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

Tsuyoshi Isshiki

Dept. of Information and Communications Engineering
Institute of Science Tokyo

June 23th, 2026

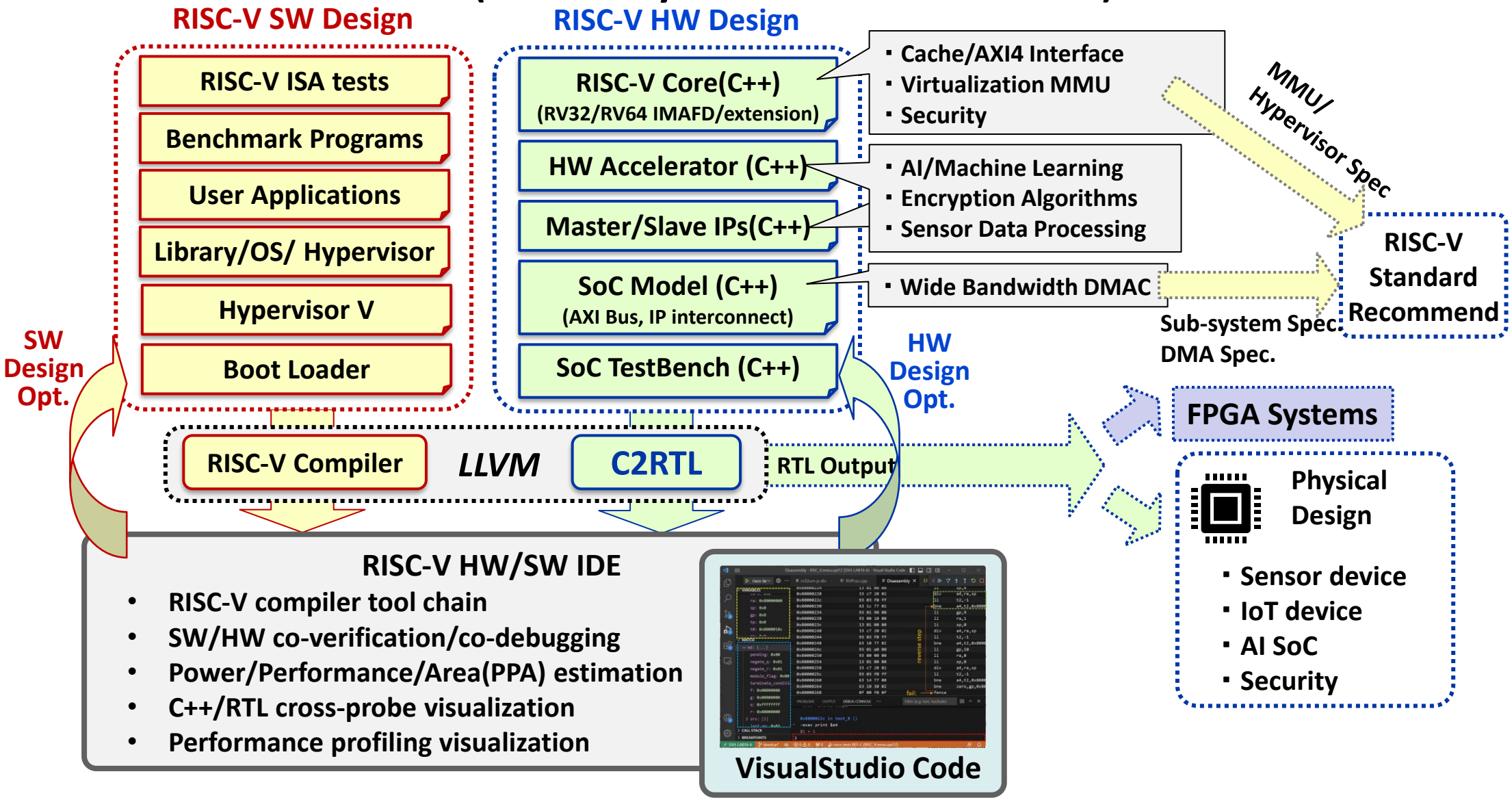
RISC-V Based AI SoCs

- **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 consumer/enterprise/data center/automotive SoC markets**
  - **RISC-V ratifications : vector, scalar cryptography, hypervisor → RVA23 profile**
- **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***

