Institute of SCIENCE TOKYO

RISC-V AI Accelerator Design Platform on C2RTL System Design Verification Framework

Tsuyoshi Isshiki

Dept. of Information and Communications Engineering
Institute of Science Tokyo

June 17th, 2025

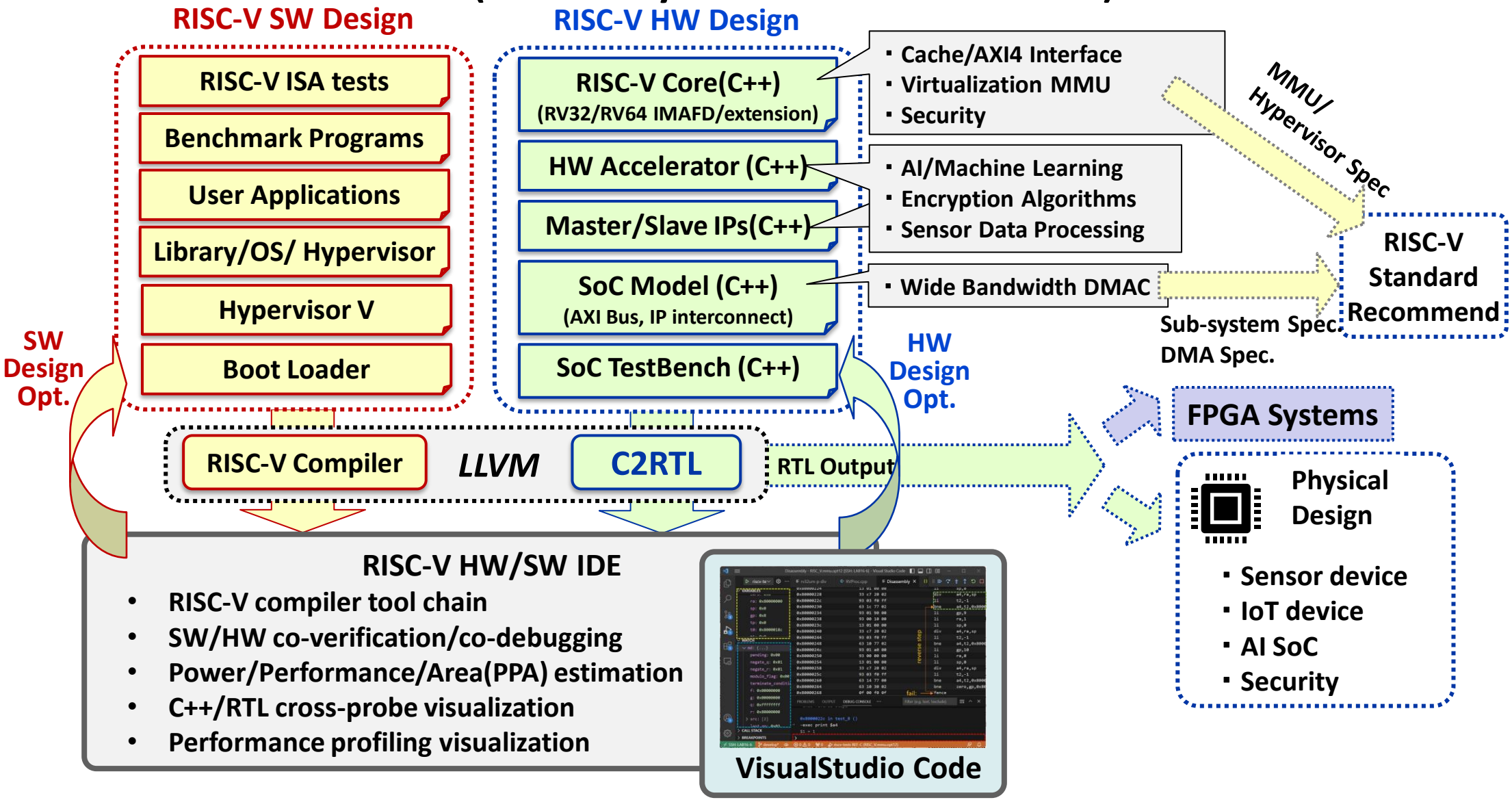
RISC-V Based AI SoCs

- **25 billion RISC-V Based AI SoCs by 2027 (Semico Research)**
  - **RISC-V Based AI SoC revenue : $291B (2027)**
  - **RISC-V CPU Semiconductor IP (SIP) 34.9% CAGR through 2027**
  - **RISC-V SW ecosystem expands adoption in consumuer/enterprise SoC markets**
  - **RISC-V ratifications of 15 new specifications : vector, scalar cryptography, hypervisor**
- **Design opportunities and challanges of RISC-V architecture**
  - **RISC-V ISA specifications help build SW ecosystems for RISC-V based SoCs**
  - **Large opportunities for drastic improvements in compute/power efficiencies with *custom instruction extensions* and *HW accelerations***

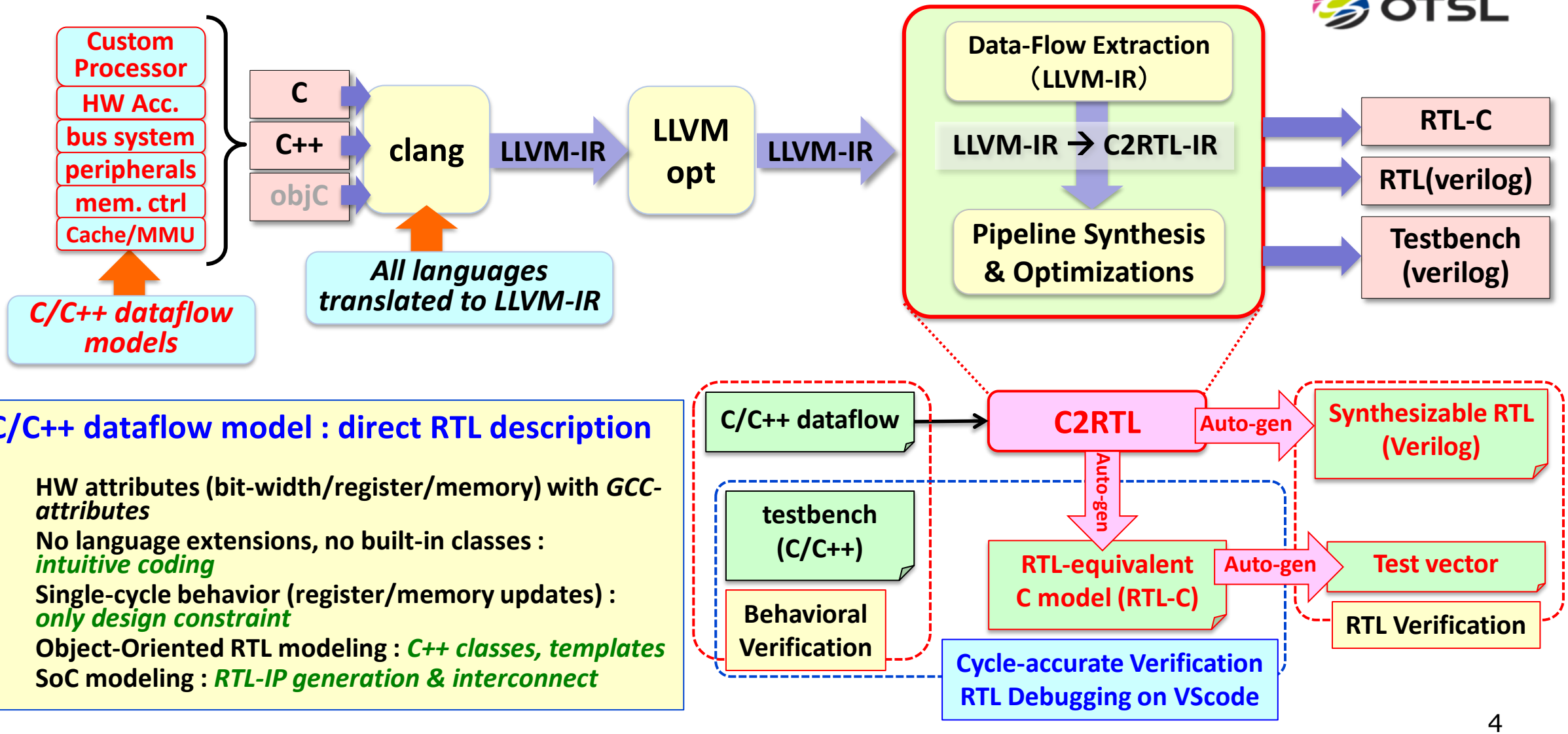
→ ***Need System-Level SW/HW Design Environment for efficiently building customized RISC-V cores, HW accelerators and SoC architectures***

RISC-V HW/SW Integrated Design Environment (IDE)

(Funded by NEDO: 2022.8 – 2025.3)



