



Accelerator Architectures for Machine Learning (AAML)

Lecture 3: RISC-V for AI

Tsung Tai Yeh

Department of Computer Science
National Yang-Ming Chiao Tung University



Acknowledgements and Disclaimer

- Slides was developed in the reference with
Joel Emer, Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, ISCA 2019
tutorial
Efficient Processing of Deep Neural Network, Vivienne Sze, Yu-Hsin
Chen, Tien-Ju Yang, Joel Emer, Morgan and Claypool Publisher, 2020
Yakun Sophia Shao, EE290-2: Hardware for Machine Learning, UC
Berkeley, 2020
CS231n Convolutional Neural Networks for Visual Recognition,
Stanford University, 2020



Outline

- RISC-V for AI
 - RISC-V Instruction Extension
 - CPU Microarchitecture Optimization
- AXI Protocol



Instruction Set Architecture (ISA)

- **What is ISA?**
 - Interface between HW and SW
 - List of supported instructions that the CPU must support

```
void matrixAdd (int *A, int *B, int *C, int N, int M)
{
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            C[i*N+j] = A[i*WIDTH+j] + B[i*WIDTH+j]
}
```

Compiler

```
start_loop:
    lw    a0, 0(a5)
    lw    t1, 0(a7)
    addi  a5, a5, 4
    addi  a7, a7, 4
    add   a0, a0, t1
    sw    a0, 0(a6)
    addi  a6, a6, 4
    bne   a5, t3, start_loop
```

**Processor
Design
(HW)**

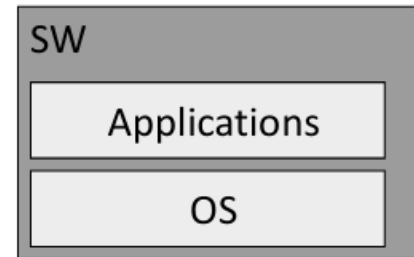
```
@18c0 -> 0007a503
@18c4 -> 0008a303
@18c8 -> 00478793
.....
```

In principle, any compiled code for a given ISA runs on any processor that implements that ISA

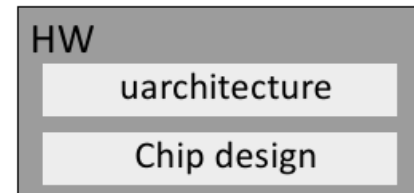


RISC-V ISA

- **Start by UC-Berkeley in 2010**
- **Contract between SW and HW**
 - Partitioned into user and privileged spec
 - External Debug
- **Standard governed by RISC-V foundation**
- **Defines 32, 64 and 128 bit ISA**
 - No implementation, just the ISA
 - Different implementations (open/close source)



Debug





RISC-V Foundation Members





What's Nice About RISC-V?

- **Open-Source (and free)**
 - Everyone can implement and use its ISA
- **Simple and Modern Design**
 - Far smaller than other commercial ISAs
 - No need to compatible with older design of same ISA (like ARM, Intel, ... which are around from 20+ years)
- **Clean-State Design**
 - Clear separation between user and privilege ISA



What's Nice About RISC-V?

- **A modular ISA**
 - Small standard base ISA + Multiple standard extensions
- **Designed for extensibility/specialization**
 - Vast opcode space available for instruction-set extensions



Are RISC-V Processors Better than XYZ?

- **Actual performance depends on the implementation**
 - RISC-V does not specify implementation details (on purpose)
- **RISC-V should deliver comparable performance**
 - If implemented well, it should perform as well as other modern ISA implementations
- **High-end processor performance is not only about ISA**
 - Implementation “details” like microarchitecture, memory hierarchy, target technology, power management are more important



What is Missing about RISC-V?

- **Still in development, but fast progress**
 - Some “advanced” extensions, including **Matrix extensions**, still being refined, adjusted
 - Tools and development environment needs to catch up
- **No canonical implementation (“the” RISC-V core)**
 - It is free to implement, so many people did so, resulting in many cores
- **Higher end (OOO, superscalar) cores not yet mature**
 - New RISC-V profile (RVA23) boosted the development of new HPC processors to be used in datacenters



RISC-V is a Load/Store Architecture

- **All operations are on internal registers**
 - Can not manipulate data in memory directly
- **Load instructions to copy from memory to registers**
- **R-type or I-type instructions to operate on them**
- **Store instructions to copy from registers back to memory**
- **Branch and Jump instructions**

C Code

```
a = b + c;
```

RISC-V assembly code

```
add a, b, c
```



RISC-V Architectural State

- **There are 32 registers, 32/64/128 bits long**
 - Name x0 to x31
 - X0 is hard wired to zero
 - A standard 'E' extension that uses only 16 registers (RV32E)
- **One program counter (PC)**
 - Byte based addressing
 - Program counter increments by 4/8/16

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6-7	t1-2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP arguments/return values	Caller
f12-17	fa2-7	FP arguments	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller



RISC-V Architectural State

- **For FP operation 32 additional FP registers**
 - “Zfinx” extension to use “integer” ones
- **Control Status Registers (CSRs)**
 - a 12-bit encoding space, allowing for up to 4,096 CSRs per hardware thread (hart)

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6-7	t1-2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP arguments/return values	Caller
f12-17	fa2-7	FP arguments	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller



RISC-V Baseline and Extensions

- **User-level ISA Spec**

- Define the normal insns for computation
- Separated into extensions, only **I** is mandatory

