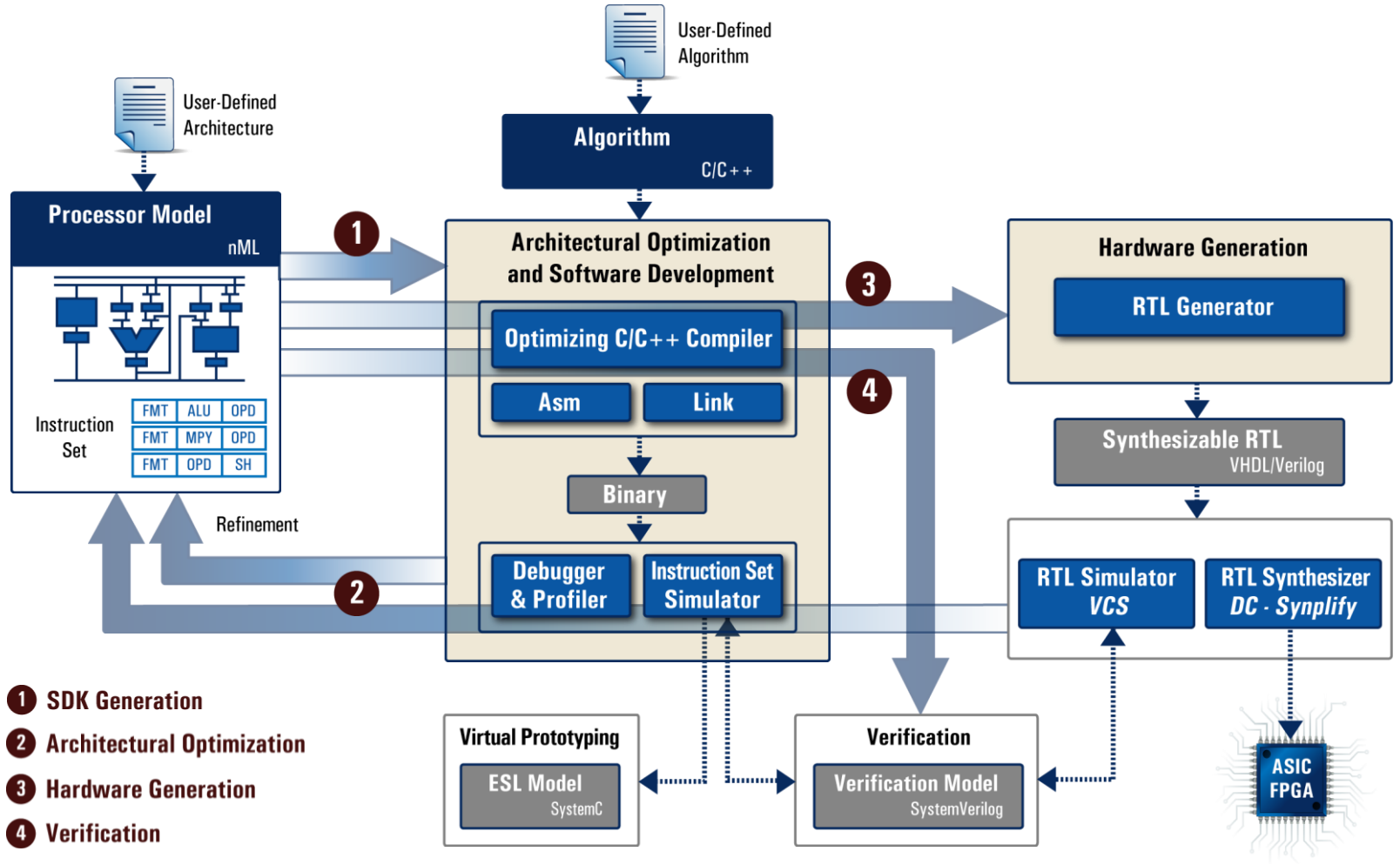
Key Innovations in the ASIP Designer Tool-Suite