LLVM-based C2RTL Framework



C/C++ dataflow model : direct RTL description

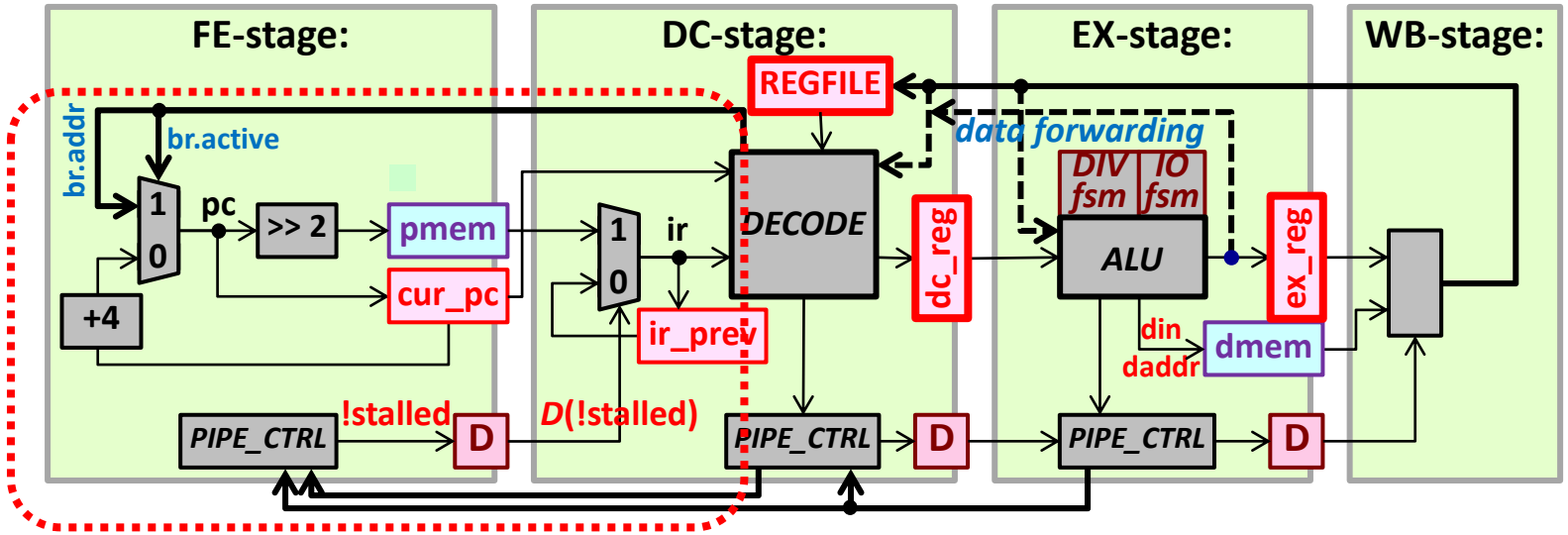
- HW attributes (bit-width/register/memory) with *GCC-attributes*
- No language extensions, no built-in classes : *intuitive coding*
- Single-cycle behavior (register/memory updates) : *only design constraint*
- Object-Oriented RTL modeling : *C++ classes, templates*
- SoC modeling : *RTL-IP generation & interconnect*

RISC-V Processor Pipeline Behavior Description

```
int CPU :: step (AXI4L::CH *axi)
{
  fetch();
  decode();
  execute(axi);
  writeback();
  return (cpu.halted == 1);
}
```

RTL top function
Defines the single cycle behavior of the total system

C++ constraint : register/memory variables updated at most once inside the RTL top function
→ C++ code directly converts to RTL




```
void CPU :: fetch() {
  fe_sig.pctl.stalled  =  dc_sig.pctl.stall |
                          ex_sig.pctl.stall;
  if ( ! fe_sig.pctl.stalled ) {
    UINT32 pc = (dc_sig.br.active) ?
                dc_sig.br.addr : fe_reg.cur_pc + 4;
    fe_reg.cur_pc = pc;
    ir = pmem[pc >> 2];
    ir_prev = ir;
  } else { ir = ir_prev;  }
}
```

RISC-V Instruction Decode Stage C++ Description

```
enum RVFields {
   RVF_funct7,
   RVF_funct3,
   RVF_funct2,
   RVF_rs3,
   RVF_rs2,
   RVF_rs1,
   RVF_rd,
   RVF_opc,
   RVF_imm12H,
   RVF_imm5L,
   RVF_imm20H,
   RVF_count,
};
```

```
struct Field { SINT32 msb, bits; };
Field  FLD[ RVF_count ]  =  {
   { 31, 7 },   /// RVF_funct7
   { 14, 3 },   /// RVF_funct3
   { 26, 2 },   /// RVF_funct2
   { 31, 5 },   /// RVF_rs3
   { 24, 5 },   /// RVF_rs2
   { 19, 5 },   /// RVF_rs1
   { 11, 5 },   /// RVF_rd
   {  6, 7 },   /// RVF_opc
   { 31, 12 }, /// RVF_imm12H
   { 11, 5 },   /// RVF_imm5L
   { 31, 20 }, /// RVF_imm20H
};
```

```
enum RVInsnOpCode {
   /// opc[6:0]
   RVI_lui      = 0x37,
   RVI_auipc    = 0x17,
   RVI_jal      = 0x6f,
   RVI_jalr     = 0x67,
   RVI_br       = 0x63,
   RVI_ld       = 0x03,
   RVI_st       = 0x23,
   RVI_compi    = 0x13,
   RVI_comp     = 0x33,
   RVI_fence    = 0x0f,
   RVI_sys      = 0x73,
};
```

RISC-V ISA format
- **Instruction field**
- **Opcode**

```
#define GET_BITS(d, m, b)   (((d) >> ((m) - (b) + 1)) & ((1 << (b)) - 1))
#define GET_FLD(fid)        (GET_BITS (ir, FLD[fid].msb, FLD[fid].bits))
#define DEC_FLD(fid)        insn.fid = GET_FLD (RVF_##fid)

/// R : funct7[31:25], rs2[24:20], rs1[19:15], funct3[14:12], rd[11:7], opc[6:0]
void CPU :: dec_R ()
{  DEC_FLD(funct7); DEC_FLD(rs2); DEC_FLD(rs1); DEC_FLD(funct3); DEC_FLD(rd); }

/// S : imm11_5[31:25], rs2[24:20], rs1[19:15], funct3[14:12], imm4_0[11:7], opc[6:0]
void CPU :: dec_S () {
   DEC_FLD(funct7); DEC_FLD(rs2); DEC_FLD(rs1); DEC_FLD(funct3); DEC_FLD(imm5L);
   SINT32 i11 = ir & 0x80000000;      /// sign bit
   UINT32 i10_5 = (insn.funct7) & 0x3f; /// 6 bits : funct7 --> same as imm11_5[31:25]
   insn.imm = (i11 >> 20) | (i10_5 << 5) | insn.imm5L;
}
```

```
void CPU :: decode() { . . . .
  DEC_FLD(opc);
  switch (insn.opc) {
    case RVI_lui:     dec_U(); . . .
    case RVI_auipc: dec_U(); . . .
    case RVI_jal:     dec_UJ(); br_type = 1; . .
    case RVI_jalr:    dec_I();   br_type = 2; . .
    case RVI_br:      dec_SB(); br_type = 3; . .
    case RVI_ld:       dec_I(); . . .
    case RVI_st:       dec_S(); . . .
    case RVI_compi: dec_I(); . . .
    case RVI_comp:  dec_R(); . . .
    . . .
```

RISC-V instruction decoder function for each opcode
→ ***adding new instruction is simple***

RISC-V EX-stage/WB stage C++ Description

```
void CPU::execute() { . . . .
  switch (dc_reg.op) { . . . .
    case RVO_ld:  /// LOAD
      wb_sig.dout = format_rd (dmem[daddr], byte_pos); /// shift-right, sign/zero extend
      break;
    case RVO_st:  /// STORE
      st_data = format_wd (src2, byte_pos);                  /// shift-left
      write_mem (&dmem[daddr], byte_en (byte_pos), st_data); /// write with byte-enable
      break;
    case RVO_comp:   /// compute
      BIT sign0 = (src0 >> 31), sign1 = (src1 >> 31);
      BIT ultFlag = (src0 < src1);
      BIT sraFlag = (dc_reg.sextFlag && sign0);
      UINT32 shamt = (src1 & 0x1f);
      switch (dc_reg.funct3) {
        case RVF3_add:  ex_out = add_out;                    break;
        case RVF3_slt:    ex_out = sign0 ^ sign1 ^ ultFlag;  break;
        case RVF3_sltu:  ex_out = ultFlag;                   break;
        case RVF3_shl:   ex_out = src0 << shamt;             break;
        case RVF3_shr:  ex_out = (src0 >> shamt) | ((sraFlag) ? ~(0xffffffffu >> shamt) : 0); break;
        case RVF3_xor:  ex_out = src0 ^ src1;                bre
        case RVF3_or:    ex_out = src0 | src1;               br
        case RVF3_and: ex_out = src0 & src1;                 br
      }
      break; . . . .
```