- **Kept very simple and extendable**

- Wide range of applications from IoT to HPC

- **RV + word-width + extensions**

- RV32IMC: 32 bit, integer, multiplication, compressed
- RV64GC: 64 bit, G = I+M+A+F+D+csr+fence and compressed

I	Integer instructions (frozen)
E	Reduced number of registers
M	Multiplication and Division
A	Atomic instructions
F	Single-Precision Floating-Point
D	Double-Precision Floating-Point
C	Compressed Instructions (frozen)
B	Bit manipulation instructions
V	Vector instructions (additional state!)
X	Non Standard Extensions



Extending ISA for ML

- Main opcodes are 7-bit (actually 5-bit due to compressed C insns)
- **Reserved** opcodes for standard extensions
- Rest of opcodes free for **custom** implementations
- **Standard extensions will be frozen/not change in the future**
- **Extensibility is integral to RISC-V ISA design!**

inst[4:2]	000	001	010	011	100	101	110	111 (> 32b)
inst[6:5]								
00	LOAD	LOAD-FP	<i>custom-0</i>	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	<i>custom-1</i>	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	<i>reserved</i>	<i>custom-2/rv128</i>	48b
11	BRANCH	JALR	<i>reserved</i>	JAL	SYSTEM	<i>reserved</i>	<i>custom-3/rv128</i>	≥ 80b



How to Build RISC-V Cores?

- **RISC-V ISA tells you the function**
 - You know which instructions are supported/want to support
 - How they are encoded
 - What they are supposed to do
- **It does **not** tell you any implementation details**
 - Pipeline stages, memory hierarchy, in-order or out-of-order
 - Everyone is free to figure out how to best implement these
- **Need to come up with a micro-architecture to implement it**



Energy-Efficient Embodied AI

- **Computing requirements/challenges for Edge AI**
 - All capacities under tight power (100 mW – 1W of budget)
 - Real-time constraints (Latency as important as Throughput)
 - Limited area budget (affects on chip memory)
 - TinyML -> low-power, embeddable, low-cost edge AI processors





Optimizing AI Deployment for Edge Devices

- **DNNs/AI Models at the edge**
 - Targets for low-power, constrained environments
 - Light-weight AI models, quantization, pruning ...
- **Specialized Hardware Architecture**
 - Optimized to run these light-weight AI models efficiently
- **Hardware-Software Co-Design**
 - Ensuring performance, energy efficiency, and reliability at the edge



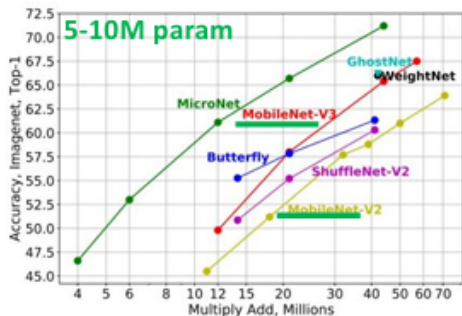
A Fast Evolving Model Zoo

- **Edge AI SoCs must provide**
 - High Performance-Power Proportionality & Flexibility

Tiny Devices (IoT)

More Powerful edge Systems

Quantized Neural Networks (QNNs)

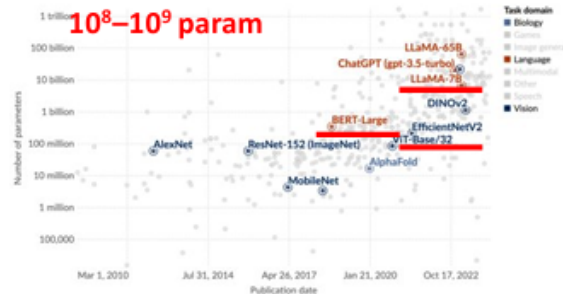


High OP/B ratio
Massive Parallelism
MAC-dominated

- Low precision is OK
- 8-/4-/2-b integer arithmetic
- Including mixed-precision

[Rusci et al., PML&S (2020)]