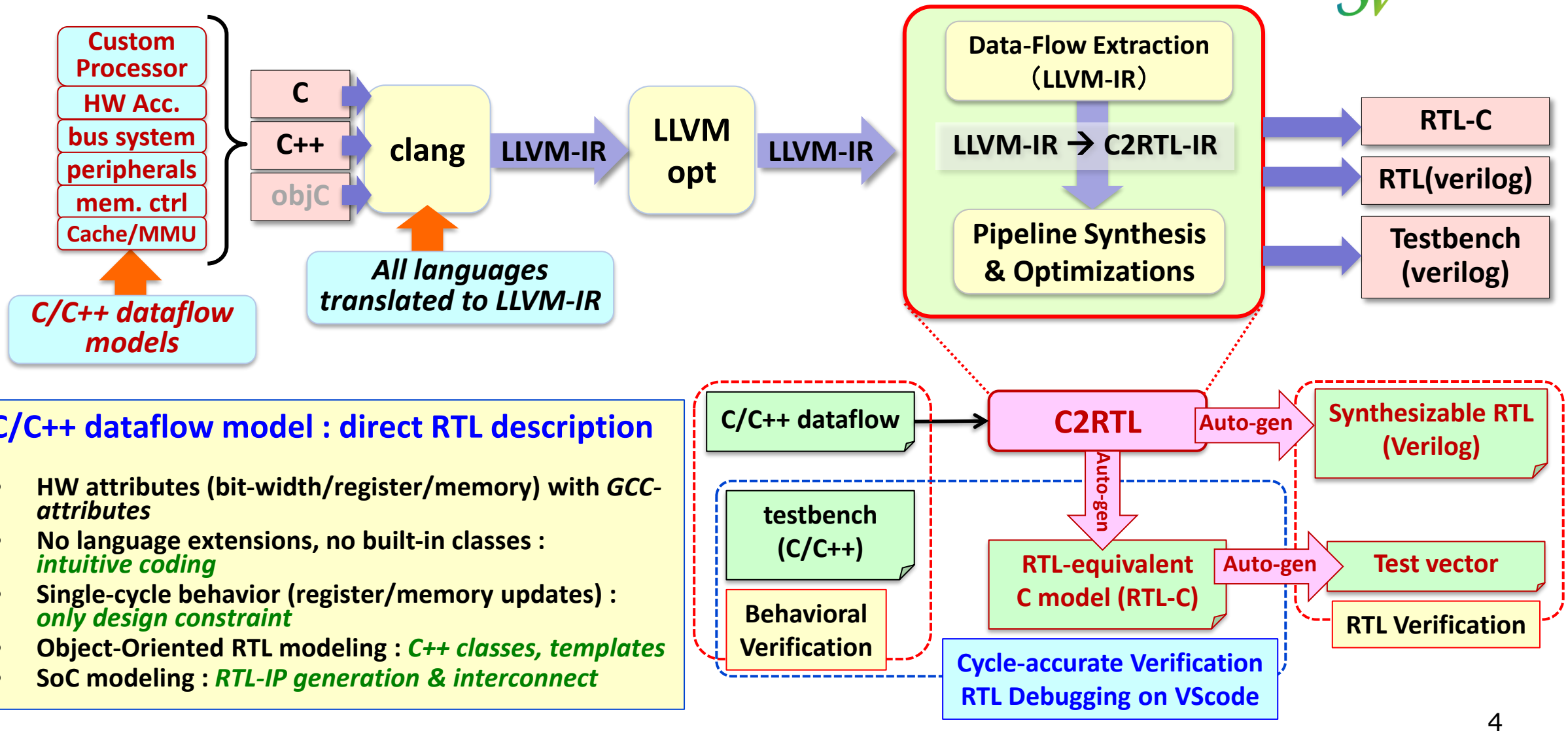
→*Need System-Level SW/HW Design Environment for efficiently building customized RISC-V cores, HW accelerators and SoC architectures*

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

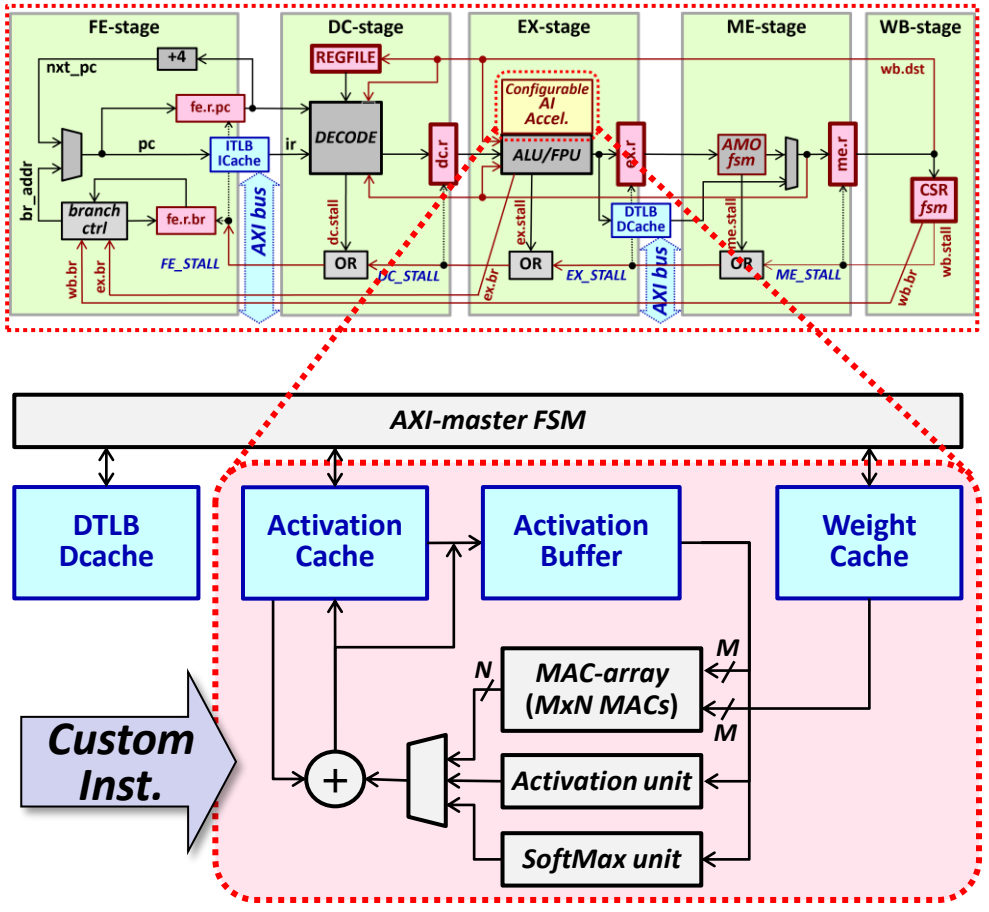
- HW attributes (bit-width/register/memory) with *GCC-attributes*
- No language extensions, no built-in classes : *intuitive coding*
- Single-cycle behavior (register/memory updates) : *only design constraint*
- Object-Oriented RTL modeling : *C++ classes, templates*
- SoC modeling : *RTL-IP generation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

RISC-V AI Accelerator Design Platform

(JST-Research and Development Program for Next-generation Edge AI Semiconductors : 2025 – 2030)