EX-stage : description for each op-type

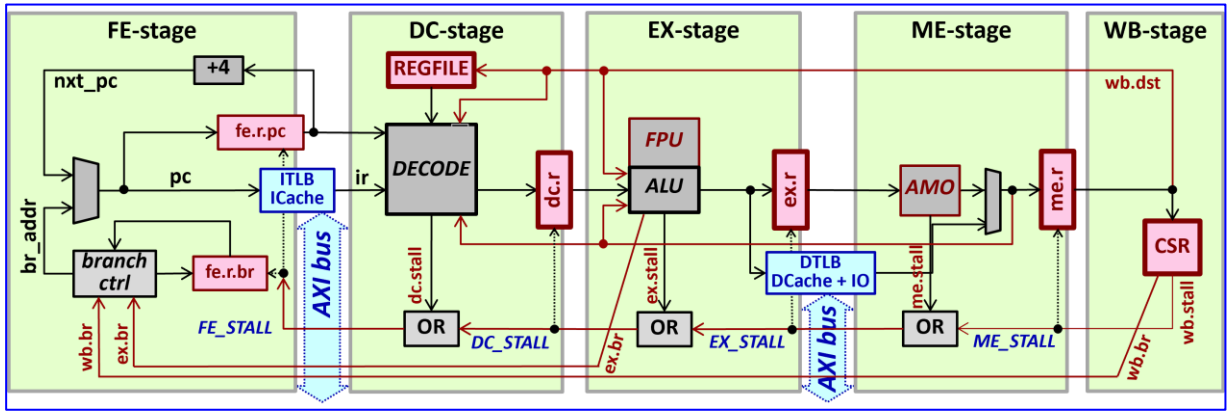
```
void write_mem (UINT32 *mem, UINT4 be, UINT32 din) {
  UINT8 *cmem = (UINT8 *)mem;
  UINT8 *cdin = (UINT8 *)&din;
  if (be & 0x1) { cmem[0] = cdin[0]; }
  if (be & 0x2) { cmem[1] = cdin[1]; }
  if (be & 0x4) { cmem[2] = cdin[2]; }
  if (be & 0x8) { cmem[3] = cdin[3]; }
}
```

Memory-Write with byte-enable

WB-stage : writeback to regfile

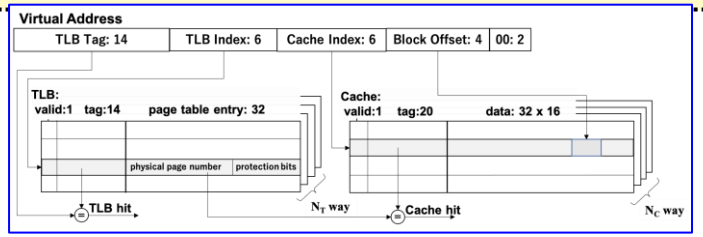
```
void CPU::writeback() { . . . .
  UINT32 wb_data = (ex_reg.wbflag == W_dout) ? wb_stt.dout : ex_reg.out;
  wb_sig.pctl.stalled  = ex_sig.pctl.stall_fw;
  if (!wb_stt.pctl.stalled_flag && ex_reg.dst_f) {
    gpr [ ex_reg.dst_id ] = wb_data;
  } . . . .
```

RISC-V Scalar Core Models From C++ Descriptions

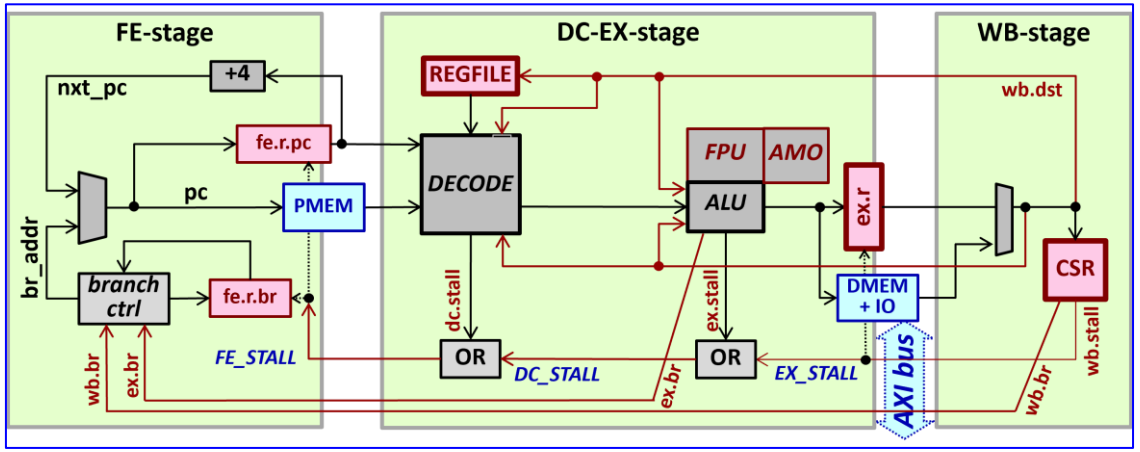


5-stage (Linux-OS enabled, MMU + Caches) : 740MHz @ 28nm

- MMU = TLB + page-walk logic (VIPT)
- I-Cache/I-TLB → AXI-Master port
- D-Cache/D-TLB/IO → AXI-Master port
- ISA : RV32/64-IMAFD (M/F/D : optional)
- Priviledge modes : U/M/HS/VU/VS



VIPT : Virtually-indexed physically tagged

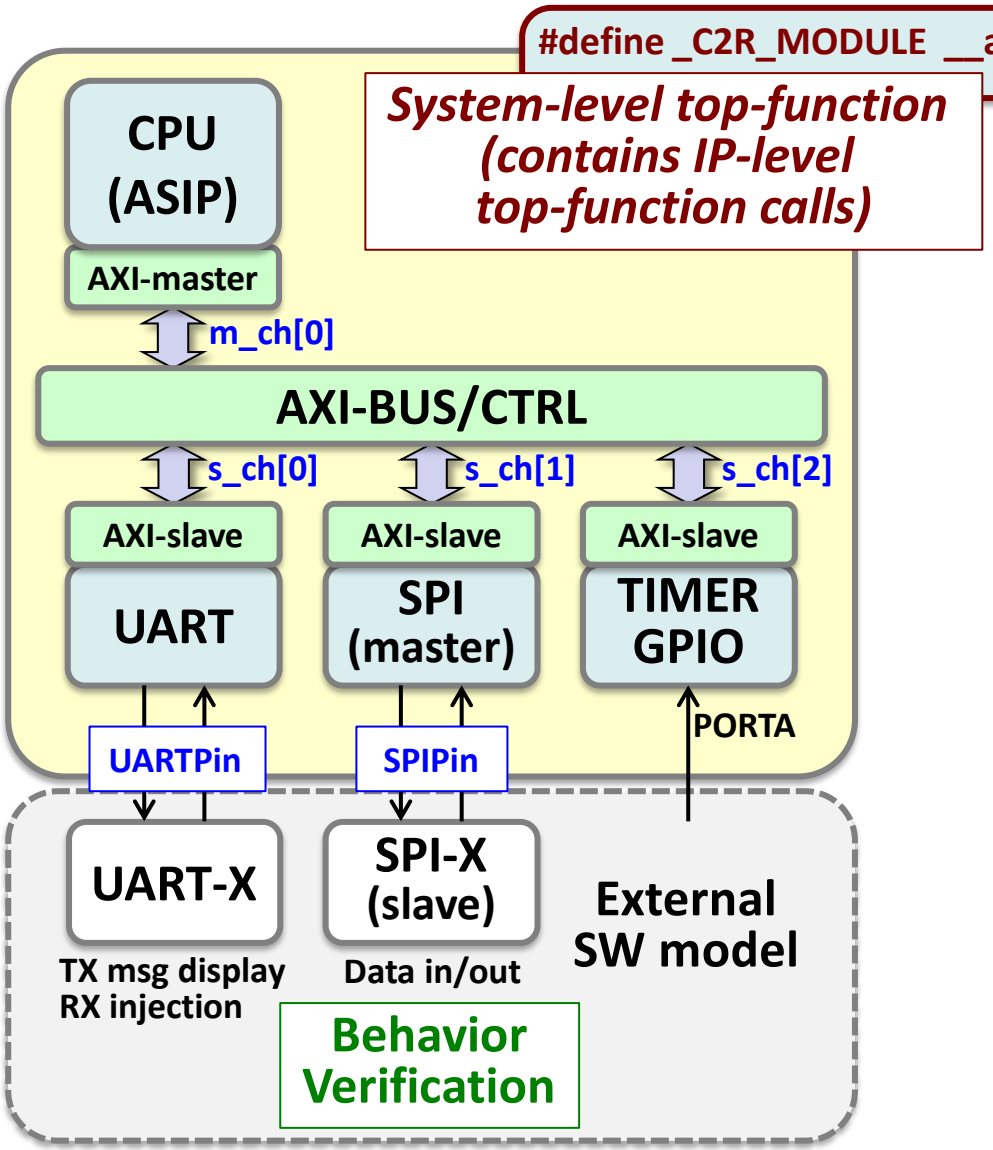


3-stage cache-less : 650MHz @ 28nm (est.)