Edge Transformers

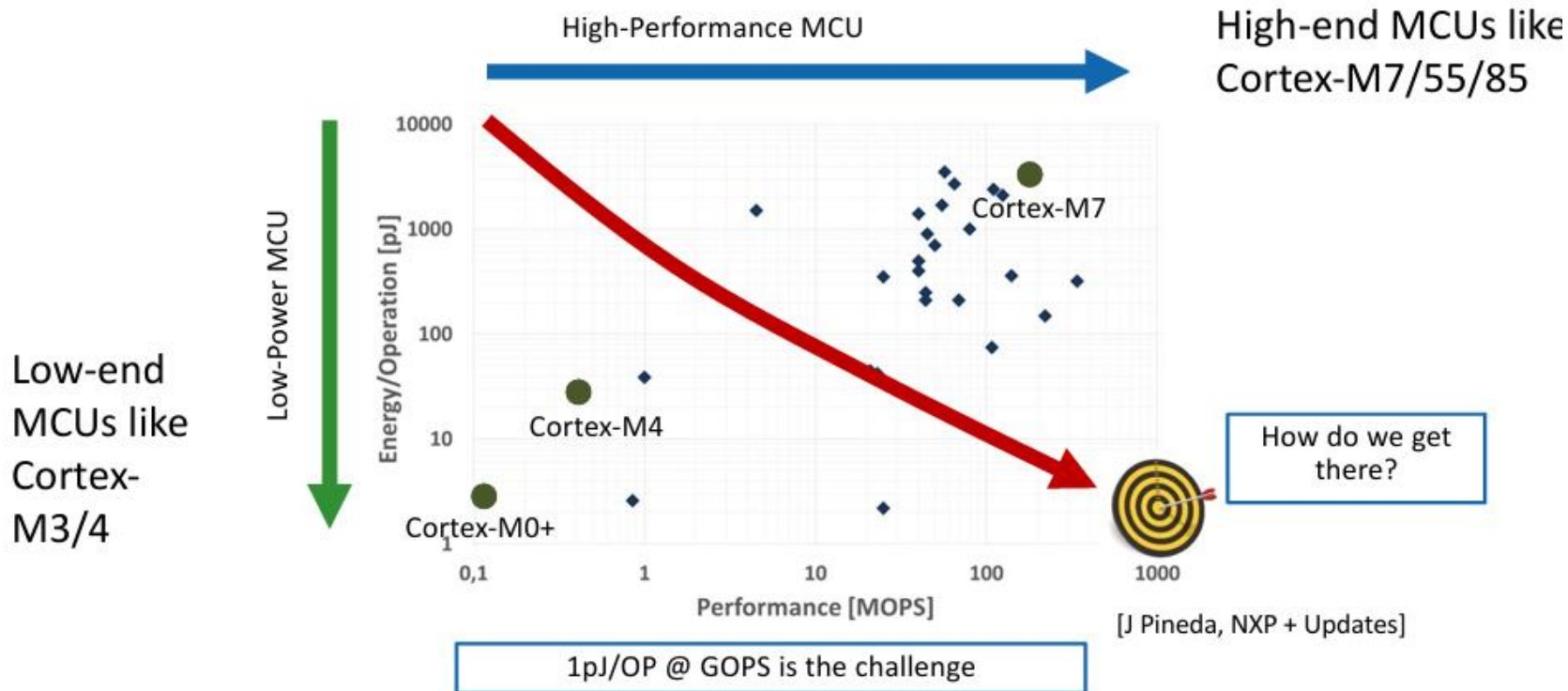


- **BF16, FP16, FP8** is the standard
- Micro-scaling fmts for reduced footprint at no accuracy loss

[Rouhani et al., arxiv23]
[Yang et al., ARITH25]



The Edge Performance-Power Proportionality Goal





Performance Metrics

- **Area**
 - in kGE equivalent (# of simple logic gates) or mm^2 (technology dependent)
- **Frequency:**
 - Depends on # of gates on longest path
- **Power:**
 - Strongly depends on the above metrics
 - **Leakage:** dissipated even when not working (Area)
 - **Dynamic Power:** dissipated on logic transitions (frequency and area)
- **CPU Design:**
 - **IPC** (Instructions per cycle)
 - IPC implicitly measured in commonly used benchmarks (Coremark, Dhrystone, SpecInt)
 - **Energy Efficiency:** OPs/Joule
- **Hardware Designer**
 - Tries to find a good balance
 - Application dependent
 - IoT and HPC have different requirements
 - **One size does not fit all**

$$P = \underbrace{ACV^2f}_{\text{Dynamic}} + \underbrace{\tau AVI_{\text{short}}f}_{\text{Short-Circuit}} + \underbrace{VI_{\text{leak}}}_{\text{Leakage}}$$



Instruction Specialization & Energy Efficiency

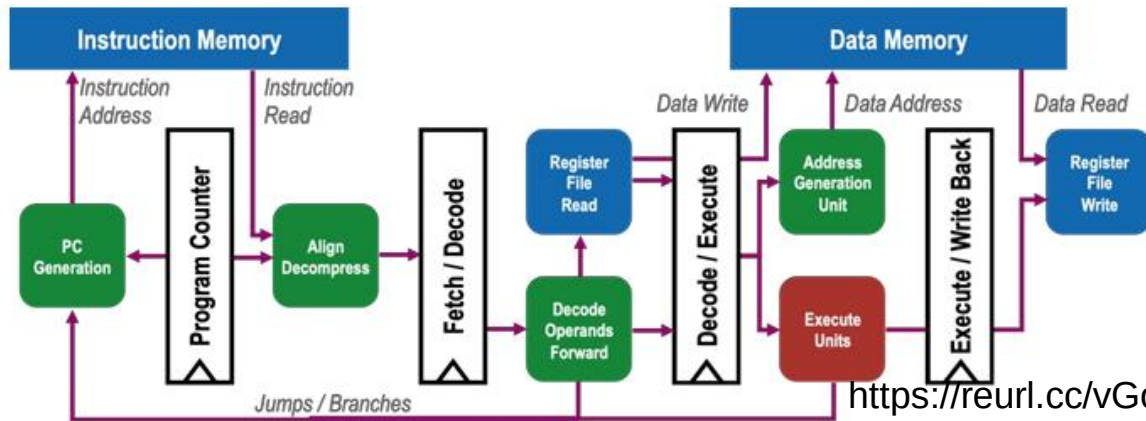
- Battery has a given energy budget (mAh@V)
- Reducing Energy consumption -> longer device lifetime
- We care about **latency & Energy**: $E = P_{avg} \cdot \text{latency}$
- More semantically dense ISA instructions helps reduce instruction count to execute a workload
- Direct advantage on latency and energy reduction if:
 - Timing of the processor (max op. freq.) not affected
 - Limited Power overhead to support new instruction



A Typical Processing Core for TinyML

- Single issue, in order is the most energy efficient
- Memory is always critical: 1 full stage for fetch, 2 full stages for load + writeback
- Single stage decode
- Single stage execute/WB

- 4-stage RISC-V RV32IMC
- 32-bit wide datapath
- 32 32-bit regs in the RF





Is RV32IM Processor Enough for AI/DSP?