- **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

RISC-V Transformer Accelerator Baseline Architecture

- **Data format : MX-INT8 (32 exponent groups, 8-bit integer quantized data) widely supported in latest GPU/NPUs**
  - Efficient integer-based MAT-MUL compute
  - Output vector renormalization to recover MX-INT8 format
- **SW implementation : tensor-level custom instructions**
  - **LoadVector :** weight matrix, input activation vector
  - **StoreVector :** logits output vectors
  - **Compute :** MAT-MUL, RMSNorm, RoPE, Softmax, SWIGLU
  - **Move :** data transfer between on-chip memories
  - **Config :** datapath setup (src/dst memory pointers, FSM counters)
- **MAC units (currently supports 2 ~ 32 units → final target up to 128 ~ 256 units)**
  - **Weight/KV-cache memory** on each MAC unit
  - **32 x 2 INT8 MACs + normalized accumulator / unit → 2048 MACs/cycle @ 32 units**
  - Element-wise **RMSNorm/RoPE/Softmax/SWIGLU** functions → 1 output / unit
- **High bandwidth on-chip/off-chip memory transfers**
  - 64-byte (512-bits) AXI-bus → **64GB/s @ 1GHz clock**
  - 64-byte wide on-chip memory / MAC unit → **2TB/s, 4TOP/s @ 32 units**

MX-INT8 for Efficient Integer MAT-MUL Compute

➢ **MX-INT8 : 32 exponent groups, 8-bit integer quantized data**

FP vector : $x_{[N]} = [x_0 \quad x_1 \quad \cdots \quad x_{N-1}]$

$= \left[2^{x_0^{\langle E \rangle}} \cdot x_0^{\langle F \rangle} \quad 2^{x_1^{\langle E \rangle}} \cdot x_1^{\langle F \rangle} \quad \cdots \quad 2^{x_{N-1}^{\langle E \rangle}} \cdot x_{N-1}^{\langle F \rangle}\right]$

Individual exponents on all vector elements

MX-INT8 vector : $x_{\langle N \rangle} = [x_0 \quad x_1 \quad \cdots \quad x_{N-1}]$

$= 2^{x^{\langle E \rangle}} [x_0^{\langle F \rangle} \quad x_1^{\langle F \rangle} \quad \cdots \quad x_{N-1}^{\langle F \rangle}] \, (N = 32)$

Single exponent shared by 32 vector elements

FP inner-product : $p = x_{[N]} \cdot y_{[N]}$

$= \sum_{i=0}^{N-1} 2^{x_i^{\langle E \rangle} + x_i^{\langle E \rangle}} \cdot x_i^{\langle F \rangle} \cdot y_i^{\langle F \rangle}$

Need normalization during accumulation

MX-INT8 inner-product : $p = x_{\langle N \rangle} \cdot y_{\langle N \rangle}$

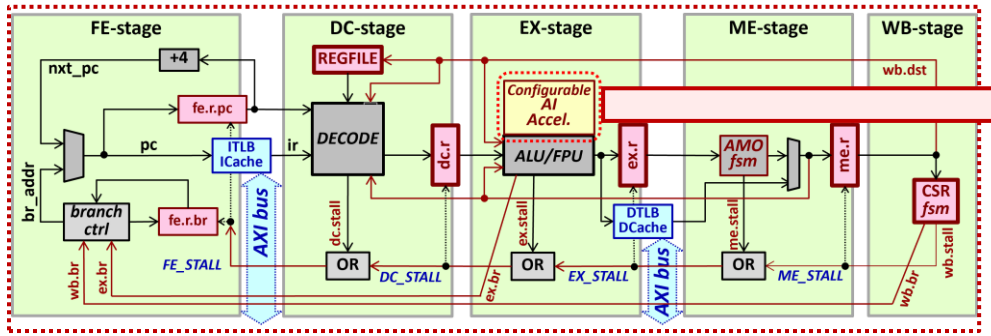
$= 2^{x^{\langle E \rangle} + y^{\langle E \rangle}} \cdot \sum_{i=0}^{N-1} x_i^{\langle F \rangle} \cdot y_i^{\langle F \rangle}$

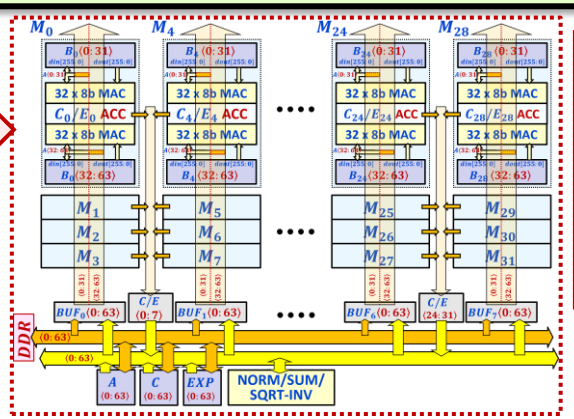
Integer inner-product

Matrix-vector product requires renormalization at each 32 element-group to recover MX-INT8 format

RISC-V Transformer Accelerator Design Flow

- **SW modeling / algorithm optimization : started from "llama2.c" open-source**
  - **MX-INT8 implementation :** exp-grouping strategies, renormalization scheme, special function implementation (Softmax, SWIGLU)
  - → **Build golden SW model for HW design verification**
- **RISC-V Transformer Accelerator HW design verification on C++/C2RTL :**
  - **Tightly-coupled accelerator interface** between RISC-V pipeline micro-architecture
  - **Custom instruction design iteration :** (HW design → SW test) ***fully done on C++ !!!***
  - **Gradual architecture refinement :** high-bandwidth off-chip memory transfer (512-bit wide AXI, burst DMA support with cache bypass), on-chip distributed memory, MAC array, data distribution scheme to MAC array, MX-INT8 normalization architecture, etc.





Transformer HW Accelerator embedded in RISC-V pipeline
(in 5000 lines C++ code!!!)