- For embedded apps: no cache, no MMU
- ISA : RV32-IMAFD (M/F/D : optional)
- Priviledge modes : U/M
- Higher IPC : no branch stalls

All RISC-V scalar cores designed in C++ and translated to RTL by C2RTL

SoC-RTL Model + SoC Simulation Model



System-level top-function (contains IP-level top-function calls)

```
#define _C2R_MODULE __attribute__((C2R_module))
```

single-cycle system behavior

SoC top

```
_C2R_MODULE
int TCTProcAXI (UARTPin *uart, SPIPin *spi,
        UINT32 porta, AXI4L::BUS<1, 3> *axi_bus) {
   axi_uart    (&axi_bus->s_ch[0], uart) ;
   axi_spim    (&axi_bus->s_ch[1], spi) ;
   axi_sysctrl (&axi_bus->s_ch[2], porta) ;
   int val = TCTProc (&axi_bus->m_ch[0]) ;
   axi_bus_ctrl (axi_bus) ;
   return val ;
}
```

auto-generate SoC-level RTL

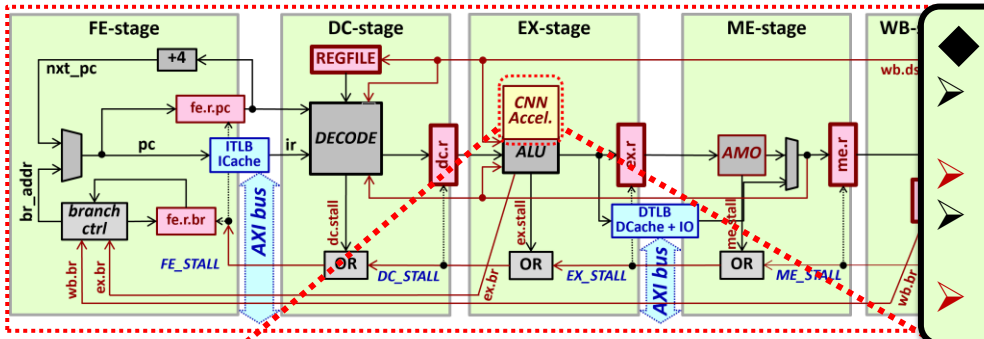
SoC simulator main function

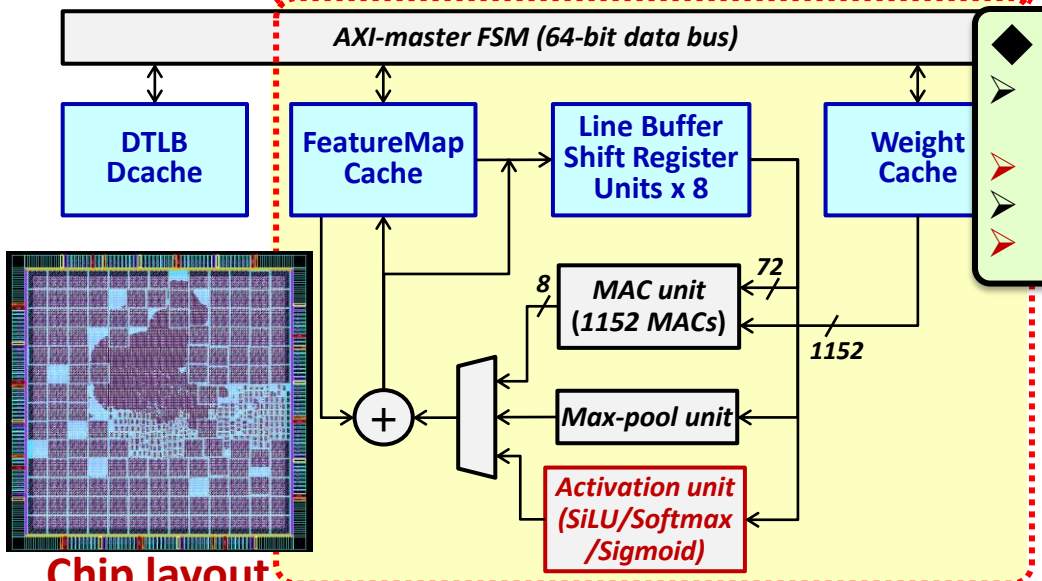
```
int main (int argc, char * argv[]) {
   UINT32 port_a ;
   UARTPin uart ;
   SPIPin spi ;
   AXI4L::BUS<1, 3> axi_bus = { 0
   /// initialization codes here...
   while ( !TCTProcAXI (&uart, &spi, port_a, &axi_bus)) {
      spi_ext_slave (&spi) ;
      uart.rx = uart_ext(uart.tx) ;
   }
}
```

Fully user-defined data structures (not built-in classes)

External SW models

Our RISC-V Embedded AI Accelerator Designs





Chip layout

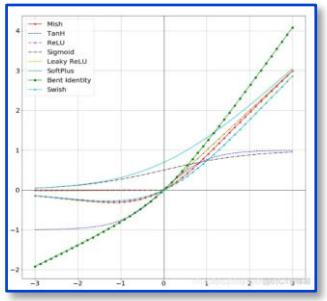
AIV-SoC architecture [2]

◆ **DNN-AS [1] Supported models :** RESNET, VGG.

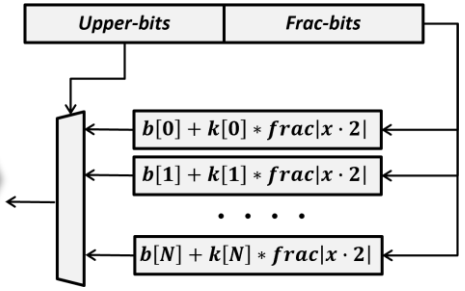
- Line buffer, data core, custom instruction, dynamic fixed point, per-group quantization and general MAC array introduced.
- **1152 MAC units, 0.622 TOPS(peak)/0.137mW @ 540MHZ, 28nm CMOS**
- Memory bandwidths: 256-bit AXI bus, 17.3 GB/s (external), 276.8 GB/s(internal)
- **Energy efficiency: 1893x improvement over GTX1080Ti (RESNET101)**

◆ **AIV-SoC [2] Supported models :** YOLOv8, RESNET, VGG.

- Based on DNN-AS, adding PLF Module, SoftMax unit, matrix-multiply, transposed convolution, YOLOv8 post-processing
- **1152 MAC units, 0.685 TOPS(peak)/202mW @ 595MHZ, 28nm CMOS**
- Memory bandwidths: 64-bit AXI bus, 4.76 GB/s (external), 304.6 GB/s (internal)
- **Energy efficiency: 166x improvement over GTX1080Ti (YOLOv8s)**



Activation, SoftMax



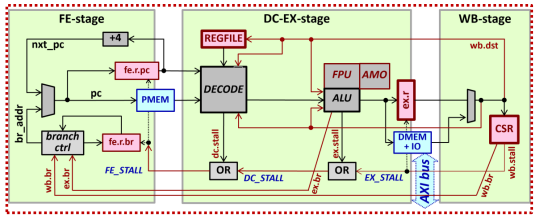
Piecewise Linear Function (PLF)

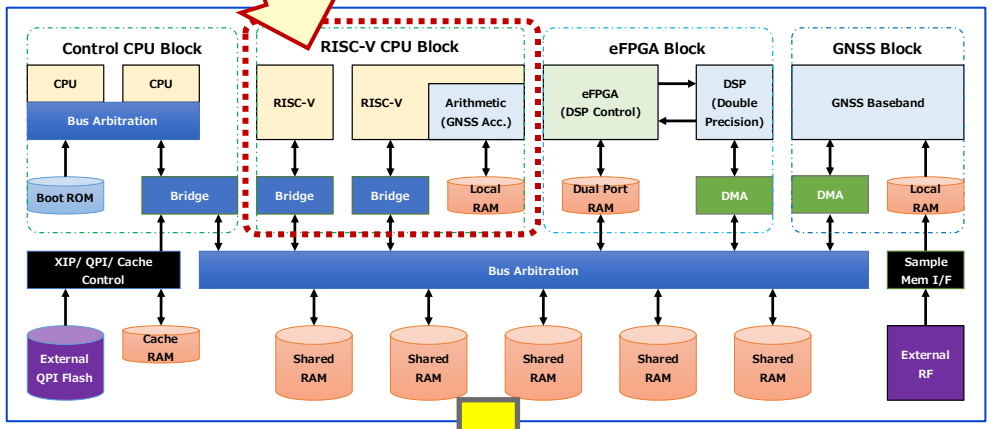
[1] H. Wang, D. Li, T. Isshiki, "A Low-Power Reconfigurable DNN Accelerator for Instruction-Extended RISC-V", IPSJ Trans. SLDM (2024)