FIR Inner Loop Exec. in RV32IM core →
this is part of a larger piece of code

```
for(int j=0; j<coeff_len; j++) {  
    sum += arr[i+j] * coeff[j];  
}
```

ANSI C

Lower-level view

```
for(int j=0; j<coeff_len; j++) {  
    int8 arr_int = arr[i+j];  
    int8 coeff_int = coeff[j];  
    arr_int = arr_int * coeff_int;  
    sum = sum + arr_int;  
}
```

ANSI C

per iteration:

1 MAC (multiply-accumulate) operation

7 instructions

Compute efficiency = $1/7 = 0.14$ ☹

Assembly code on RV32IM Architecture

Lstart:

lb t2, 0x0(t1)

lb a4, 0x0(t3)

mul a4, a4, t2

add t0, t0, a4

addi t3, t3, 0x1

addi t1, t1, 0x1

bne t3, a7, Lstart

RV32IM ASM

We want to minimize instructions
that are not DOTP/MAC in the
innermost loop of an arithmetic-
intensive kernel



Extensions Type

- **General Purpose: Increase computational density by fusing more operations in a single instruction**
 - **Fuse Multiplication and Addition in a single instruction/operation -> Multiply-and-Accumulate (MAC) operation**
 - Exploit regular access patterns of the workload -> Load/Store with Post-increment of the pointer
 - Boost Loop Control in HW to reduce branch penalties -> HW Loops
- **DSP**
 - Exploit low-bitwidth of DSP/AI operands and data -> Packed-SIMD ALU operations & dot-products



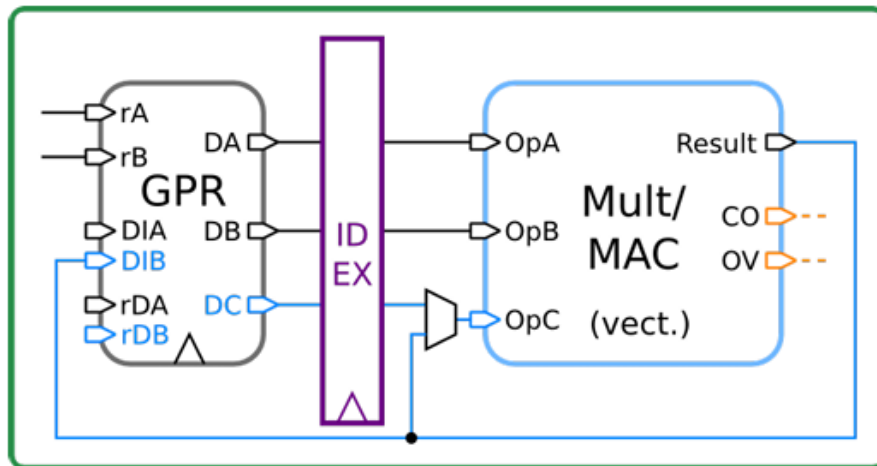
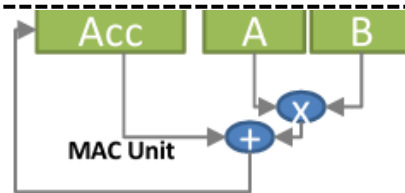
Multiply Accumulate

- Integrate the MAC unit with the register file
- Accumulator becomes also a source operand of the insn
- p.mac rD, rs1, rs2

Micro-archi requirements

- Ex unit performs MUL and ADD in the same clock cycle
 - Multiplier followed by an adder
- rD encoded to be also an input operand
 - Modifications to the Decoder
 - Additional read port to the GPR
- Pro:
 - Faster access to mac accumulation
 - Single cycle mult/mac
- Cons:
 - Additional read port on the register file
 - used for pre/post increment with register

FMA4: $\text{RegD} = (\text{RegA} * \text{RegB}) + \text{RegC}$
-> FMA RegD, RegA, RegB, RegC
FMA3: $\text{RegC} = (\text{RegC} * \text{RegB}) + \text{RegA}$
-> FMA RegC, RegA, RegB





Extensions Type

- **General Purpose: Increase computational density by fusing more operations in a single instruction**
 - Fuse Multiplication and Addition in a single instruction/operation -> Multiply-and-Accumulate (MAC) operation
 - **Exploit regular access patterns of the workload -> Load/Store with Post-increment of the pointer**
 - Boost Loop Control in HW to reduce branch penalties -> HW Loops
- **DSP**
 - Exploit low-bitwidth of DSP/AI operands and data -> Packed-SIMD ALU operations & dot-products



Post Increment Load/Store

- Standard RISC-V requires explicit update of memory pointers (**addi**)
- Solution: Automatic address update
 - Perform memory access with e.g. x10
 - In the same cycle, update x10 with the updated pointer for next memory access
 - $X10 \leftarrow (x10 + 0x1)$**