MX-INT8 Exponent Grouping Scheme on Llama2 Inference

T : Prefill token length, P : token position

$X_{[T],\langle D\rangle}$: input token activation (length : T)

Attention

$X'_{[T],\langle D\rangle} = RMSNorm\left(X_{[T],\langle D\rangle}, [W_{RMS\_A}]_{\langle D\rangle}\right)$

for $(h = 0; h < H_{KV}; h++)$ { // KV-heads

$K_{h,[T],\langle D_H\rangle} = RoPE\left(X'_{[T],\langle D\rangle} \times [W_K]_{h,[D_H],\langle D\rangle}\right)$;

$KC_{h,[P+T],\langle D_H\rangle} = \text{CONCAT}(KC_{h,[P],\langle D_H\rangle}, K_{h,[T],\langle D_H\rangle})$;

$V_{h,\langle T\rangle,[D_H]} = X'_{[T],\langle D\rangle} \times [W_V]_{h,[D_H],\langle D\rangle}$;

$VC_{h,\langle P+T\rangle,[D_H]} = \text{CONCAT}(VC_{h,\langle P\rangle,[D_H]}, V_{h,\langle T\rangle,[D_H]})$;

for $(h' = 0; h' < H_Q/H_{KV}; h'++)$ { // Q-heads

$\hat{h} = h \cdot H_Q/H_{KV} + h'$;

$Q_{[T],\langle D_H\rangle} = RoPE\left(X'_{[T],\langle D\rangle} \times [W_Q]_{\hat{h},[D_H],\langle D\rangle}\right)$;

$S_{[T],\langle P:T\rangle} = \frac{1}{\sqrt{D_H}} Q_{[T],\langle D_H\rangle} \times KC_{h,[P+T],\langle D_H\rangle}$;

$A_{[T],\langle P:T\rangle} = SoftMax(S_{[T],\langle P:T\rangle})$;

$O_{[T],\langle D_H\rangle} = A_{[T],\langle P:T\rangle} \times VC_{h,\langle P+T\rangle,[D_H]}$;

$X_{[T],\langle D\rangle} \mathrel{+}= O_{[T],\langle D_H\rangle} \times [W_O]_{\hat{h},\langle D_H\rangle,[D]}$;

} }

$V_{h,[T],[D_H]} = X'_{[T],\langle D\rangle} \times [W_V]_{h,[D_H],\langle D\rangle}$;

→ translates to :

for $(t = 0; t < T; t++)$

for $(k = 0; k < D_H; k++)$

$V_{h,t,k} = \sum_{d=0}^{D-1} X'_{t,d} \cdot [W_V]_{h,k,d}$

$K_{h,[T],\langle D_H\rangle}$: exp-group on D_H dim

$V_{h,\langle T\rangle,[D_H]}$: exp-group on T dim

V-cache exp-group scheme has large impact on design complexity, speed and model accuracy

Feed-Forward

$X'_{[T],\langle D\rangle} = RMSNorm\left(X_{[T],\langle D\rangle}, [W_{RMS\_F}]_{\langle D\rangle}\right)$;

for $(h = 0; h < H_M; h++)$ { // M-blocks

$H1_{[T],\langle M_H\rangle} = X'_{[T],\langle D\rangle} \times [W_1]_{h,[M_H],\langle D\rangle}$;

$H2_{[T],\langle M_H\rangle} = X'_{[T],\langle D\rangle} \times [W_3]_{h,[M_H],\langle D\rangle}$;

$H3_{[T],\langle M_H\rangle} = Swish(H1_{[T],\langle M_H\rangle}) \odot H2_{[T],\langle M_H\rangle}$;

$X_{[T],\langle D\rangle} \mathrel{+}= H3_{[T],\langle M_H\rangle} \times [W_2]_{h,[D],\langle M_H\rangle}$;

}

$S_{[T],\langle P:T\rangle} = Q_{[T],\langle D_H\rangle} \times KC_{h,[P+T],\langle D_H\rangle}$;

→ translates to :

for $(t = 0; t < T; t++)$

$S_{t,P+t} = \sum_{k=0}^{D_H-1} Q_{t,k} \cdot KC_{h,P+t,k}$

1% PPL accuracy drop vs FP32 on Llama2-7B

On-chip memory footprint :

Llama2-1.1B : 0.946 MB

Llama2-7B : 2.70 MB

Llama3-8B : 3.78 MB

Weight partitioning scheme for small on-chip memory footprint

Tensor-Level Custom Instructions

C Macro call (inline assembler)	ID names : dstID, srcID, compID, moveID, cfgID
Each custom instruction triggers hardwired pipelined FSMs TFM_LoadVector (addr, dstID)	TFM_Weight_Exp, TFM_Logits_Weight_Exp, TFM_VecA_Exp, TFM_VecC_Exp, TFM_LUT_Exp, TFM_Weight_Data, TFM_Logits_Weight_Data, TFM_VecA_Data, TFM_VecC_Data, TFM_LUT_Data, TFM_RMSWeight_Data, TFM_Rope_Data, TFM_Model_Param, TFM_ExtSizeInfo, TFM_MemOffsetInfo, TFM_RMS_Param, TFM_Counter, TFM_ArchInfo
TFM_StoreVector (addr, srcID)	TFM_VecA_Exp, TFM_VecC_Exp, TFM_VecA_Data, TFM_VecC_Data
TFM_Compute (compID)	TFM_COMP_MatMulColNorm, TFM_COMP_MatMulRowNorm, TFM_COMP_MatMulAcc, TFM_COMP_Rope, TFM_COMP_RMS, TFM_COMP_SoftMax, TFM_COMP_Swig
TFM_Move (moveID)	TFM_MV_C2A, TFM_MV_A2B, TFM_MV_VCache, TFM_MV_VCache_RowExp
TFM_Config (cfgID)	TFM_CFG_RMS, TFM_CFG_KCache, TFM_CFG_KQ, TFM_CFG_VCache, TFM_CFG_VCacheTail, TFM_CFG_V, TFM_CFG_Score, TFM_CFG_AttScore, TFM_CFG_WoAcc, TFM_CFG_H1, TFM_CFG_H3, TFM_CFG_Swig, TFM_CFG_W2Acc, TFM_CFG_FinalRMS, TFM_CFG_Logits
TFM_PosTokenLen (pos, tk_len)	pos : token position, tk_len : input token length