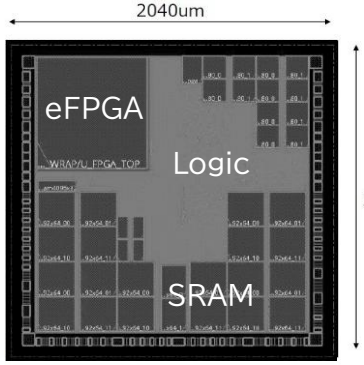
[2] H. Wang, D. Li, T. Isshiki, "Energy-Efficient Implementation of YOLOv8, Instance Segmentation, and Pose Detection on RISC-V SoC", IEEE Access (2024)

Low Power IoT RISC-V SoC (SEIKO EPSON CORP)

RISC-V + GNSS accelerator (positioning calculation on FP64) designed on C2RTL



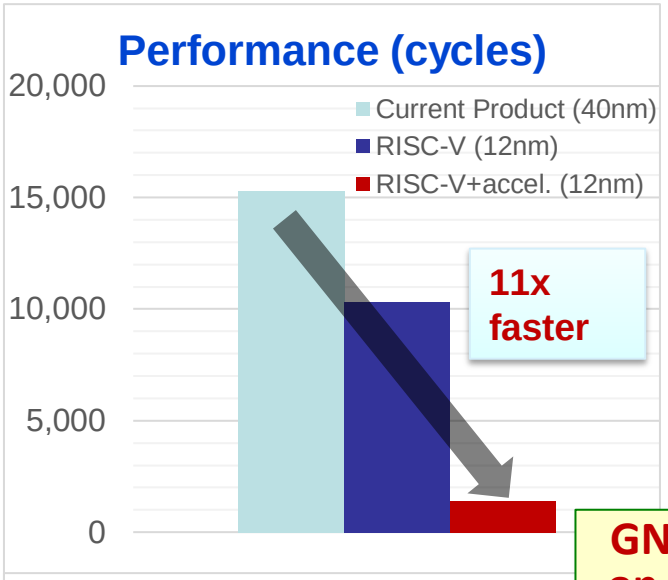




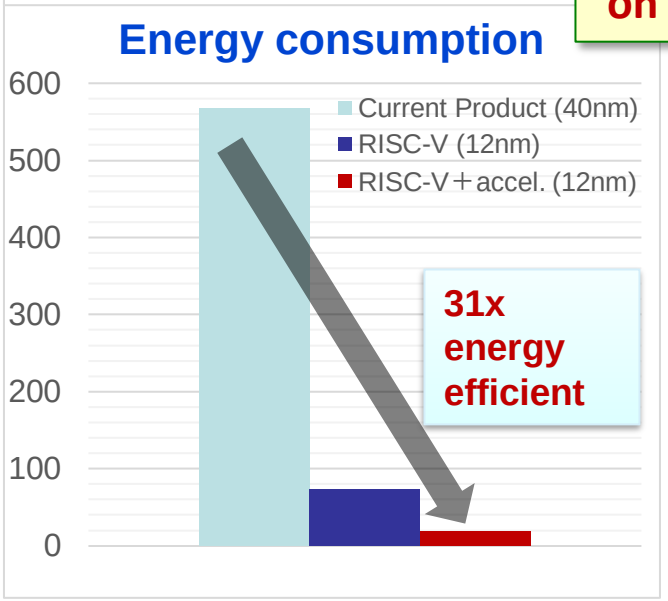
Chip Design (12nm CMOS)

- Chip 4mm²
- RISC-V with GNSS accelerator
- eFPGA (configurable accelerator)
- GNSS baseband

Power simulation (extracted from chip layout)



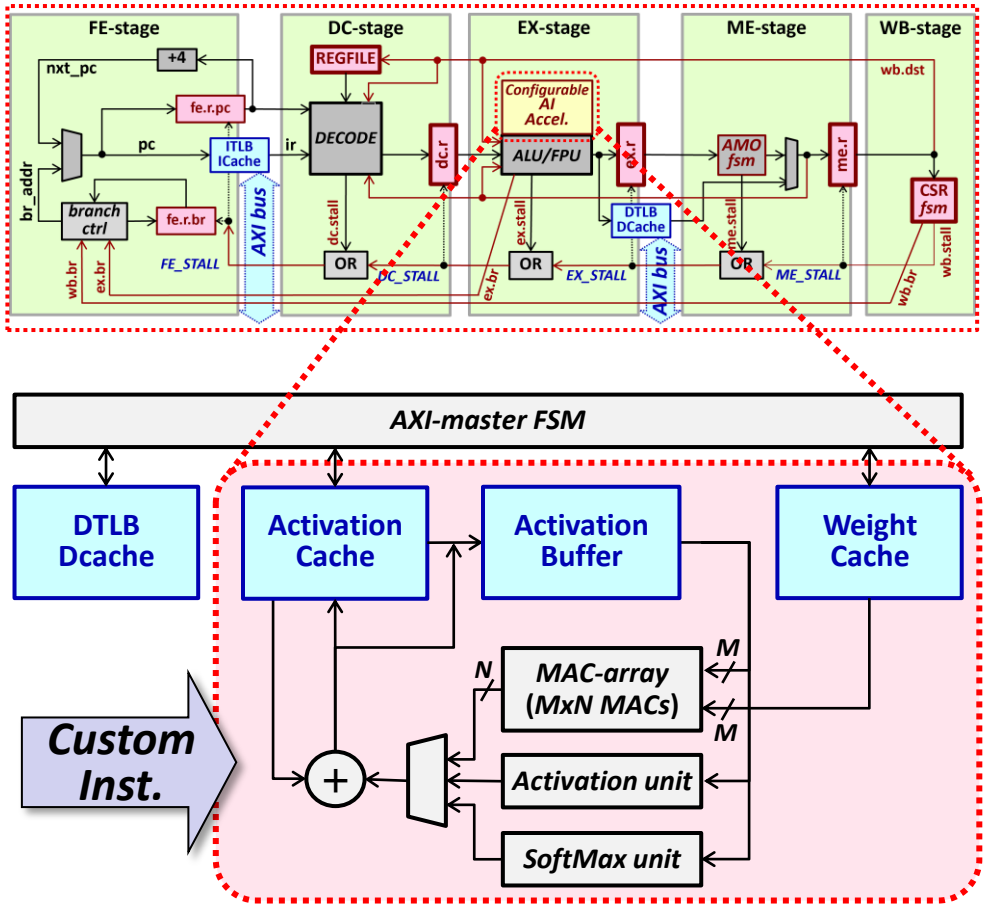
GNSS positioning on FP64 compute



RISC-V AI Accelerator Design Platform (*in progress*)

- **Target AI / analytics algorithms**
  - AI : Transformers (LLM, SLM), CNN, etc.
  - Analytics : FFT, wavelet, PCA, etc.
- **Accelerator design on C++/C2RTL → auto-generate RTL model & simulation models**
  - MAC-arrays
  - Non-linear function units (SoftMax, activation)
  - High bandwidth dedicated cache system
  - High bandwidth wide AXI-bus
  - SW-controlled AI accelerator (custom inst.)
- **Enable AI algorithm designers to generate optimized hardware on PPA* design space**

PPA : Power Performance Area*



CNN/Transformer mode

LLM (Llama3) Training on 16K H100 GPUs

Nvidia H100 spec [1]	
clock freq	1.8GHz
# tensor cores	528
tensor FP16/BF16 FLOPS	989 TFLOPS
tensor FP8 FLOPS	1979 TFLOPS
memory bandwidth	3.35 TB/s
memory size	80GB
L2 cache size	50MB
TDP	700W
TSMC process	N4

tensor BF16 FLOPS usage : 38% ~ 43%

GPUs	TP	CP	PP	DP	Seq. Len.	Batch size/DP	Tokens/Batch	TFLOPs/GPU	BF16 MFU
8,192	8	1	16	64	8,192	32	16M	430	43%
16,384	8	1	16	128	8,192	16	16M	400	41%
16,384	8	16	16	8	131,072	16	16M	380	38%

Power (W/GPU)	700W
Power (W@16K GPUs)	11.4 MW

Llama3.1 Training Cost [3]	8B	70B	405B
Training Time (GPU hours)	1.46M	7.0M	30.84M
Training Energy consumption	1.0GWh	4.9GWh	21.6GWh
Training Duration @ 16K GPUs	3.7 days	17.8 days	78.4 days