Gert Goossens
Sr. Director of Engineering

ASIP University Day – November 17, 2021

ASIP Designer

Tool Flow



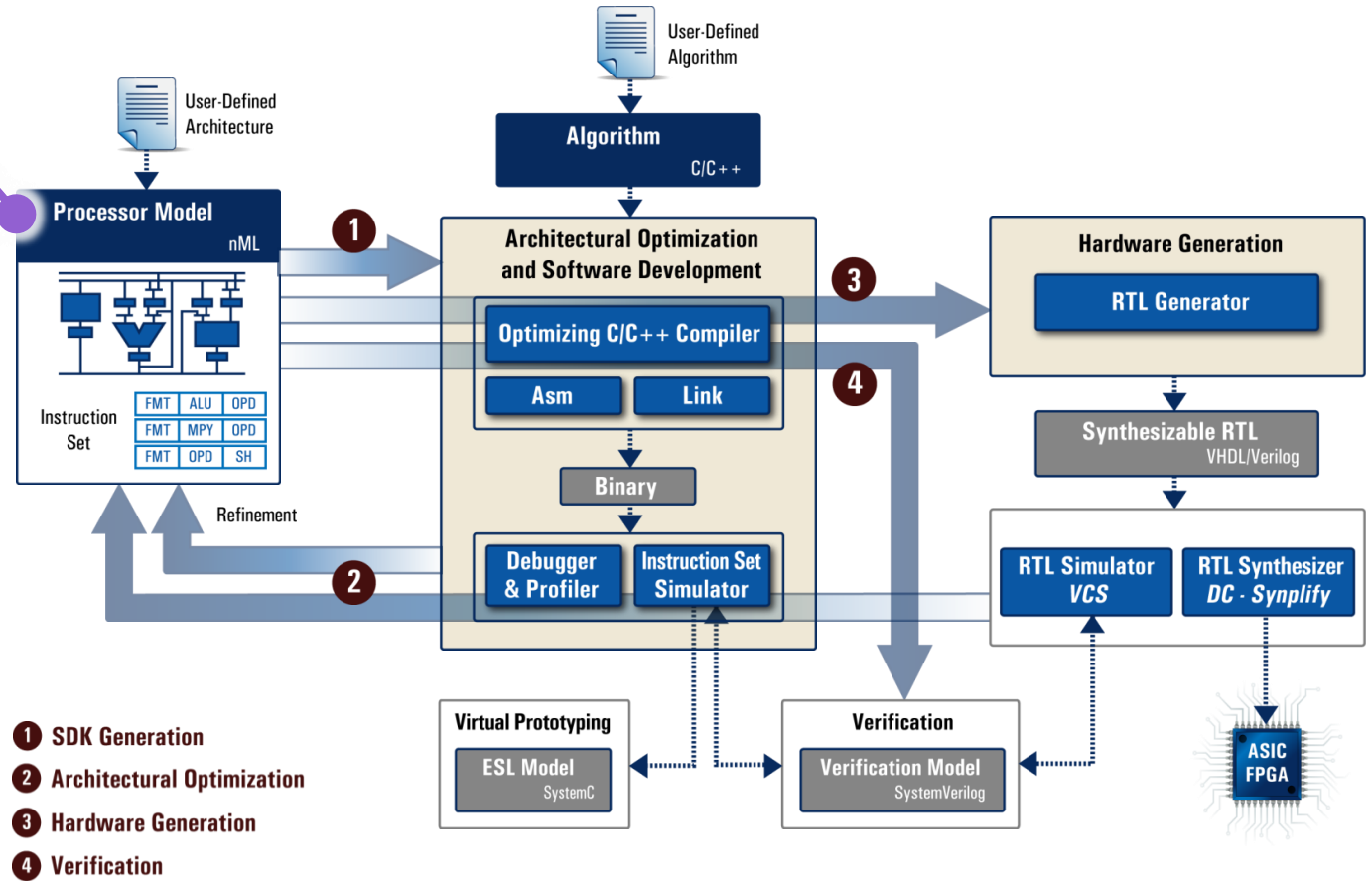
Supported design steps

- Modeling of instruction-set architectures: nML language
- Automatic generation of software development kit, including an efficient C compiler
- Algorithm-driven architectural exploration: **Compiler-in-the-Loop**™
- Automatic generation of synthesizable RTL **Synthesis-in-the-Loop**™
- Design verification

Tool Innovations in ASIP Designer

Processor modeling

- nML: instruction-set and μ-architecture
- PDG: datapath, PCU & I/O interface behavior



Processor Modeling

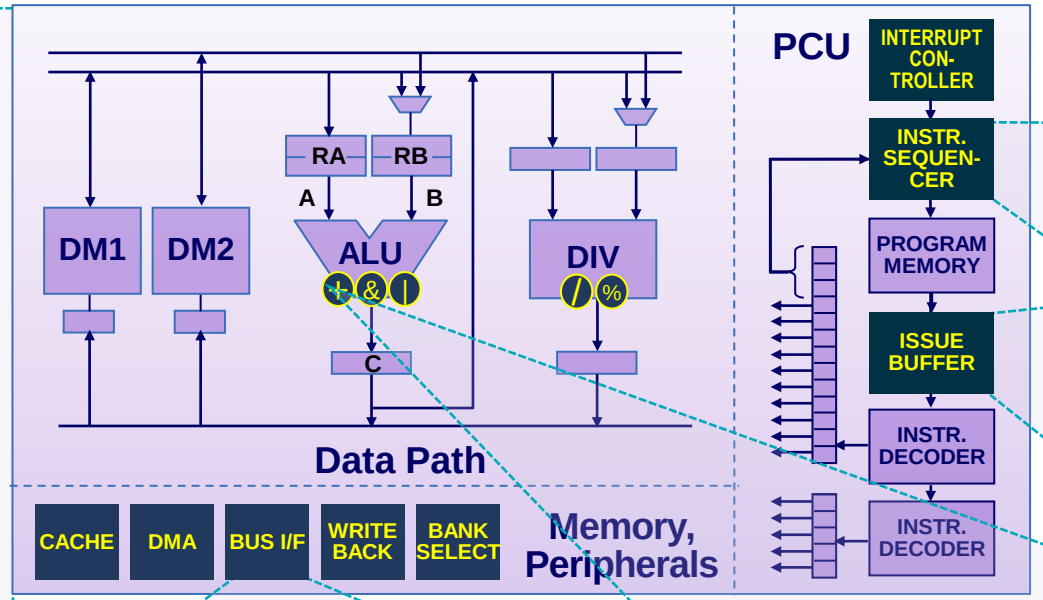
Instruction-Set & Micro-Architecture — nML

```
// Resource definition
   mem DM1[1024]<word,addr>;
   reg RA[2]<word,uint1>;
   reg RB[2]<word,uint1>;
   trn A<word>; trn B<word>;
   pipe C<word>;
   fu alu;
   ...

// Instruction-set grammar
   opn my_core (arith_inst | ctrl_inst);

   opn arith_inst (a:alu_inst,
     d: div_inst, l:load_store_inst);

   opn alu_inst (op:opcod, x:c1u, y:c1u,
     z:c1u) {
     action {
       stage EX1:
         A = RA[x];
         B = RB[y];
         switch (op) {
         case add: C = add(A, B) @alu;
         case and: C = and(A, B) @alu;
         case or: C = or(A, B) @alu;
         ...
         }
       stage EX2:
         RA[z] = C @alu;
     }
     syntax: op " RA" y ", RB" x ", RA" z;
     image: "0"::op::x::y::z;
   }
   ...
```



PCU Behavior — PDG

```
void my_asip::
user_next_pc(){
  // manipulation of
  // program counter
}
```

```
void my_asip::
user_issue(){
  // creation of issue
  // packets from
  // program words
}
```

I/O Interface Behavior — PDG

```
io_interface my_bus_if(){
  void process_result(){
    // transactions before
    // processor actions
  }
  void process_request(){
    // transactions after
    // processor actions
  }
}
```

Datapath Behavior — PDG

```
// 16-bit saturating addition
word add(word a, word b) {
  int17_t x = (int17_t)a + (int17_t)b;
  if (x > MAX) x = MAX;
  else if (x < MIN) x = MIN;
  return x[15:0];
}
```

Processor Modeling

nML Components

New in ASIP Designer
R-2021.03

- Objective: new concept in nML processor description language, to
  - Introduce modularity
  - Encourage reuse of code fragments, within and across processor models
- Benefits
  - Reduced development effort
  - Increased maintainability of processor models

```
// Component definition
component alu_slot(mode mR) {
   fu  au;
   trn tA <word>;  trn tB <word>;  trn tC <word>;

   opn alu_instr (op: alu_op, a: mR, b: mR) {
      ...
      tC = add (tA = a, tB = b) @ au;
      ...
   }
}

// Instantiation
instantiate alu_slot alu0(mRx);
instantiate alu_slot alu1(mRy);

// Use of rule names in other rules
opn par_alu (a : alu0.alu_instr, b : alu1.alu_instr) {
   ...
}

// Use of transitory/fu names in other rules
opn sp_add(imm : c32s) {
   action {
      spw = alu1.tC = add(alu1.tA = spr, alu1.tB = imm)@alu1.au;
   }
}
```

Processor Modeling

nML Components

New in ASIP Designer
R-2021.03

nML component
- Any portion of nML code that is to be reused can be made an nML component
- nML components are named

nML component instantiation
- nML components can be instantiated multiple times
- Each instance is named
- Any element from the component can be reused in an instance, by referring to **<instance name>.<element name>**

- Example: reuse of an operation rule

- Example: reuse of transitories and functional units

```
// Component definition
component alu_slot(mode mR) {
   fu  au;
   trn tA <word>;  trn tB <word>;  trn tC <word>;

   opn alu_instr (op: alu_op, a: mR, b: mR) {
      ...
      tC = add (tA = a, tB = b) @ au;
      ...
   }
}

// Instantiation
instantiate alu_slot alu0(mRx);
instantiate alu_slot alu1(mRy);

// Use of rule names in other rules
opn par_alu (a : alu0.alu_instr, b : alu1.alu_instr) {
   ...
}

// Use of transitory/fu names in other rules
opn sp_add(imm : c32s) {
   action {
      spw = alu1.tC = add(alu1.tA = spr, alu1.tB = imm)@alu1.au;
   }
}
```