For V-cache

Tensor-Level Custom Instructions

Attention Score Calculation

$$Q_{[T],\langle D_H\rangle} = RoPE\left(X'_{[T],\langle D\rangle} \times [W_Q]_{h',[D_H],\langle D\rangle}\right);$$
$$S_{[T],\langle P,T\rangle} = \frac{1}{\sqrt{D_H}} Q_{[T],\langle D_H\rangle} \times K_{h,[P+T],\langle D_H\rangle};$$
$$A_{[T],\langle P,T\rangle} = SoftMax(S_{[T],\langle P,T\rangle});$$
$$O_{[T],\langle D_H\rangle} = A_{[T],\langle P,T\rangle} \times V_{h,\langle P+T\rangle,[D_H]};$$
$$X_{[T],\langle D\rangle} \mathrel{+}= O_{[T],\langle D_H\rangle} \times [W_O]_{h',\langle D_H\rangle,[D]};$$

```
TFM_Config (TFM_CFG_KQ); /////// Q projection ////////////////////
TFM_Load (w0[3], TFM_Weight);                 /// LOAD (Wq);
TFM_Compute (TFM_COMP_MatMulColNorm);  /// Q ← X' * Wq;
TFM_Compute (TFM_COMP_Rope);                  /// Q ← Rope(Q);
TFM_Config (TFM_CFG_Score);  //// SoftMax score ////////////////
TFM_Compute (TFM_COMP_MatMulColNorm);  /// S ← Q * K;
TFM_Compute (TFM_COMP_SoftMax);               /// A ← SoftMax(S);
TFM_Config (TFM_CFG_AttScore);  /// att_score ////////////////////
TFM_Compute (TFM_COMP_MatMulColNorm);  /// O ← A * V;
TFM_Move (TFM_MV_C2A);     /// relocate O : Cmem → Amem
TFM_Config (TFM_CFG_AttAcc);  ////  att_output ////////////////////
TFM_Load (w0[4], TFM_Weight);                 /// LOAD (Wo);
TFM_Compute (TFM_COMP_MatMulAcc);         /// X ← X + O * Wo ;
```

```
#define TFM_Load(qv, dtype) TFM_LoadVector(qv.e, dtype##_Exp); TFM_LoadVector(qv.q, dtype##_Data)
```

```
TFM_Load (w0[3], TFM_Weight);
```

```
TFM_LoadVector (w0[3].e, TFM_Weight_Exp);
TFM_LoadVector (w0[3].q, TFM_Weight_Data);
```

Tensor-Level Custom Instructions

Feed-Forward

$$\text{for } (h = h_0; h < H_M; h++) \{$$
$$H1_{[T],\langle M_H\rangle} = X'_{[T],\langle D\rangle} \times [W_1]_{h,\langle D\rangle,[M_H]} \;;$$
$$H2_{[T],\langle M_H\rangle} = X'_{[T],\langle D\rangle} \times [W_3]_{h,\langle D\rangle,[M_H]} \;;$$
$$H3_{[T],\langle M_H\rangle} = Swish\left(H1_{[T],\langle M_H\rangle}\right) \odot H2_{[T],\langle M_H\rangle} \;;$$
$$X_{[T],\langle D\rangle} \mathrel{+}= H3_{[T],\langle M_H\rangle} \times [W_2]_{h,[D]\langle M_H\rangle} \;;$$
$$\}$$

```
for (int h = 0; h < model_param.num_hblks; h ++) {
  TFM_Config (TFM_CFG_H1); /////// H1 projection ////////////////////
  TFM_Load (w0[6], TFM_Weight);                  /// LOAD (W1);
  TFM_Compute (TFM_COMP_MatMulColNorm);  /// H1 ← X' x W1;
  TFM_Config (TFM_CFG_H3); /////// H2 projection ////////////////////
  TFM_Load (w0[8], TFM_Weight);                  /// LOAD (W3);
  TFM_Compute (TFM_COMP_MatMulColNorm);  /// H2 ← X' x W3;
  TFM_Config (TFM_CFG_Swig); /////// SWIGLU ////////////////////
  TFM_Compute (TFM_COMP_Swig);                   /// H3 ← Swish(H1) * H2;
  int last_hblk = (h == model_param.num_hblks - 1);
  TFM_Config2 (TFM_CFG_SwigAcc, (last_hblk << 1)); /////// FFN output /////
  TFM_Load (w0[7], TFM_Weight);                  /// LOAD (W2);
  TFM_Compute (TFM_COMP_MatMulAcc);          /// X ← X + H3 * W2 ;
  for (int j = 6; j < 9; ++j) { /////// increment weight pointers /////
    QVec_inc_pos(&w0[j], QVEC_INC_DEPTH(wqvi[j], last_hblk));
  }
}
```