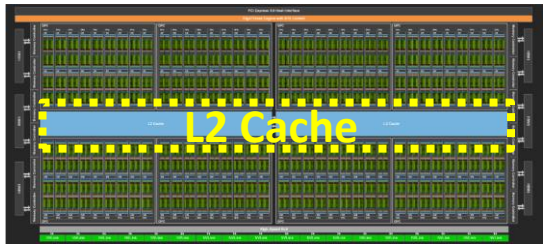
		8B	70B	405B
dim	D	4096	8192	16384
n_layers		32	80	126
n_heads	H_Q	32	64	128
n_kv_heads	H_{KV}	8	8	8
head_dim	D_H	128	128	128
max_token_length 3:8K, 3.1:128K	N	8192 128000	8192 128000	128000
mlp_intermediate_dim	M	14336	28672	53248

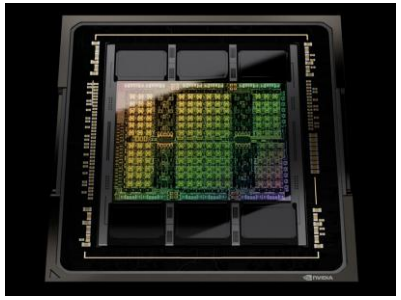
Category	Benchmark	Llama 3 8B	Gemma 2 9B	Mistral 7B	Llama 3 70B	Mixtral 8x22B	GPT 3.5 Turbo	Llama 3 405B	Nemotron 4 340B	GPT-4 (0125)	GPT-4o	Claude 3.5 Sonnet
General	MMLU (5-shot)	69.4	**72.3**	61.1	**83.6**	76.9	70.7	87.3	82.6	85.1	89.1	**89.9**
	MMLU (0-shot, CoT)	**73.0**	72.3△	60.5	**86.0**	79.9	69.8	88.6	78.7◁	85.4	**88.7**	88.3
	MMLU-Pro (5-shot, CoT)	**48.3**	–	36.9	**66.4**	56.3	49.2	73.3	62.7	64.8	74.0	**77.0**
	IFEval	**80.4**	73.6	57.6	**87.5**	72.7	69.9	**88.6**	85.1	84.3	85.6	88.0
Code	HumanEval (0-shot)	**72.6**	54.3	40.2	**80.5**	75.6	68.0	89.0	73.2	86.6	90.2	**92.0**
	MBPP EvalPlus (0-shot)	**72.8**	71.7	49.5	**86.0**	78.6	82.0	88.6	72.8	83.6	87.8	**90.5**
Math	GSM8K (8-shot, CoT)	**84.5**	76.7	53.2	**95.1**	88.2	81.6	**96.8**	92.3◇	94.2	96.1	96.4◇
	MATH (0-shot, CoT)	**51.9**	44.3	13.0	**68.0**	54.1	43.1	73.8	41.1	64.5	**76.6**	71.1
Reasoning	ARC Challenge (0-shot)	83.4	**87.6**	74.2	**94.8**	88.7	83.7	**96.9**	94.6	96.4	96.7	96.7
	GPQA (0-shot, CoT)	32.8	–	28.8	**46.7**	33.3	30.8	51.1	–	41.4	53.6	**59.4**
Tool use	BFCL	**76.1**	–	60.4	84.8	–	**85.9**	88.5	86.5	88.3	80.5	**90.2**
	Nexus	**38.5**	30.0	24.7	**56.7**	48.5	37.2	**58.7**	–	50.3	56.1	45.7
Long context	ZeroSCROLLS/QuALITY	81.0	–	–	90.5	–	–	**95.2**	–	**95.2**	90.5	90.5
	InfiniteBench/En.MC	65.1	–	–	78.2	–	–	**83.4**	–	72.1	82.5	–
	NIH/Multi-needle	98.8	–	–	97.5	–	–	98.1	–	**100.0**	**100.0**	90.8
Multilingual	MGSM (0-shot, CoT)	**68.9**	53.2	29.9	**86.9**	71.1	51.4	**91.6**	–	85.9	90.5	**91.6**

[1] https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c?ncid=no-ncid
[2] https://ai.meta.com/research/publications/the-llama-3-herd-of-models/
[3] https://github.com/meta-llama/llama-models/blob/main/models/llama3_1/MODEL_CARD.md

GPU HW Resources Used in LLM Training/Inference

Nvidia H100 (2022) spec [1]	
clock freq	1.8GHz
# tensor cores	528 (512 FP16 MACs per core)
tensor FP16/BF16 FLOPS	989 TFLOPS
tensor FP8 FLOPS	1979 TFLOPS
memory bandwidth	3.35 TB/s
memory size (HBM3)	80GB
# SM (Steaming Multiprocessor)	132 (4 tensor cores/SM)
Register file size	33.8MB (256KB x 132)
Shared Memory size	29.6MB (224KB x 132) (L1 cache : 256KB)
L2 cache size	50MB
Inter-GPU bandwidth	25 GB/s x 18 channels x 2 directions (900 GB/s)
TDP	700W
TSMC process	N4
Transistors	80 Billion
Die size	814 mm^2 (28.5mm sq.)





LLM Training/Inference H100 resource

528 Tensor Cores
(270K FP16 MAC units)
(512 FP16 MACs/core)

10K tr/MAC (est.)

2.7B Tr

Shared Memory
29.6MB (224KB x 132)

6 Tr/bit SRAM

1.4B Tr

L2 Cache
50MB

2.4B Tr

Total 6.4B Tr (est.)

H100 : 80B Tr

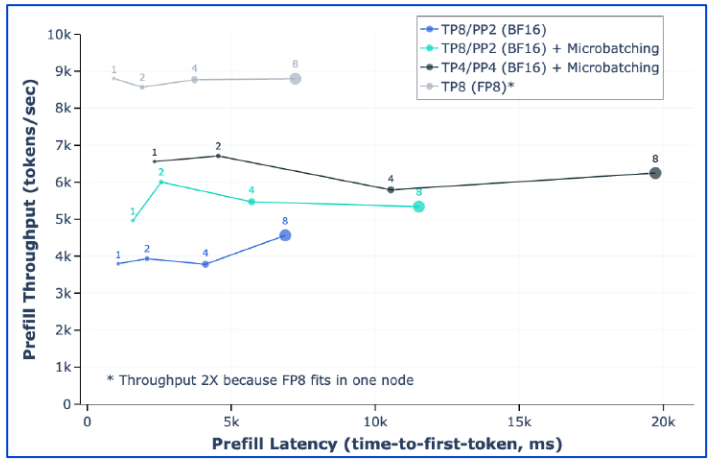
HBM3 Memory
80GB, 3.34 TB/s

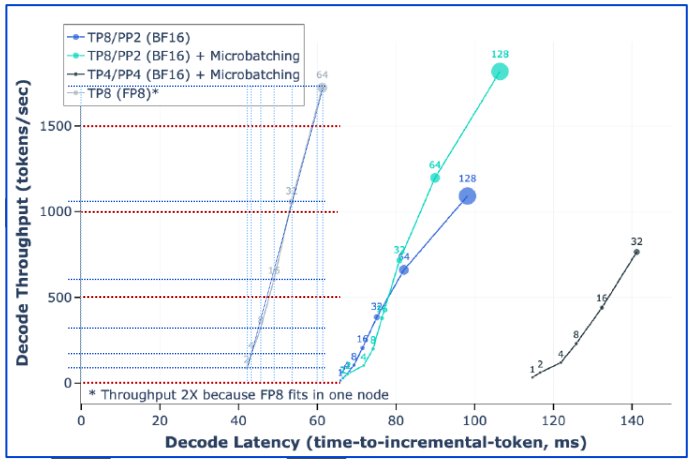
[1] https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c?ncid=no-ncid

GPU HW Resource Under-utilized in LLM Inference ?