Anecdotal: Trv32p3 processor model extended to static dual-issue machine, using nML components
- Effort: < 1 person-day
- Performance: 3.3 → 5.15 CoreMark/MHz

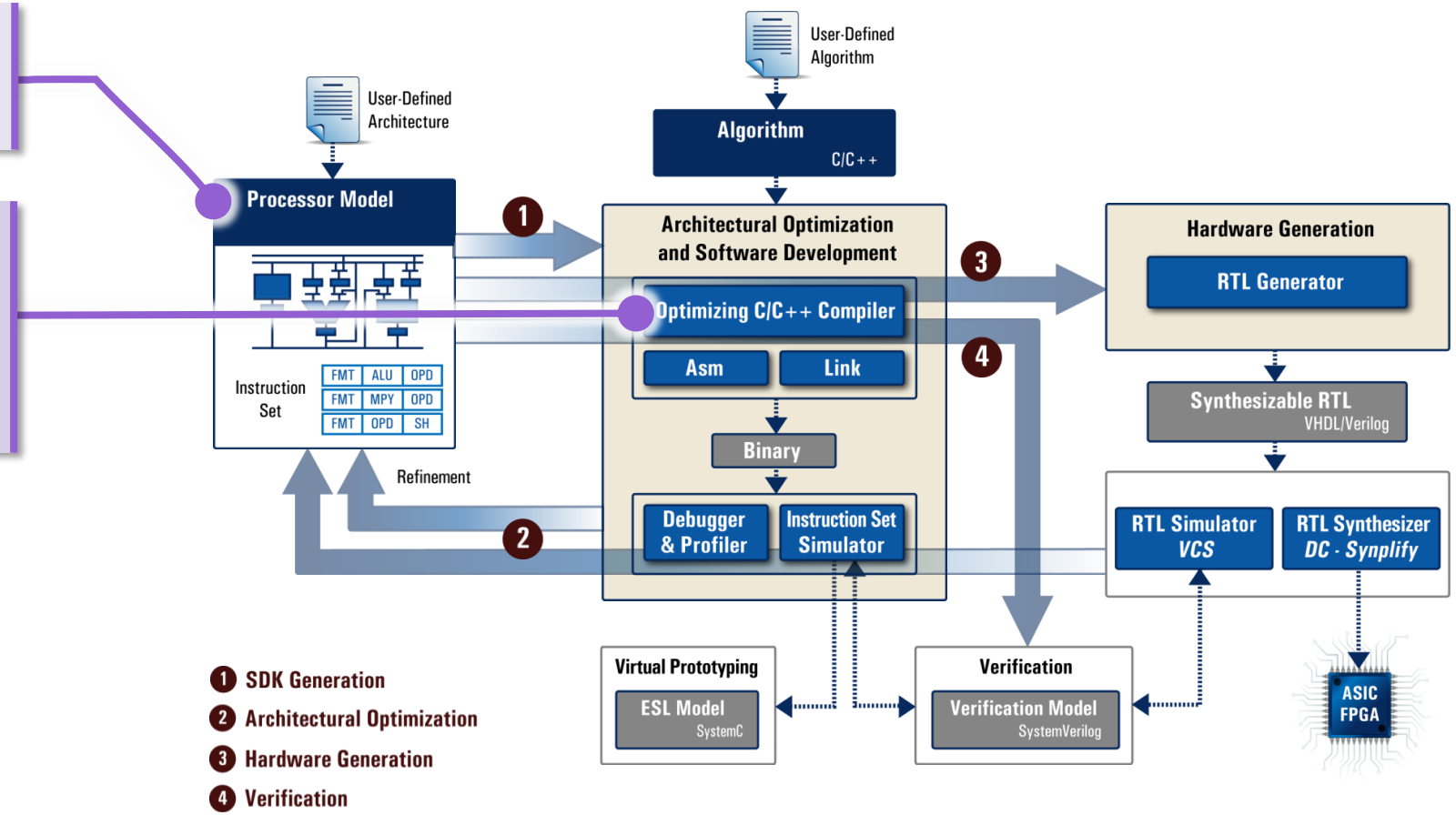
Tool Innovations in ASIP Designer

Processor modeling

- nML: instruction-set and μ-architecture
- PDG: datapath, PCU & I/O interface behavior

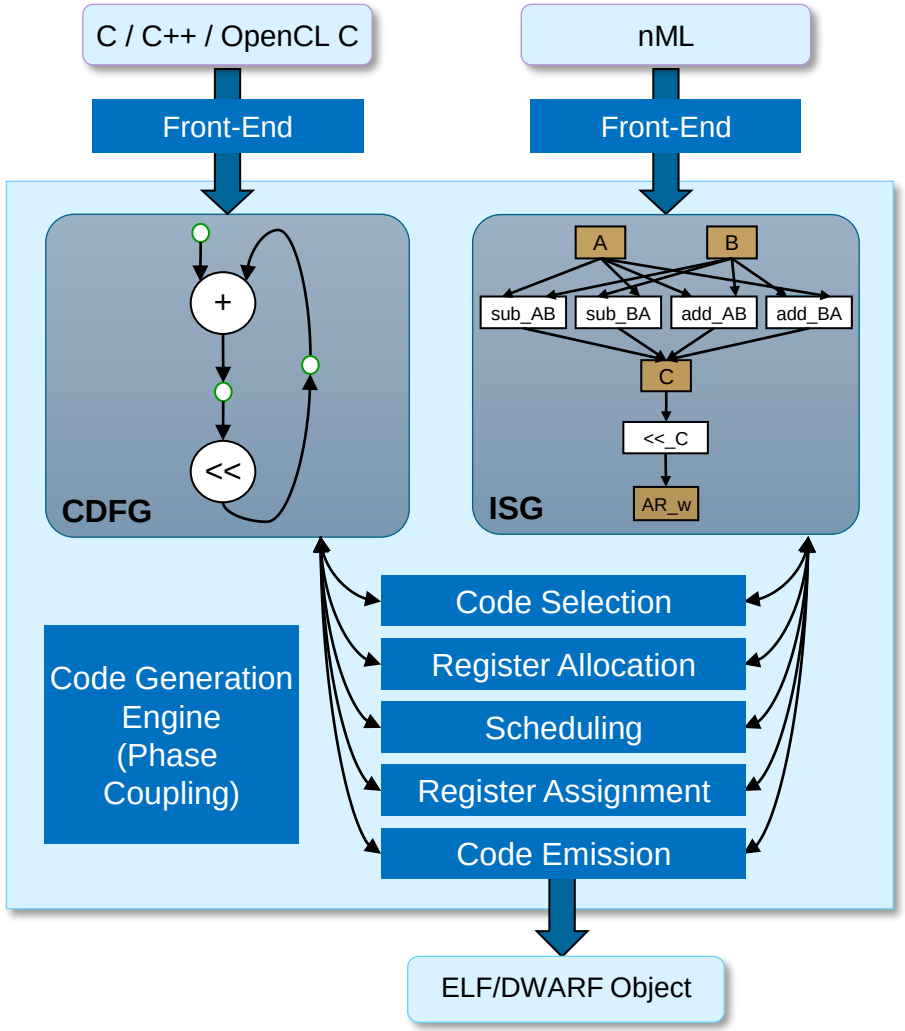
Retargetable compilation

- Original and LLVM-based front-ends
- Retargetable back-end, exploiting heterogeneous parallel architectures
- Enables unique “Compiler-in-the-Loop” methodology for architectural optimization