Original RISC-V

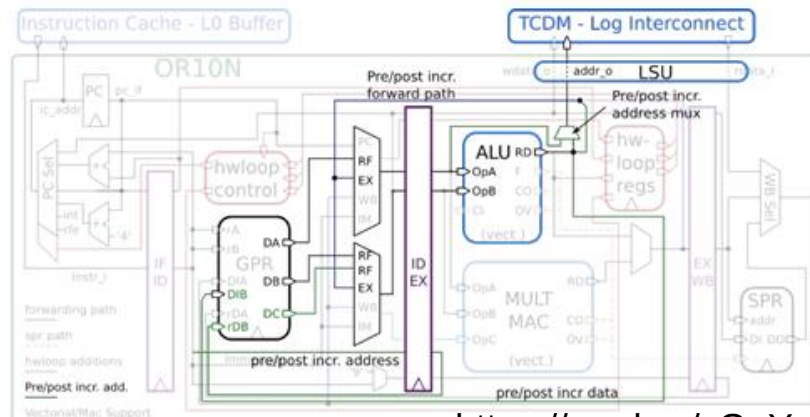
```
lb x2, 0(x10)
lb x3, 0(x12)
addi x10, x10, 1
addi x12, x12, 1
```



With auto-increment

```
p.lb x2, 0(x10!)
p.lb x3, 0(x12!)
```

You save two instructions!





Post Increment Load/Store

- **Micro-architectural requirements**

- LSU for the memory access
- ALU must be active for address update
- Additional write port to RF

- **Cost**

- 12% area overhead w.r.t. RV32IMC

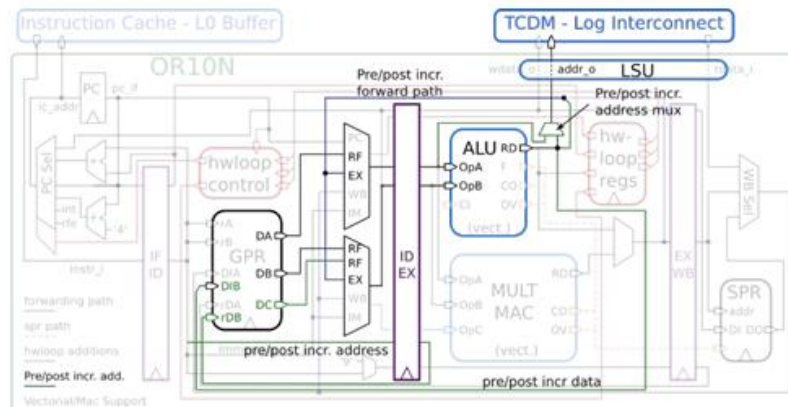
Original RISC-V

```
lb x2, 0(x10)
lb x3, 0(x12)
addi x10, x10, 1
addi x12, x12, 1
```

With auto-increment

```
p.lb x2, 0(x10!)
p.lb x3, 0(x12!)
```

You save two instructions!





RV32IM + MAC + pointer post-incr.

```
for(int j=0; j<coeff_len; j++) {  
    sum += arr[i+j] * coeff[j];  
}
```

ANSI C



Lower-level view

```
for(int j=0; j<coeff_len; j++) {  
    int8 arr_int = arr[i+j];  
    int8 coeff_int = coeff[j];  
    arr_int = arr_int * coeff_int;  
    sum = sum + arr_int;  
}
```

per iteration:

1 MAC (multiply-accumulate)
operation

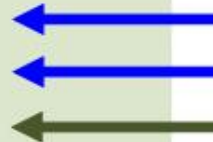
4 instructions

efficiency = $1/4 = 0.25$ ☹



a7 = coeff_len

```
Lstart:  
p.lb t2, 0x0(t1!)  
p.lb a4, 0x0(t3!)  
p.mac t0, a4, t2  
bne t3, a7, Lstart
```



RV32IMXpulp ASM

Implicit update after load operation:
t1 = t1 + 0x1 (Byte)

one instruction is spent to control the loop
→ Can I delegate this operation to
Hardware?

<https://reurl.cc/vGoYyL>



Extensions Type

- **General Purpose: Increase computational density by fusing more operations in a single instruction**
 - Fuse Multiplication and Addition in a single instruction/operation -> Multiply-and-Accumulate (MAC) operation
 - Exploit regular access patterns of the workload -> Load/Store with Post-increment of the pointer
 - **Boost Loop Control in HW to reduce branch penalties -> HW Loops**
- **DSP**
 - Exploit low-bitwidth of DSP/AI operands and data -> Packed-SIMD ALU operations & dot-products



Hardware (Zero-Overhead) Loops

- Loops setup beforehand through three custom instructions (outside hot loop)
 - Lp.start** → sets a SP reg with start addr. of the loop
 - Lp.end** → sets a SP reg with end addr. of the loop
 - Lp.count(lp.counti)** → sets # of iterations
 - no restrictions on start/end address

- Fast instruction

- lp.setup, lp.setupi**

- start_addr = PC+4
- end_addr = start + offset
- counter = IMM or GP-REG

```
c = 0;
for(i=0;i<100;i++)
    c = c + a[i]*b[i];
```