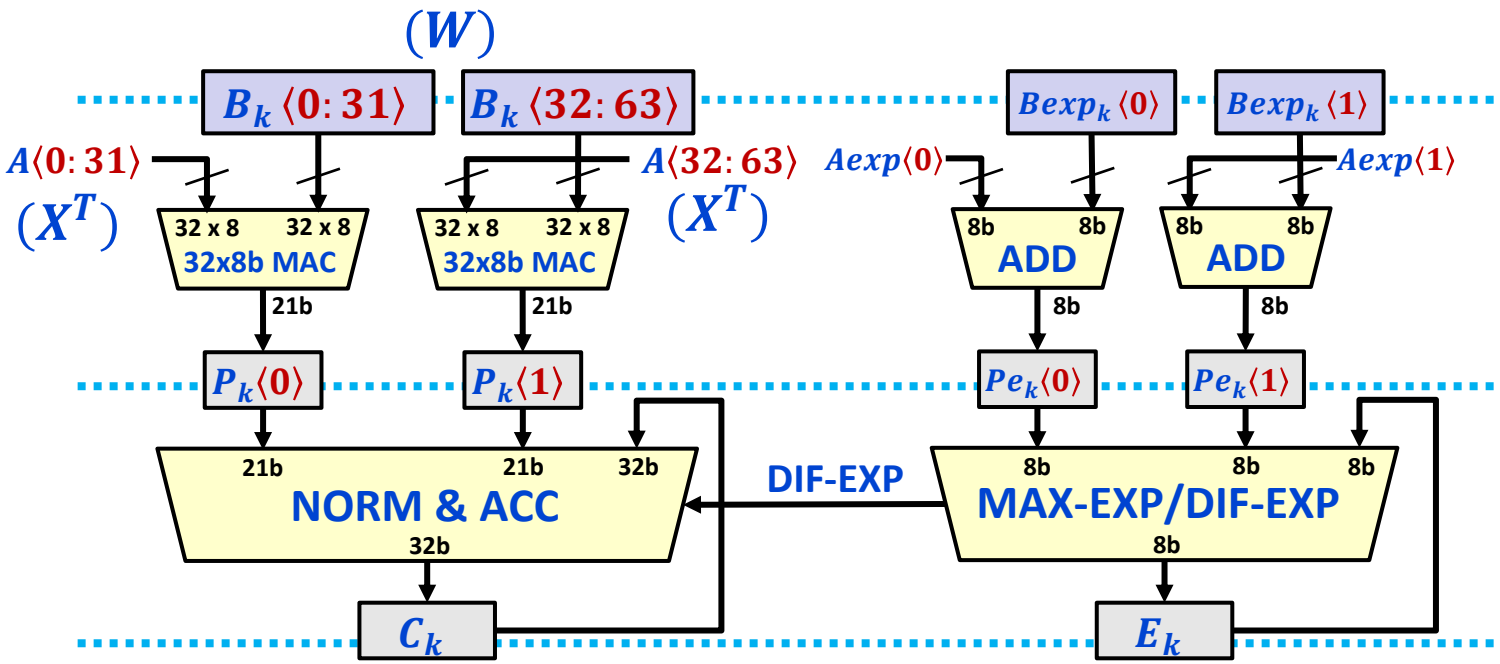
```
#define TFM_Load (qv, dtype) ¥
  TFM_LoadVector (qv.e, dtype##_Exp); ¥
  TFM_LoadVector (qv.q, dtype##_Data)
```

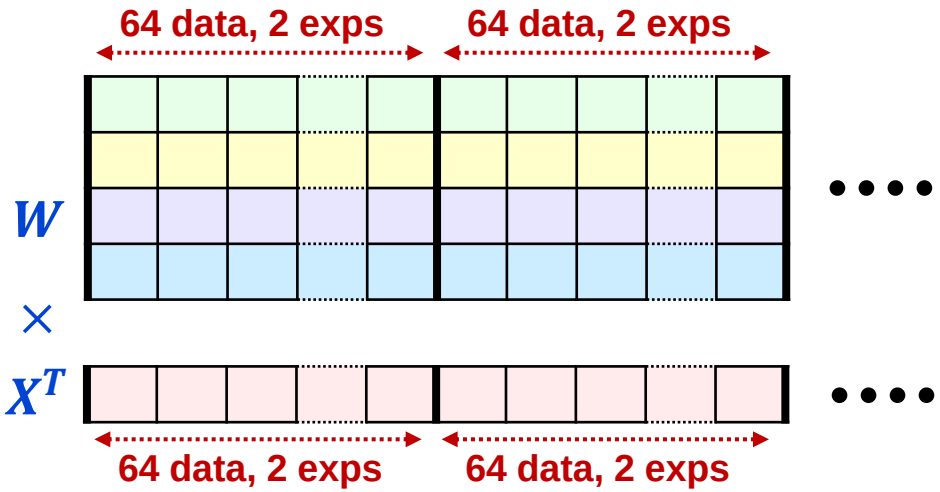
```
TFM_Load (w0[6], TFM_Weight);
```

```
TFM_LoadVector (w0[6].e, TFM_Weight_Exp);
TFM_LoadVector (w0[6].q, TFM_Weight_Data);
```

MAC Unit : MAC Pipeline

- **MAC units (2 ~ 32 units) → 2048 MACs/cycle**
  - Weight/KV-cache memory
  - **32 x 2 INT8 MACs** + normalized accumulator
  - 2-stage pipeline : MAC-stage → NORM/ACC-stage





W : stored at MAC unit memory

X^T : broadcast to all units

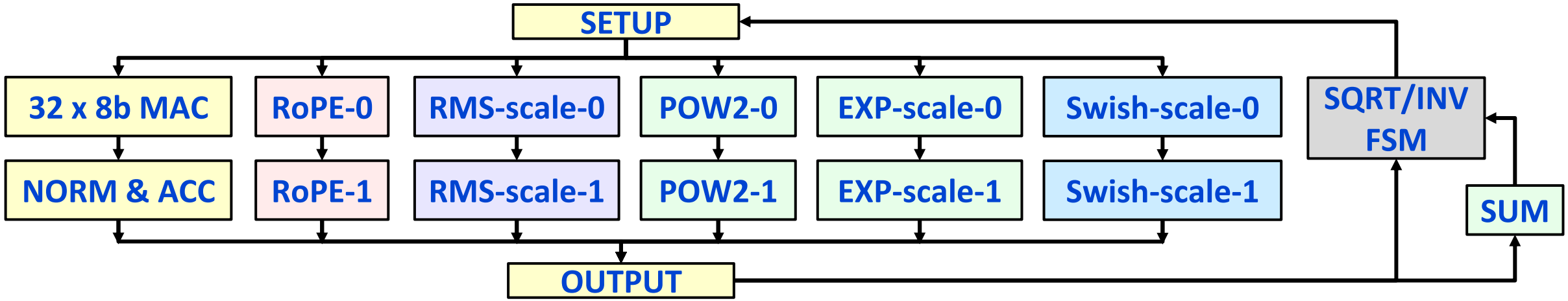
MAC pipeline : 64 MACs / cycle at each MAC unit