Retargetable Compilation

- C language front-end
  - C / C++ / OpenCL C → “Control-Data Flow Graph” (CDFG)
- nML front-end
  - nML processor model → “Instruction-Set Graph” (ISG)
  - ISG is closer to HW than other compilers’ machine models
    - HW resources, data types, connectivity, instruction encoding, instruction-level parallelism, instruction pipeline
  - Supports heterogeneous parallel architectures
- Graph-based compiler back-end (code generator)
  - Maps CDFG onto ISG, in multiple optimization phases
  - Graph-based optimization algorithms work on any ISG
  - Steered by code generation engine
    - Phase coupling through prediction and backtracking (iteration)
  - Combines retargetability with high code efficiency
  - Patented



Retargetable Compilation

LLVM Front-End Extended for ASIP Architectures

Extended to support ASIP architectures
- Datatypes
- Memory model
- DSP optimizations

Combine high-level optimizations from LLVM and original front-end

C / C++

Clang

LLVM IR

LLVM Optimizer

LLVM IR

Partial LLVM Back-End

CDFG

Original Front-End

CDFG

User can choose: LLVM or original front-end

LLVM front-end	Original front-end
• C and C++ support	• C and OpenCL C support
• Additional high-level optimizations	• Better user-control: preferred during archit. exploration

C / OpenCL C

C / C++ / OpenCL C

Front-End

nML

Front-End

CDFG

ISG

A B sub_AB sub_BA add_AB add_BA C <<_C AR_w

Code Generation Engine (Phase Coupling)

Code Selection

Register Allocation

Scheduling

Register Assignment

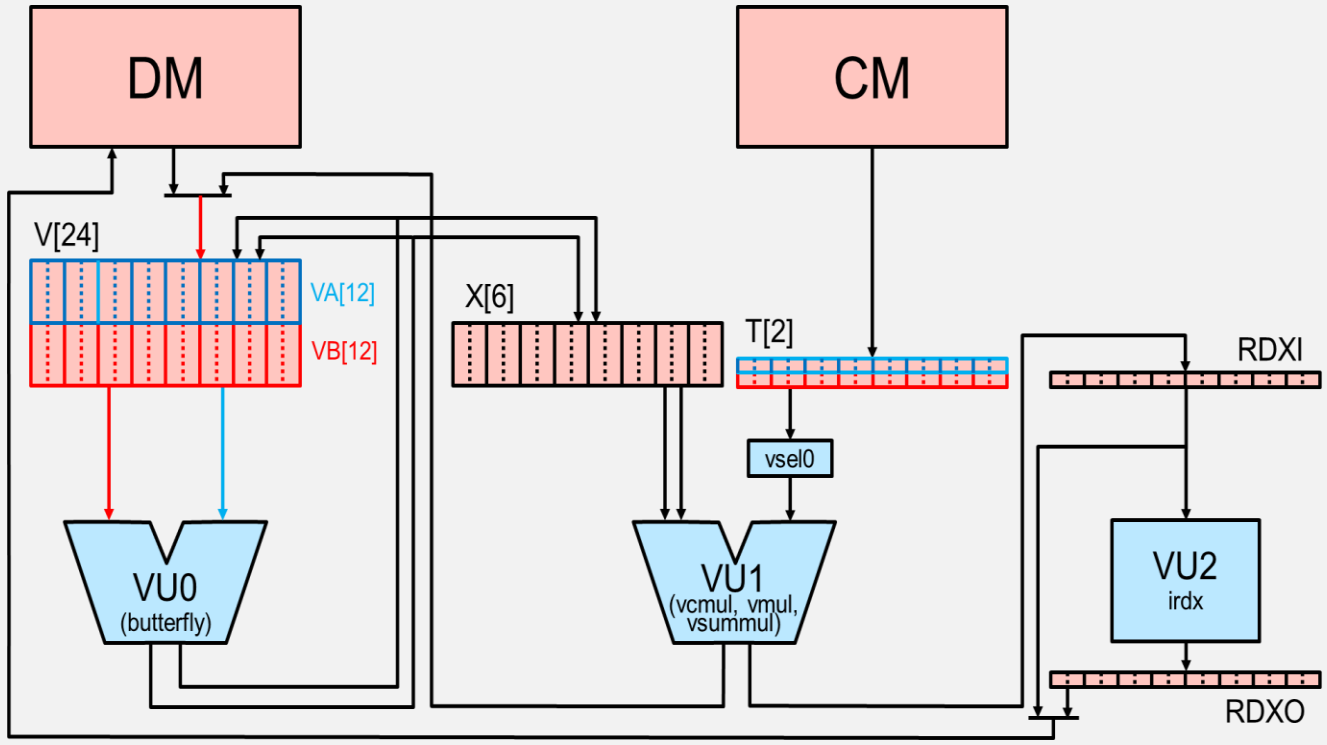
Code Emission

ELF/DWARF Object

Retargetable Compilation

Compiler Back-End Can Exploit Advanced Architectural Features

- Instruction-level parallelism (ILP)
  - 2..10 static issue slots typically
  - Variable-length instructions
- Data-level parallelism (SIMD)
  - Multiple vector widths
  - 2..128 lanes, 16..2K bits typically
- Deep instruction pipeline
  - 5..10 stages typically
  - Pipeline hazard protection: SW stalls, register bypasses, HW stalls
- Heterogeneous memory & register structure
  - Multiple memories & register files
  - Dedicated connectivity



▲ PrimeCore ASIP

Optimized for FFT & DFT in 4G/5G baseband

- ILP: 5-slot VLIW, 16/32/64-bit instructions
- SIMD: 256-bit, 8 complex lanes
- Instruction pipeline: 5 stages