Original RISC-V

```
mv    x4, 100
mv    x5, 0
Lstart:
addi  x4, x4, -1
lw    x8, 0(x9)
lw    x10, 0(x11)
add   x9, x9, 4
add   x11, x11, 4
mul   x8, x8, x10
add   x5, x5, x8
bne   x4, x0, Lstart
```

w/ HW Loops

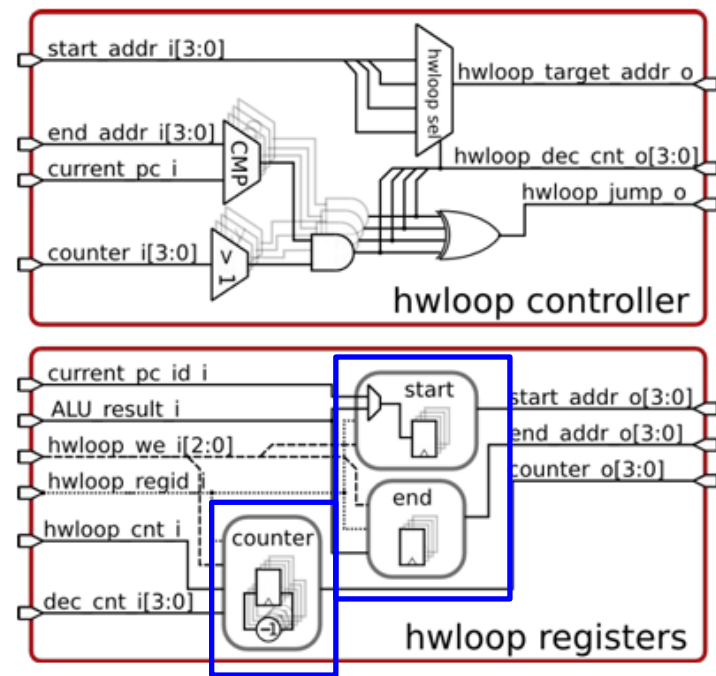
```
mv    x5, 0
lp.setupi 100, Lend
lw    x8, 0(x9)
lw    x10, 0(x11)
add   x9, x9, 4
add   x11, x11, 4
mul   x8, x8, x10
Lend: add x5, x5, x8
```

**No explicit instruction to
ctrl the loop!**



Hardware Loop Implementation

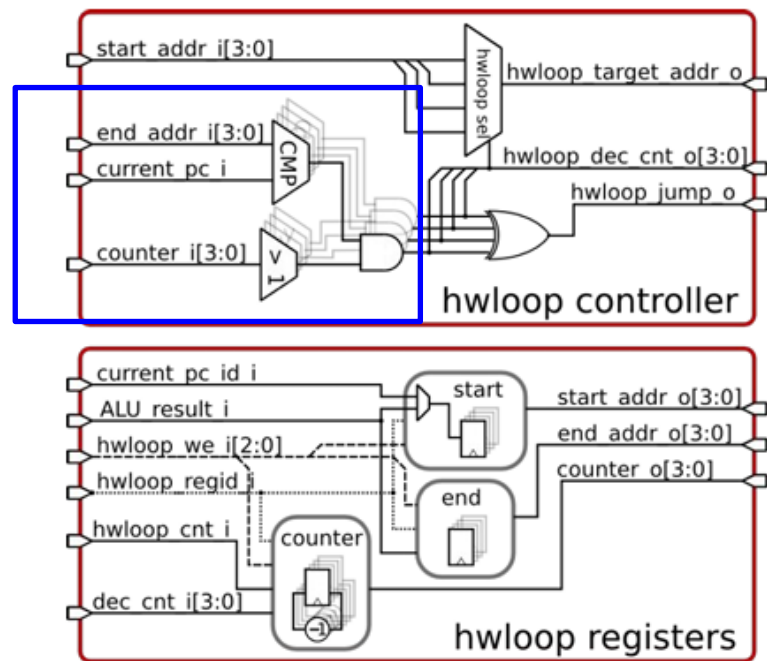
- **Small HW engine automatically handles the loop**
- **Hwloop registers: two sets of three regs**
 - Start, end addresses of loop + counter of # of iterations
 - Two sets are needed to accelerate nested loops
 - More sets would require higher micro-archi overhead and bring no significant benefits





Hardware Loop Implementation

- **Hwloop controller:** logic unit that automatically decrements the counter
 - Once # of iterations reached -> the controller will generate a “jump” for the processor
- **Area cost**
 - 5% area overhead
- **Performance**
 - Speedup can be a factor of 2X!
 - Small loops benefit more!





RV32IM + MAC + pointer post-incr. + HW Loop

```
for(int j=0; j<coeff_len; j++) {  
    sum += arr[i+j] * coeff[j];  
}
```

ANSI C



Lower-level view

```
for(int j=0; j<coeff_len; j++) {  
    int8 arr_int = arr[i+j];  
    int8 coeff_int = coeff[j];  
    arr_int = arr_int * coeff_int;  
    sum = sum + arr_int;  
}
```

per iteration:

1 MAC (multiply-accumulate)

operation

3 instructions

efficiency = $1/3 = 0.33$ 😊



a7 = coeff_len

lp.setup t3, a7, Lend ←

Lstart:

p.lb t2, 0x0(t1!) ←

p.lb a4, 0x0(t3!) ←

Lend:

p.mac t0, a4, t2 ←

RV32IMXpulp ASM

one element at time (1 Byte) is loaded and then processed. However, my processor datapath has a 32-bit wide parallelism → How to exploit it?