Llama3.1 405B inference system [2]	
# H100 GPUs	8 x H100 GPUs
total tensor FP8 FLOPS	15,832 TFLOPS
total memory bandwidth	26.8 TB/s
total power	5600W
precision	FP8
tensor parallelism (TP)	8
# input tokens	4096
# output tokens	256

		8B	70B	405B
dim	D	4096	8192	16384
n_layers		32	80	126
n_heads	H_Q	32	64	128
n_kv_heads	H_{KV}	8	8	8
head_dim	D_H	128	128	128
max_token_length 3:8K, 3.1:128K	N	8192 128000	8192 128000	128000
mlp_intermediate_dim	M	14336	28672	53248





prefill latency = 0.93s (4096 input tokens on 8 H100s)

prefill FLOPS = 3638 TFLOPS

prefill FLOPS usage = 22.9%

prefill memory bandwidth usage = 7.5%

decode thoughput : 13.5 Tokens/s/user (64 batches)

decode FLOPS = 56.9 TFLOPS

decode FLOPS usage = 4.85%

decode memory bandwidth usage = 31.6%

[2] https://ai.meta.com/research/publications/the-llama-3-herd-of-models/

LLM (Llama-3) Inference Accelerator PPA Analysis

KV activation (Prefill)

$$
\begin{aligned}
&\text{for } (i = 0; i < N; i++)\{\\
&\quad \text{for } (h = 0; h < H_{KV}; h++)\{\\
&\qquad \text{for } (k = 0; k < D_H; k++)\{\\
&\qquad\quad K_{i,k,h} = \textstyle\sum_{d=0}^{D-1} X_{i,d} \cdot [W_K]_{h,k,d}\\
&\qquad\quad V_{i,k,h} = \textstyle\sum_{d=0}^{D-1} X_{i,d} \cdot [W_V]_{h,k,d}\\
&\qquad \}\\
&\quad \}\\
&\}
\end{aligned}
$$

Attention score (Prefill)

$$
\begin{aligned}
&\text{for } (i = 0; i < N; i++)\{\\
&\quad \text{for } (h = 0; h < H_Q; h++)\{\\
&\qquad \text{for } (k = 0; k < D_H; k++)\\
&\qquad\quad Q_{i,k,h} = \textstyle\sum_{d=0}^{D-1} X_{i,d} \cdot [W_Q]_{h,k,d}\\
&\qquad \text{for } (j = 0; j < N; j++)\{\\
&\qquad\quad S_{i,j,h} = \frac{1}{\sqrt{D_H}} \textstyle\sum_{k=0}^{D_H-1} Q_{i,k,h} \cdot K_{j,k,h/R}\\
&\qquad\quad \alpha_{i,j,h} = \frac{\exp(S_{i,j,h})}{\sum_{i=0}^{N-1} \exp(S_{i,i,h})}\\
&\qquad \}\\
&\qquad \text{for } (k = 0; k < D_H; k++)\\
&\qquad\quad O_{i,k,h} = \textstyle\sum_{j=0}^{N-1} \alpha_{i,j,h} \cdot V_{j,k,h/R}\\
&\quad \}\\
&\quad \text{for } (d = 0; d < D; d++)\{\\
&\qquad Y'_{i,d} = \textstyle\sum_{h=0}^{H_Q-1} \sum_{k=0}^{D_H-1} O_{i,k,h} \cdot [W_O]_{h,k,d}\\
&\qquad Y_{i.d} = LayerNorm(Y'_{i,d} + X_{i,d})\\
&\quad \}\\
&\}
\end{aligned}
$$

N : token length
D : model dimension
M : FFN dimension
D_H : head dimension (fixed to 128)
H_Q : # of attention heads ($D = D_H * H_Q$)
H_{KV} : # of KV-heads ($R = H_Q / H_{KV}$)

		8B	70B	405B
$[W_K]_{h,k,d}, [W_V]_{h,k,d}$	$H_{KV} \times D_H \times D$	4M	8M	16M
$[W_Q]_{h,k,d}, [W_O]_{h,k,d}$	$H_Q \times D_H \times D$	16M	32M	64M
$[W_1]_{m,d}, [W_2]_{m,d}, [W_3]_{m,d}$	$M \times D$	59M	235M	872M

Feed-Forward (Prefill)

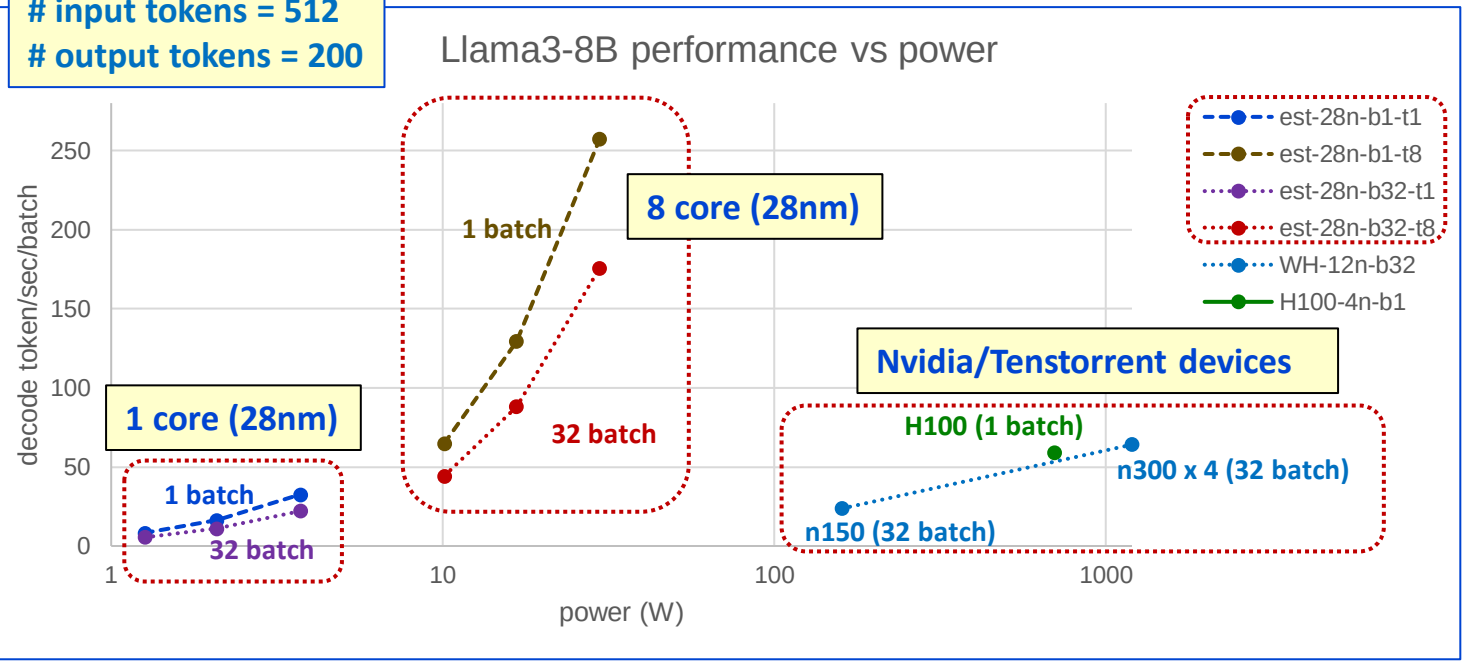
$$
\begin{aligned}
&\text{for } (i = 0; i < N; i++)\{\\
&\quad \text{for } (m = 0; m < M; m++)\{\\
&\qquad Z1'_{i,m} = \textstyle\sum_{d=0}^{D-1} Y_{i,d} \cdot [W_1]_{d,m}\\
&\qquad Z1_{i,m} = SILU(Z1'_{i,m})\\
&\qquad Z2_{i,m} = \textstyle\sum_{d=0}^{D-1} Y_{i,d} \cdot [W_3]_{d,m}\\
&\qquad Z3_{i,m} = Z1_{i,m} \cdot Z2_{i,m}\\
&\quad \}\\
&\quad \text{for } (d = 0; d < D; d++)\{\\
&\qquad Z'_{i,d} = \textstyle\sum_{m=0}^{M-1} Z3_{i,m} \cdot [W_2]_{d,m}\\
&\qquad Z_{i,d} = LayerNorm(Z'_{i,d} + Y_{i,d})\\
&\quad \}\\
&\}
\end{aligned}
$$

Prefill-stage calculation

- **Llama-3 model : auto-regressive transformer**
  - On-chip memory size controlled by weight-matrix/vector multiply loop unrolling
- **PPA Analysis assumptions**
  - ***KV-cache*** stored in on-chip SRAM
  - Layer activations ($X_{[N][D]}$, $Y'_{[N][D]}$, $Y_{[N][D]}$, $Z'_{[N][D]}$, $Z_{[N][D]}$) stored in off-chip DDR during ***prefill-stage***
  - Layer activations ($X_{0,[D]}$, $Y'_{0,[D]}$, $Y_{0,[D]}$, $Z'_{0,[D]}$, $Z_{0,[D]}$) stored in on-chip SRAM during ***decode-stage***
- **HW Accelerator parameters for load analysis:**
  - Weight partitioning strategy
  - Clock frequency
  - # MAC units
  - Memory transfer bandwidth (# words/clk)
  - Tensor parallelism (multi-core)
  - Inter-core transfer bandwidth (# words/clk) (used in reduce/scatter when tensor parallelism enabled)
- **HW cycle estimation assumption**
  - Tensor cycles = (# MAC ops) / (# MAC units)
  - High-order function (SoftMax, SILU) excluded from estimation
  - Total cycles = (Tensor cycles) + (Memory Transfer cycles) + (Inter-core transfer cycles) → no overlapping of tensor operation cycles and data transfer cycles assumed