MAC Unit : Element-Wise Special Function Pipelines

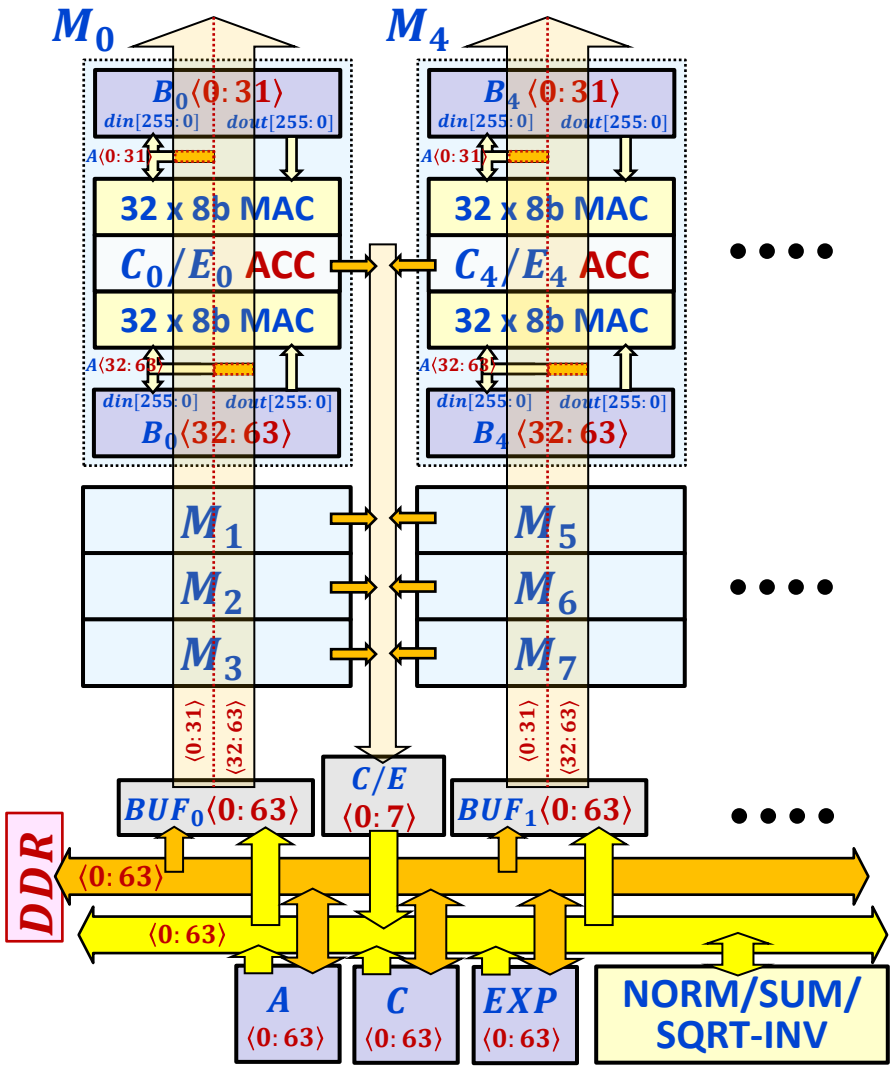
- **RoPE :** Pairwise rotation on K/Q activations
- **RMSNorm :**
  - i. Squared sum (reuse MAC pipeline : A*A)
  - ii. Square-root inverse
  - iii. Scaling with weights
- **Softmax :**
  - i. $\exp(x) = \mathrm{pow}(2, x \cdot \log_2 e)$
  - ii. Exp-sum inverse
  - iii. Exponent sum-inverse scaling
- **SWIGLU :** Element-wise scaling of Swish(H1) and H2

- **PWL LUTs :**
  - i. $\mathrm{pow}(2, x)$
  - ii. Swish(x) = $x/(1+\exp(x))$
- **SQRT/INV FSM (Newton-Raphson):**
  - i. Square-root inverse : $x^{-0.5}$
  - ii. Inverse : x^{-1}