Extensions Type

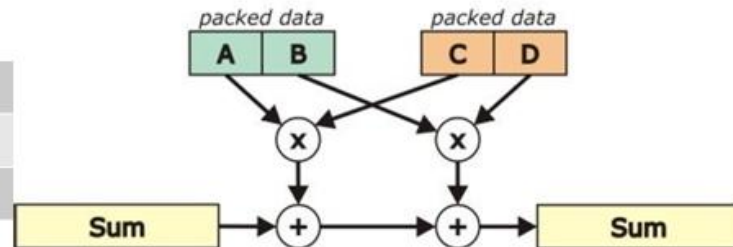
- **General Purpose: Increase computational density by fusing more operations in a single instruction**
 - Fuse Multiplication and Addition in a single instruction/operation -> Multiply-and-Accumulate (MAC) operation
 - Exploit regular access patterns of the workload -> Load/Store with Post-increment of the pointer
 - Boost Loop Control in HW to reduce branch penalties -> HW Loops
- **DSP**
 - **Exploit low-bitwidth of DSP/AI operands and data -> Packed-SIMD ALU operations & dot-products**



Packed-SIMD

- **DSP/TinyML inference are OK with low-bitwidth operands**
- **Packed-SIMD extensions**
 - Well known idea from classical DSP -> widely used (ARM, x86)
 - Efficient usage of hardware resources to target low-bitwidth arithmetic
 - Target embedded systems -> RV Vector ISA more for high performance
- **SIMD in 32-bit machines**
 - Vectors are either four 8-bits elements or two 16-bit elements

Computation	add, sub, shift, avg, abs, dot product
Compare	min, max, compare
Manipulate	extract, pack, shuffle



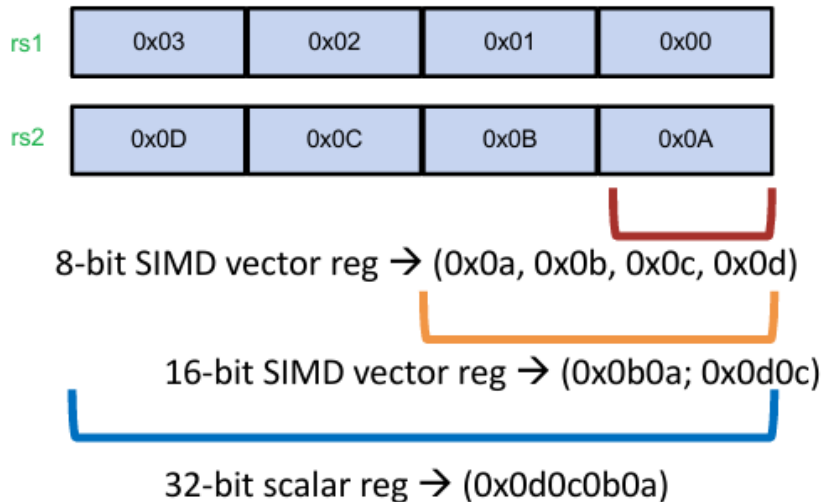


Packed-SIMD ALU Instructions

- **Same Register-file**

- The instruction encodes how to interpret the content of the register

add rD, rs1, rs2	$rD = 0x03020100 + 0x0D0C0B0A$
add.h rD, rs1, rs2	$rD[0] = 0x0100 + 0x0B0A$ $rD[1] = 0x0302 + 0x0D0C$
add.b rD, rs1, rs2	$rD[0] = 0x00 + 0x0A$ $rD[1] = 0x01 + 0x0B$ $rD[2] = 0x02 + 0x0C$ $rD[3] = 0x03 + 0x0D$





Packed-SIMD Vectorial Adder

- We reuse the scalar adder
- Supported mode: word, half, byte
- We just play some tricks with the carry chain! (32b adder -> 35b adder)

word carry

- $A[34:0] = \{A[31:24], 1'b1, A[23:16], 1'b1, A[15:8], 1'b1, A[7:0]\}$
- $B[34:0] = \{B[31:24], 1'b0, B[23:16], 1'b0, B[15:8], 1'b0, B[7:0]\}$

halfword carry

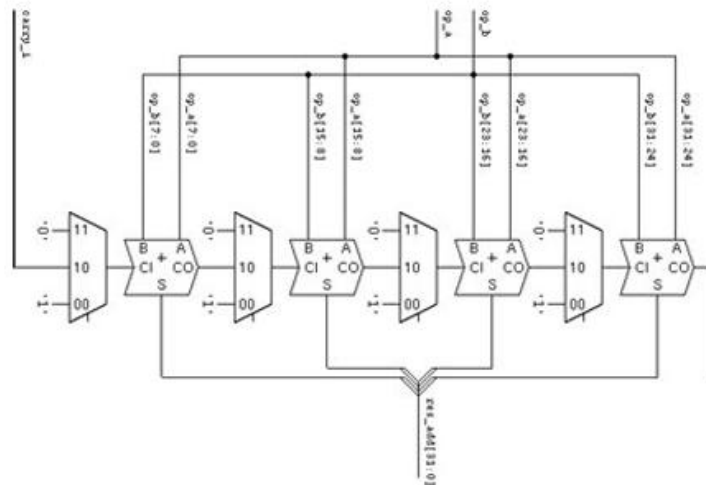
- $A[34:0] = \{A[31:24], 1'b1, A[23:16], 1'b0, A[15:8], 1'b1, A[7:0]\}$
- $B[34:0] = \{B[31:24], 1'b0, B[23:16], 1'b0, B[15:8], 1'b0, B[7:0]\}$

byte carry

- $A[34:0] = \{A[31:24], 1'b0, A[23:16], 1'b0, A[15:8], 1'b0, A[7:0]\}$
- $B[34:0] = \{B[31:24], 1'b0, B[23:16], 1'b0, B[15:8], 1'b0, B[7:0]\}$