Retargetable Compilation

Compiler Back-End Can Exploit Advanced Architectural Features

Tutorial

▼ C source code for PrimeCore

```
inline void vmodule10io(vcmplx_t
  i0,..i9, vcmplx_t& o0,..o9, base_t
  c5_1, cmplx_t cm1, cmplx_t cm2)
{
  vcmplx_t t0,t1,t2,t3,t4,t5,t6,t7,
    t8,t9;

  vmodule5io(i0,i6,i2,i8,i4,
    t0,t6,t2,t8,t4, c5_1, cm1, cm2);
  vmodule5io(i5,i1,i7,i3,i9,
    t5,t1,t7,t3,t9, c5_1, cm1, cm2);

  vcbf0(t0,t5,o0,o5,0,0,SCALE);
  vcbf0(t6,t1,o6,o1,0,0,SCALE);
  vcbf0(t2,t7,o2,o7,0,0,SCALE);
  vcbf0(t8,t3,o8,o3,0,0,SCALE);
  vcbf0(t4,t9,o4,o9,0,0,SCALE);
}

inline void vmodule5io(vcmplx_t i0,..,
  i4, vcmplx_t& o0,..o4, base_t c1,
  cmplx_t cm1, cmplx_t cm2)
{
  vcmplx_t t1,t2,t3,t4,t5,t6,t7,
    y1,y2,y3,y4,y6;
  vcbf0(i1,i4,t1,t3,0,0,SCALE);
  vcbf0(i2,i3,t2,t4,0,0,SCALE);
  vcbf0(t1,t2,t5,t6,0,0,SCALE);
  y6 = t6 * c1;
  vcbf0(i0,t5,o0,t7,SCALE*2,2,SCALE);
  vcbf0(t7,y6,y1,y2,0, 0,0);
  y3 = vsummul(t3,t4,cm1);
  y4 = vsummul(t3,t4,cm2);
  vcbf1(y1,y4,o4,o1,0,0);
  vcbf1(y2,y3,o3,o2,0,0);
}
```

Application-specific data-types

Overloaded operator

Intrinsic function call

▼ Compiler-generated machine code on PrimeCore

VU0	VU1	DM	CM	VU2/CTL

```
vmod10_s1 void_vmod10_s1__Pvcmplx_t___Pvcmplx_t___sint
397 nop                                  ; nop                       ; v03=dm[p0+6*m0] ; nop             ; p2=#0
398 v04=v03                              ; nop                       ; v13=dm[p0+4*m0] ; nop             ; r0=m0
399 v06=(v04 +v13 )/2; x01=(v04 -v13 )/2; nop                       ; v15=dm[p0+7*m0] ; tc=cm[p2]       ; p2=#640
400 v05=v15                              ; nop                       ; v02=dm[p0+2*m0] ; nop             ; m1=#16
401 nop                                  ; nop                       ; v14=dm[p0+8*m0] ; nop             ; nop
402 v17=(v02 +v14 )/2; x05=(v02 -v14 )/2; nop                       ; v01=dm[p0+1*m0] ; nop             ; nop
403 ...         )/2; x07=(v06 -v17 )/2; nop                       ; v12=dm[p0+3*m0] ; nop             ; nop
404 ...         )/2; x00=(v05 -v12 )/2; v14 = vmul(x07,tc[[1]])   ; v15=dm[p0+9*m0] ; nop             ; nop
405 v07=(v01 +v15 )/2; x03=(v01 -v15 )/2; nop                       ; v00=dm[p0+0*m0] ; nop             ; nop
406 v10=(v00/4+v18/4)/2; v09=(v00/4-v18/4)/2; v13 =vsmul(x03,x00,tc[0]); v03=dm[p0+5*m0] ; nop      ; nop
407 v19=(v07 +v16 )/2; x07=(v07 -v16 )/2; nop                       ; nop             ; nop             ; do r0,429
408 v20=(v03/4+v19/4)/2; v11=(v03/4-v19/4)/2; v15 = vmul(x07,tc[[1]]) ; nop           ; [p0+m1]         ; nop
409 x06=(v10 +v20 )/2; x07=(v10 -v20 )/2; v12 =vsmul(x01,x05,tc[1]); nop             ; tw=cm[p2+5*m0]  ; nop
410 v08=(v11 +v15 )  ; v11=(v11 -v15 )  ; v1...                     ; v03=dm[p0+6*m0] ; nop             ; ...
411 v23=(v11 +v13r ) ; v21=(v11 -v13r ) ; ...                       ; v02=dm[p0+2*m0] ; ...             ; ...
412 v09=(v09 +v14 )  ; v10=(v09 -v14 )  ; r...                      ; v13=dm[p0+4*m0] ; tw=cm[...       ; ...
413 v11=(v10 +v15r ) ; v10=(v10 -v15r ) ; v15 =vsmul(x03,x00,tc[1]); v14=dm[p0+8*m0] ; nop             ; rdxo=irdx(rdxi)
414 x07=(v11 +v23 )/2; x05=(v11 -v23 )/2; nop                       ; dm[p1+5*m0]=rdxo; nop             ; rdxo=irdx(rdxi)
415 x02=(v10 +v21 )/2; x07=(v10 -v21 )/2; rdxi=vcmul(x07,tw)        ; dm[p1+0*m0]=rdxo; tw=cm[p2+7*m0]  ; nop
416 v22=(v08 +v15r ) ; v23=(v08 -v15r ) ; rdxi=vcmul(x07,tw)        ; v15=dm[p0+7*m0] ; tw=cm[p2+6*m0]  ; nop
417 v10=(v09 +v12r ) ; v11=(v09 -v12r ) ; nop                       ; v12=dm[p0+3*m0] ; nop             ; rdxo=irdx(rdxi)
418 x07=(v11 +v23 )/2; x04=(v11 -v23 )/2; nop                       ; dm[p1+8*m0]=rdxo; nop             ; rdxo=irdx(rdxi)
419 x07=(v10 +v22 )/2; x06=(v10 -v22 )/2; rdxi=vcmul(x07,tw)        ; dm[p1+7*m0]=rdxo; tw=cm[p2+4*m0]  ; nop
420 v04=v03           ;v05=v15           ; rdxi=vcmul(x07,tw)        ; v01=dm[p0+1*m0] ; tw=cm[p2+3*m0]  ; nop
421 v06=(v04 +v13 )/2; x01=(v04 -v13 )/2; rdxi=vcmul(x05,tw)        ; v15=dm[p0+9*m0] ; tw=cm[p2+2*m0]  ; rdxo=irdx(rdxi)
422 v17=(v02 +v14 )/2; x05=(v02 -v14 )/2; rdxi=vcmul(x02,tw)        ; dm[p1+6*m0]=rdxo; tw=cm[p2+1*m0]  ; rdxo=irdx(rdxi)
423 v16=(v05 +v12 )/2; x00=(v05 -v12 )/2; rdxi=vcmul(x04,tw)        ; dm[p1+4*m0]=rdxo; tw=cm[p2+9*m0]  ; rdxo=irdx(rdxi)
424 v18=(v06 +v17 )/2; x07=(v06 -v17 )/2; rdxi=vcmul(x06,tw)        ; dm[p1+3*m0]=rdxo; [p2+m1]         ; rdxo=irdx(rdxi)
425 v07=(v01 +v15 )/2; x03=(v01 -v15 )/2; v14 = vmul(x07,tc[[1]])   ; dm[p1+2*m0]=rdxo; tw=cm[p2+5*m0]  ; rdxo=irdx(rdxi)
426 v19=(v07 +v16 )/2; x07=(v07 -v16 )/2; v12 =vsmul(x01,x05,tc[1]); v00=dm[p0+0*m0] ; nop             ; nop
427 v10=(v00/4+v18/4)/2; v09=(v00/4-v18/4)/2; v15 = vmul(x07,tc[[1]]) ; v03=dm[p0+5*m0] ; [p0+m1]   ; nop
428 v20=(v03/4+v19/4)/2; v11=(v03/4-v19/4)/2; v13 =vsmul(x03,x00,tc[0]); dm[p1+1*m0]=rdxo; nop      ; rdxo=irdx(rdxi)
429 x06=(v10 +v20 )/2; x07=(v10 -v20 )/2; nop                       ; dm[p1+9*m0]=rdxo; [p1+m1]         ; nop
430 rt
```