All special function pipelines align with the MAC pipeline



Full MX-INT8 Pipeline with Output Normalization

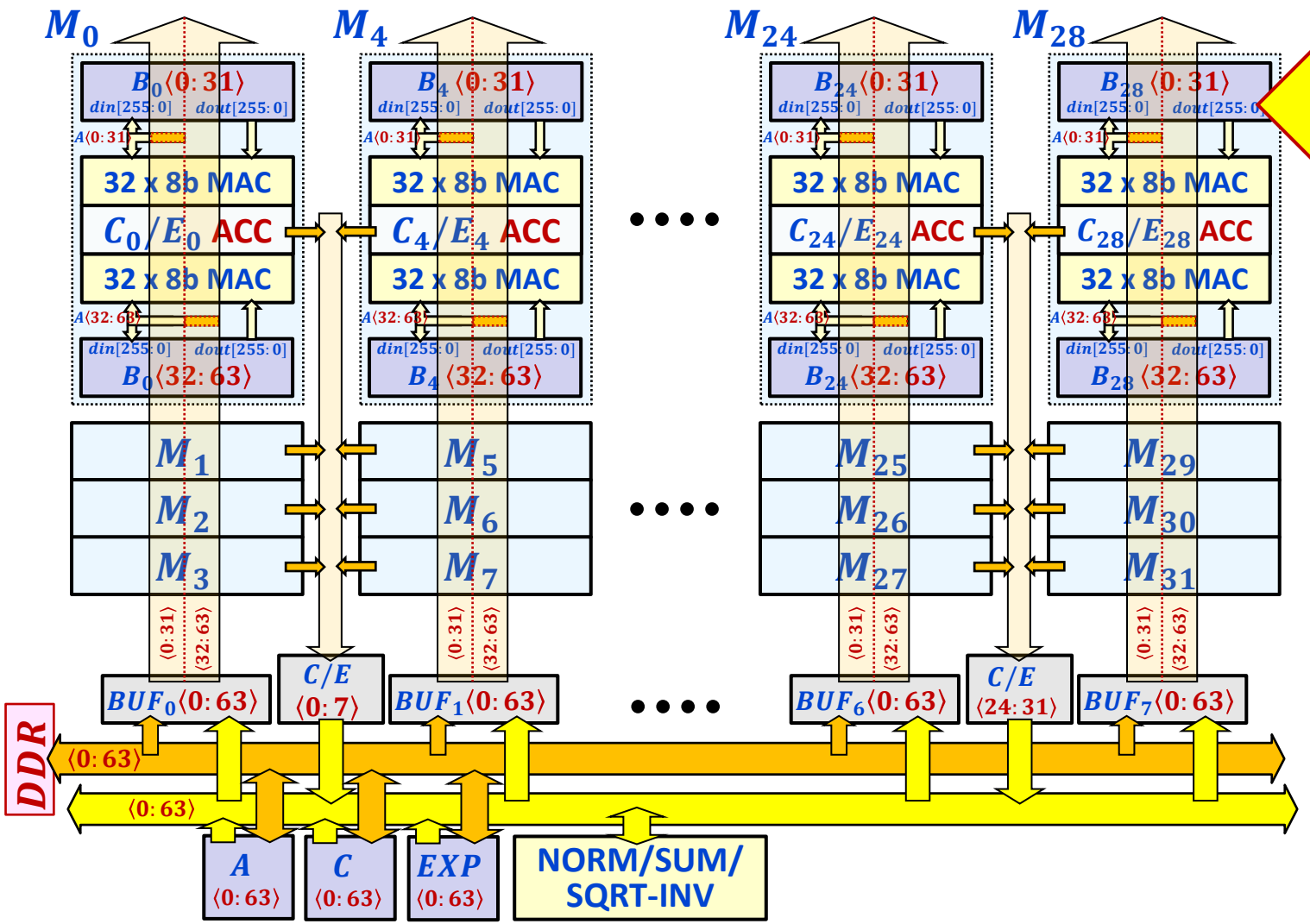


$M_{R,C} \cdot X_{C,T}$ matrix multiplication : $R \cdot C \cdot T$ MACs
$\rightarrow (R \cdot C \cdot T)/N + (7 + 32/N)$ cycles

stage	Main pipeline operations (N : # of MAC units)
0	Read Amem (64 activations)
1	Broadcast 64 activations
2	32 x 8b x N MACs / element-wise FUNC(0)
3	Normalize & accumulate / element-wise FUNC(1)
4	Latch MAC outputs
5, 6, 7	Column-wise group normalization (If $N < 32$, $32/N - 1$ extra latencies added)

stage	Row-wise group normalization pipeline (VCache)
0, 1	repeats $C \cdot T/N$ times after main pipeline completion

High-Bandwidth On-Chip/Off-Chip Memory Transfer



Transformer HW Accelerator embedded in RISC-V pipeline (in 5000 lines C++ code!!!)

64-byte wide MAC-unit memory → 2TB/s @ 32 units, 1GHz clock

Decode throughput :
(off-chip bandwidth limited)
Llama2-1.1B : 56.4 tokens/s
Llama2-7B : 9.2 tokens/s
Llama3-8B : 7.72 tokens/s

64-byte (512-bits) AXI-bus → 64GB/s @ 1GHz clock

DDR4 : 25.6GB/s per channel
DDR5 : 67.4GB/s per channel
HDM4 : >2TB/s per stack

Summary

- **RISC-V Transformer Accelerator HW design verification on C++/C2RTL :**
  - **Tightly-coupled accelerator interface** between RISC-V pipeline
  - **MX-INT8 data format** for efficient integer MAT-MUL compute
  - **Tensor-level custom instructions** : LoadVector, StoreVector, Compute, Move, Config
  - **2048 MACs/cycle @ 32 MAC units (32 x 2 INT8 MACs /unit)**
  - **RMSNorm/RoPE/Softmax/SWIGLU** functions inside MAC unit
  - Off-chip data transfer : **64GB/s @ 1GHz clock**
  - On-chip memory bandwidth : **2TB/s @ 32 MAC units, 4 TOP/s @ 1GHz**
  - **Accelerator design in C++ (5000 lines) → automatic RTL generation**
- **On-going works :**
  - **FPGA implementation :** most custom instructions tested on small testbench, will soon deploy a full stack LLM demo on Linux
  - **Chip implementation :** 32～64 MAC-unit test chip on 16nm (1st tape-out this year)
  - **System-level AI accelerator simulator :** PPA analysis, accuracy (perplexity) analysis, architecture exploration on SOTA LLM models for edge AI devices
  - Automatic SW code generation flow from open-source LLM Frameworks (ex. Hugging Face)

On-chip memory :
Llama2-1.1B : 0.946 MB
Llama2-7B : 2.70 MB
Llama3-8B : 3.78 MB

Decode throughput :
(off-chip bandwidth limited)
Llama2-1.1B : 56.4 tokens/s
Llama2-7B : 9.2 tokens/s
Llama3-8B : 7.72 tokens/s

This work was supported by JST-Research and Development Program for Next-generation Edge AI Semiconductors Grant Number JPMJES2515.