LLM (Llama3-8B/FP8) Inference Accelerator PPA Analysis

 # input tokens = 512
 # output tokens = 200



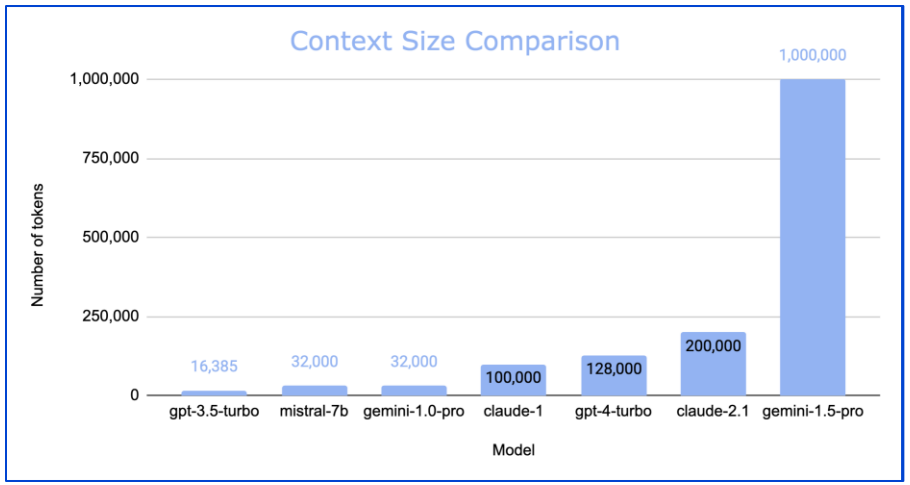
- **HW cycle estimation assumption**
  - Tensor cycles = (# MAC ops) / (# MAC units)
  - ***High-order function (SoftMax, SILU) excluded in estimation***
  - Total cycles = (Tensor cycles) + (Memory Transfer cycles) + (Inter-core transfer cycles)
- **Power/Area estimation assumption (28nm CMOS)**
  - **Based on our AIV-SoC (28nm CMOS)**
  - Cell area/power based on # MAC units ratio
  - RAM area/power based on RAM size ratio
  - ***DDR I/F excluded from power/area estimation***
- **Power comparison on comparable performance**
  - **42.9x** power reduction against **n150**
  - **69.1x** power reduction against **H100**
  - **71.9x** power reduction against **n300x4 (QuietBox)**

# cores (0.9GHz)	# MAC units per core	# DDR4 channels per core (1.8GHz)	prefill latency (sec)	decode throughput (1 user) (token/s/user)	decode throughput (32 users) (token/s/user)	area (mm^2) (28nm)	Total power (W)	Total SRAM size (MB)
1	8192	2	0.722	8.128	5.529	16.076	1.266	5.636
1	16384	4	0.361	16.256	11.059	23.194	2.085	5.636
1	32768	8	0.181	32.511	**22.117**	37.43	3.723	5.636
8	8192	2	0.109	**64.869**	44.163		10.128	45.088
8	16384	4	0.064	129.433	**88.185**		16.68	45.088
8	32768	8	0.041	257.651	175.806		29.784	45.088

n150, n300 : Tenstorrent Wirmhole (12nm process)
H100 : Nvidia (N4 process)

device	Total # MAC units	decode throughput (1 user) (token/s/user)	decode throughput (32 users) (token/s/user)	Total power (W)	Total SRAM size (MB)
n150	131000	NA	**23.8**	160	108
H100	540672	**59.16**	NA	700	50
n300 x 4	1864000	NA	**64.3**	1200	768

Current LLM Trends : Large Context Size

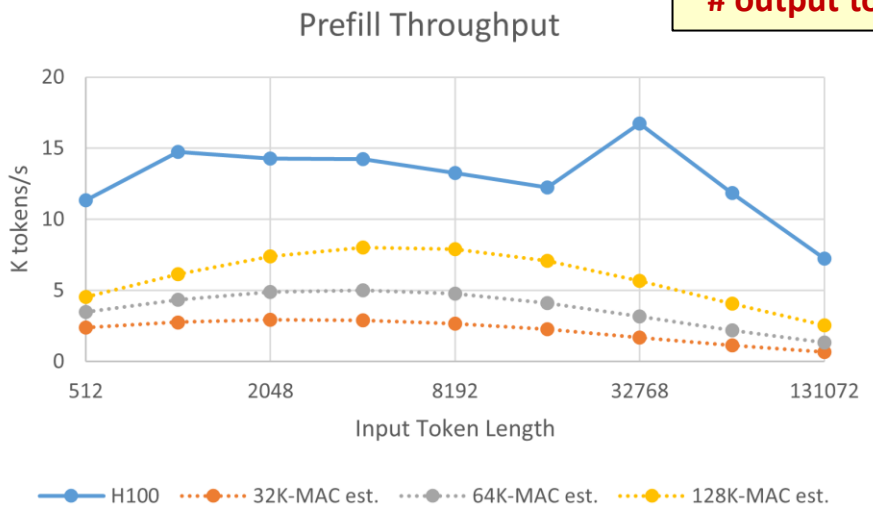


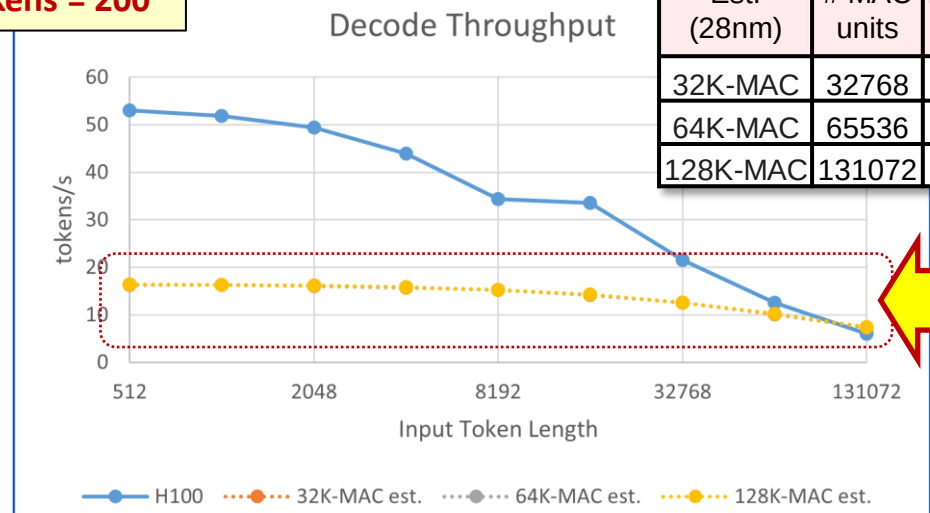
- **Rapid growth in LLM context size**
  - **8K** tokens (Llama-3) → **128K** tokens (Llama-3.1) → **1M~10M** tokens (Llama-4)
- **Large context size degrades performance for LLM inference → need further studies in SW & HW implementations**
  - ***Tensor library not well-tuned for large context size?***
  - ***HW architecture issues for large context size? (on-chip RAM size?)***

Inference HW platform : H100 (540K FP8-MACs, 700W, 3.35 TB/s)
SW library : Huggingface Transformer (Llama-3.1-8B)

https://huggingface.co/docs/transformers/index

 # output tokens = 200





Est. (28nm)	# MAC units	DDR4 chan.	Memory bandwidth	area (mm^2)	power (W)	SRAM (MB)
32K-MAC	32768	4	0.11 TB/s	85.134	6.27	37.75
64K-MAC	65536	4	0.11 TB/s	116.77	9.52	37.75
128K-MAC	131072	4	0.11 TB/s	173.89	16.01	37.75

Decode throughput bounded by memory bandwidth

MAC throughput has minimal impact on decode throughput

Summary

- **RISC-V system design platform (enabled by C2RTL Framework):**
  - **RISC-V core: 32/64-bit IMAFD ISA-subset configurable from a single source**
  - **HW design : RISC-V core + instruction extension + HW accelerator**
  - **SW design : (LLVM-based) compiler (ext. support), applications, middleware, OS**
  - **HW/SW co-verification : debugger probes (reg-file, memory) extending to RTL signals (pipeline registers, feedback wires)**
- **RISC-V Embedded AI Accelerator Design Platform:**
  - **Target AI/analytics algorithms**
    - AI : Transformers (LLM, SLM), CNN, etc.
    - Analytics : FFT, wavelet, PCA, etc.
  - **Accelerator design on C++/C2RTL → auto-generate RTL model & simulation models**
    - MAC-arrays, non-linear function units (SoftMax, activation)
    - High bandwidth dedicated cache system, wide AXI-bus
    - SW-controlled AI accelerator (instr. extension)
  - **Enable AI algorithm designers to generate optimized hardware on PPA* design space**
  - ***Large opportunities for optimization in LLM inference implementation!***