Loop scheduled in 20 cycles

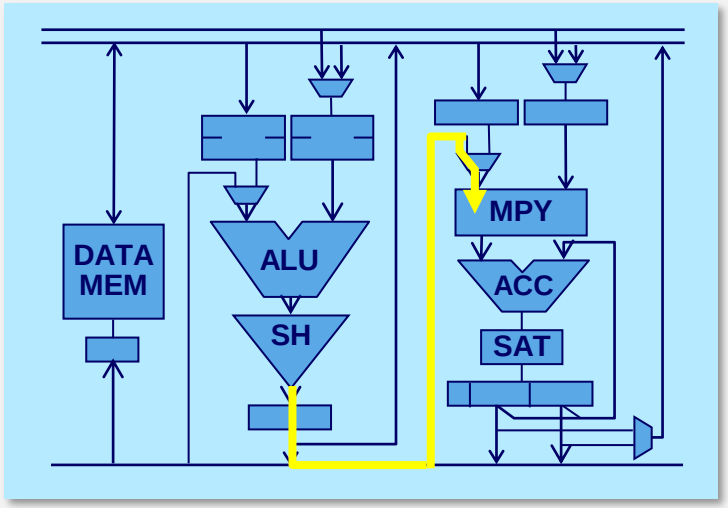
Zero-overhead loop

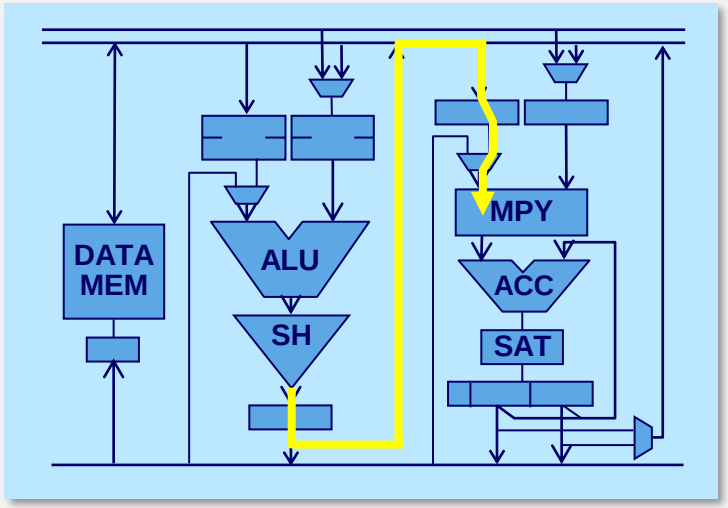
Critical slot filled: 19 butterflies

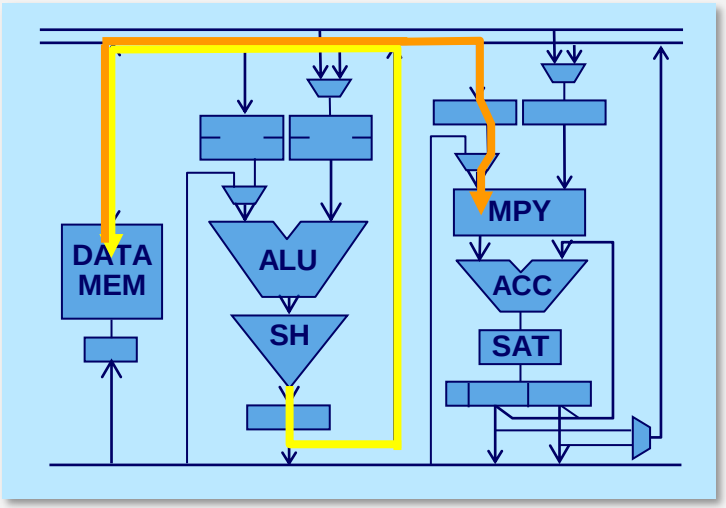
Critical slot filled: 20 loads or stores

Retargetable Compiler Back-End

Register Allocation for Heterogeneous Memory & Register Architectures



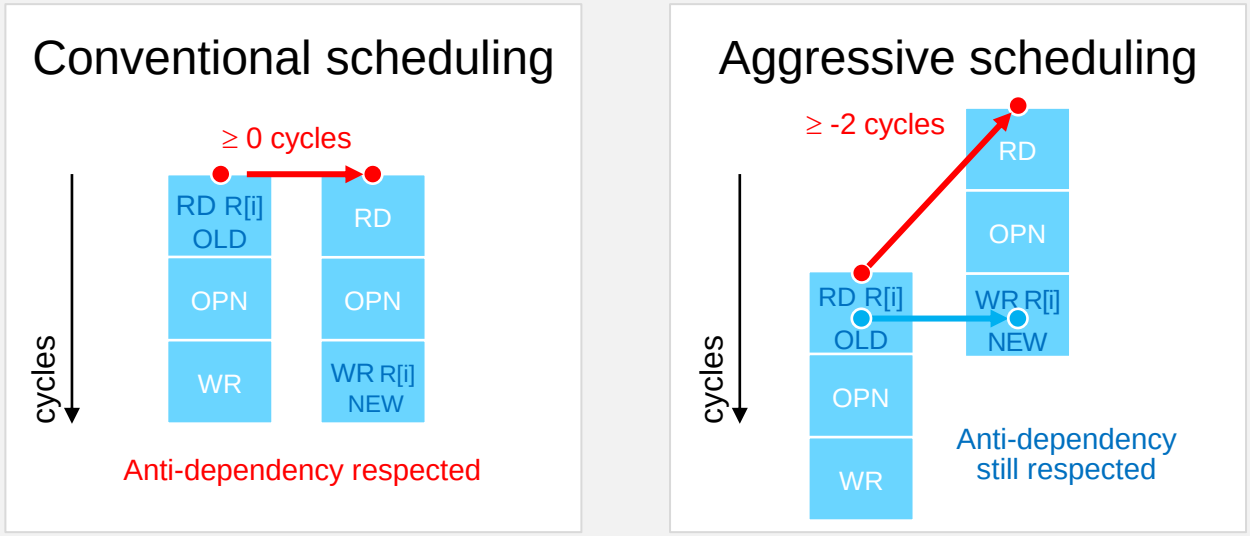




- Challenges
  - Alternative routing paths between computational resources: best choice depends on context
  - Mixed memory/register operands
  - Constraints: read-modify-write, coupled operands, sub-range access
- Graph-based data-routing technology
  - Trace-based graph search algorithms with branch-and-bound

Retargetable Compiler Back-End

Aggressive Scheduling for Deeply Pipelined, Parallel Architectures



- Aggressive scheduling option enables more scheduling freedom
  - Results in lower cycle count and lower register utilization
  - Efficient SW pipelining of zero-overhead loops, without need for loop-unrolling
- Aggressively scheduled code blocks are non-interruptable
  - Compiler sets control signal (flag) during execution of such regions, so that interrupts can be masked

FFT Size	Cycle Count *Conventional*	Cycle Count *Aggressive*	Gain
8	5	5	0.0%
16	16	14	12.5%
32	25	20	20.0%
64	42	29	31.0%
128	75	52	30.7%
256	248	180	27.4%
512	446	328	26.5%
1024	878	654	25.5%
2048	1700	1288	24.2%
4096	4412	3585	18.7%
		Average:	21.6%

▲ FFT implementation on PrimeCore ASIP (5-slot VLIW, 5-stage pipeline)

Tool Innovations in ASIP Designer

Processor modeling

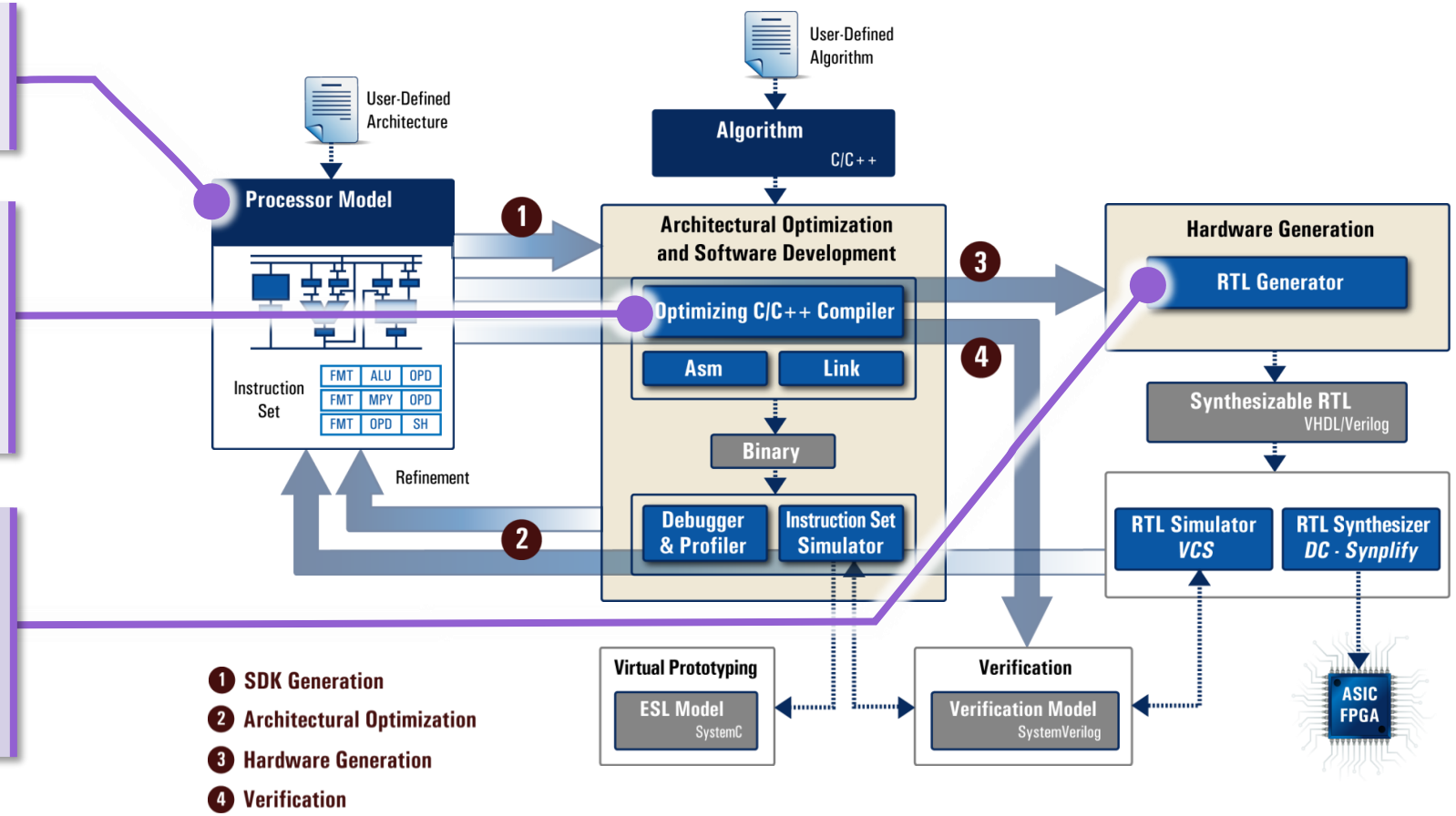
- nML: instruction-set and μ-architecture
- PDG: datapath, PCU & I/O interface behavior

Retargetable compilation

- Original and LLVM-based front-ends
- Retargetable back-end, exploiting heterogeneous parallel architectures
- Enables unique “Compiler-in-the-Loop” methodology for architectural optimization

RTL generation

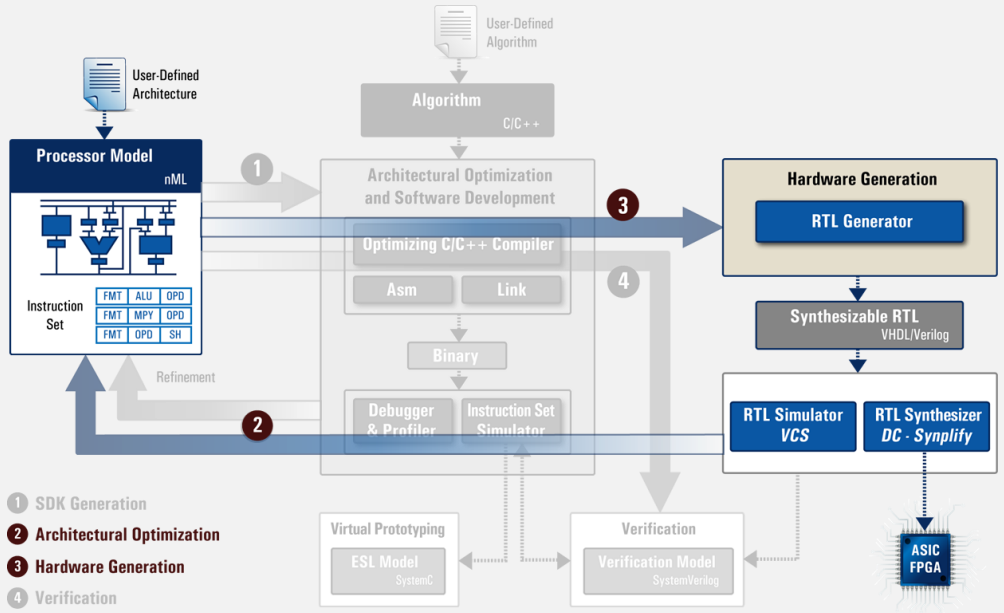
- Smooth integration with Synopsys’ implementation & verification flows
- Enables unique “Synthesis-in-the-Loop” methodology for (micro-)architectural optimization



PPA Estimation

New in ASIP Designer
S-2021.12

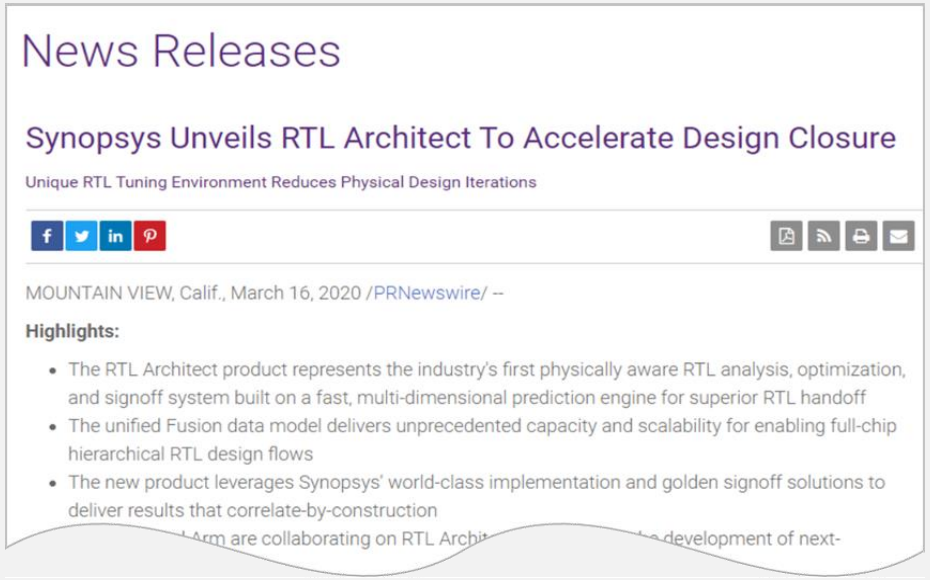
Synthesis-in-the-Loop™



To achieve desired performance/power/area (PPA):

- Tune μA in processor model (nML and PDG)
- Set RTL generator options to influence PPA tradeoffs and RTL style

RTL Architect™



News Releases

Synopsys Unveils RTL Architect To Accelerate Design Closure

Unique RTL Tuning Environment Reduces Physical Design Iterations

MOUNTAIN VIEW, Calif., March 16, 2020 /PRNewswire/ --

Highlights:

- The RTL Architect product represents the industry's first physically aware RTL analysis, optimization, and signoff system built on a fast, multi-dimensional prediction engine for superior RTL handoff
- The unified Fusion data model delivers unprecedented capacity and scalability for enabling full-chip hierarchical RTL design flows
- The new product leverages Synopsys' world-class implementation and golden signoff solutions to deliver results that correlate-by-construction

◀ Press Release, Mar 16, 2020

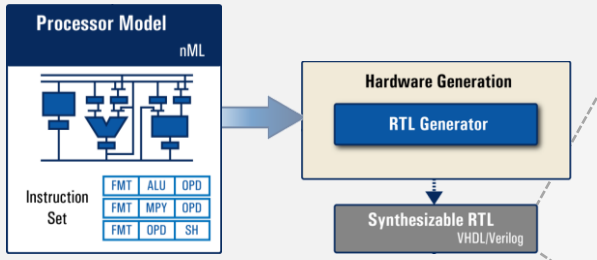
- Synopsys' new PPA optimization and prediction tool
- Up to 3x **faster PPA estimation** than full synthesis with Fusion Compiler, at ~ 95% accuracy
- Fast exploration by **"sweeping" of design parameters**
  → Can be used for comparative analysis of ASIP Designer's RTL generator options

PPA Estimation

“Sweeping” of Design Parameters in RTL Architect

New in ASIP Designer
S-2021.12

- Example: ASIP Designer’s RTL Generator option “**`instruction_pipe_write_enable`**”
  - Enables the insertion of clock-gates in the instruction decoder, resulting in power reduction




```
always @ (posedge clock or posedge reset)
begin : EX_stage_start
  if (reset)
  begin
    reg_EX_valid <= 1'b0;
    reg_IR_EX <= NOP;
    EX_enabling <= 0;
  end
  else
  begin
    reg_EX_valid <= trn_EX_valid;
    reg_IR_EX <= reg_IR_ID;
    EX_enabling <= predec_EX_enabling;
  end
end
```

`instruction_pipe_write_enable: 0;`
No clock gating
“block 1”

```
always @ (posedge clock or posedge reset)
begin : EX_stage_start
  if (reset)
  begin
    reg_EX_valid <= 1'b0;
    reg_IR_EX <= NOP;
    EX_enabling <= 0;
  end
  else
  begin
    reg_EX_valid <= trn_EX_valid;
    if (trn_EX_valid)
      reg_IR_EX <= reg_IR_ID;
    EX_enabling <= predec_EX_enabling;
  end
end
```

`instruction_pipe_write_enable: 1;`
Enable clock gating for:
- instruction pipeline register in decoder

“block3”

```
always @ (posedge clock or posedge reset)
begin : EX_stage_start
  if (reset)
  begin
    reg_EX_valid <= 1'b0;
    reg_IR_EX <= NOP;
    EX_enabling <= 0;
  end
  else
  begin
    reg_EX_valid <= trn_EX_valid;
    if (trn_EX_valid)
      reg_IR_EX <= reg_IR_ID;
    if (trn_EX_valid || reg_EX_valid)
      EX_enabling <= predec_EX_enabling;
  end
end
```

`instruction_pipe_write_enable: 2;`
Enable clock gating for:
- instruction pipeline register in decoder
- opcode registers connecting to datapath

“block2”

  - RTL Architect’s QoR Summary Table

QORsum

Runs | Metrics | Info | Settings

QOR Summary
Host Info
App Option Details
Timing / LDRC

	Run	Setup			Netlist			Power				Routability	Parame…
		WNS	TNS	NVE	Util	StdCellArea	StdCells	TotalPwr	LeakPwr	Gated%	Bits/Flop	Overflow%	block
1	block1	-0.06	-1.4	30	78.6	5510	15655	764088	88	85.5	1.00	2.09	block1
2	block3	0.0%	-14.3%	0.0%	0.5%	0.7%	-0.1%	-12.7%	1.6%	-3.9%	0.0%	9.6%	block3
3	block2	0.0%	7.1%	3.3%	-0.3%	-0.1%	-0.9%	-11.6%	-1.4%	-2.1%	0.0%	2.9%	block2

Conclusions

- ASIP Designer is deployed world-wide in advanced SoC designs for exciting application domains
- Universities are at the forefront of new applications

- Synopsys differentiates through unique tool technologies
  - Processor modeling languages
  - Retargetable compilation
  - Compiler-in-the-Loop™
  - Synthesis-in-the-Loop™
  - Interoperability with Synopsys & 3rd-party tools

Tutorial

- Our tool innovation continues, so that users can address next application challenges
  - Broader architectural scope
  - Better design performance, power, area (PPA)
  - Easier adoption of ASIPs

New tool innovations

SYNOPSYS®
Silicon to Software™

Thank You