RESULT

- $C[34:0] = A+B$; **$RES[31:0] = \{C[34:27], C[25:18], C[16:9], C[7:0]\}$**



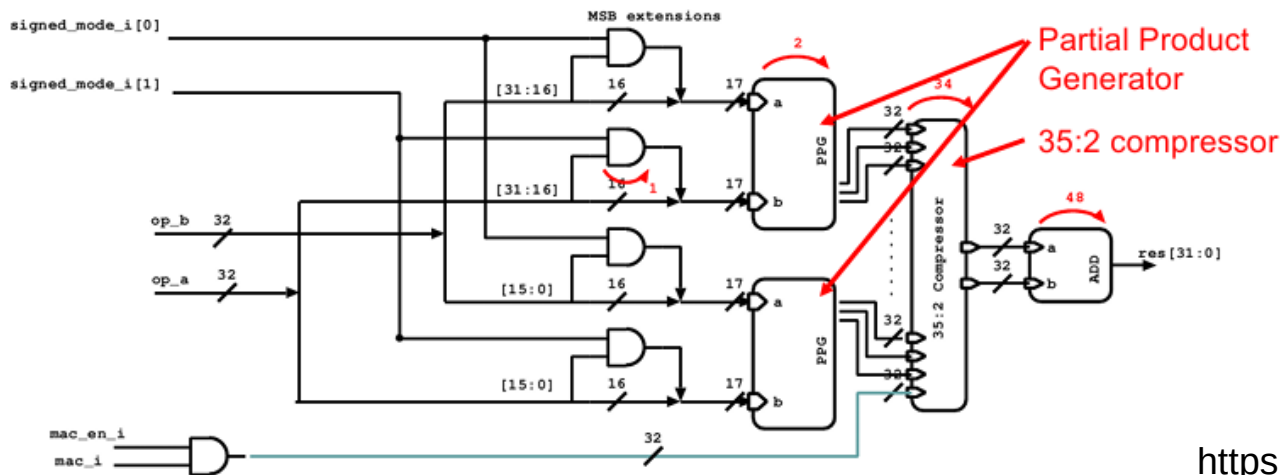


Expanding SIMD Dot Product

- Packed-SIMD Dot Product: half word example**

$$C[31:0] = \underbrace{A[31:16] * B[31:16]}_{16*16 \text{ bit}} + \underbrace{A[15:0] * B[15:0]}_{16*16 \text{ bit}} + \underbrace{C[31:0]}_{32 \text{ bit}}$$

→ 2 multiplications, 1 addition, 1 accumulation in 1 cycle (2x for bytes)





Benefits of Packed-SIMD Instructions

- **Parallelizes operations (2x to 4x speed gain)**
- **Minimizes the number of Load/Store instruction** for exchange between memory and register file (up 2 or 4 data transferred at once)
 - Vector elements likely at contiguous addresses in memory
Fetch 32-bit at a time (e.g. two 16-bit elements of a vector or four 8-bit elements of a vector)
 - Improve register file use (1 register holds 2 or 4 values)
- Can be used to dramatically increase the performance of most DSP algorithms running on embedded systems



Compiler Builtins for Extension

No Packed SIMD data type in GCC or LLVM compiler → Hardware SIMD operations to be inferred 'manually' in C code

- We can exploits Intrinsics: special functions that map directly to inlined DSP instructions.

- **Dot-product without accumulation between unsigned/ signed char vectors (v4u/v4s):**

```
S = __builtin_dotu/sp4(A, B); // S = A[0]*B[0] + A[1]*B[1] + A[2]*B[2] + A[3]*B[3],  
A is v4u/v4s, B is v4u/v4s
```

- **Also with mixed signs:**

```
S = __builtin_dotusp4(A, B); // S = A[0]*B[0] + A[1]*B[1] + A[2]*B[2] + A[3]*B[3] ,  
A is v4u, B is v4s
```

- **All of these are also available for 16-b and with accumulation (over accumulator S):**

```
S = __builtin_sdotup4(A, B, S);  
S = __builtin_sdotsp4(A, B, S);  
S = __builtin_sdotusp4(A, B, S);  
S = __builtin_sdotup2(A, B, S);  
S = __builtin_sdotsp2(A, B, S);  
S = __builtin_sdotusp2(A, B, S);
```



RV32IM + MAC + pointer post-incr. + HW Loop + DSP

```
for(int j=0; j<coeff_len; j++) {  
    sum += arr[i+j] * coeff[j];  
}
```

ANSI C



Lower-level view

```
for(int j=0; j<coeff_len/4; j++) {  
    int8 arr_int = arr[i+j];  
    int8 coeff_int = coeff[j];  
    arr_int = arr_int * coeff_int;  
    sum = sum + arr_int;  
}
```



```
a7 = coeff_len / 4  
lp.setup t3, a7, Lend
```

```
Lstart:  
p.lw t2, 0x0(t1!)  
p.lw a4, 0x0(t3!)  
Lend:  
pv.sdotp (t0, a4, t2)
```

Load 1 SIMD
vector (32-bit), of
four 8-bit
elements

Loop
coeff_len/ 4
times!

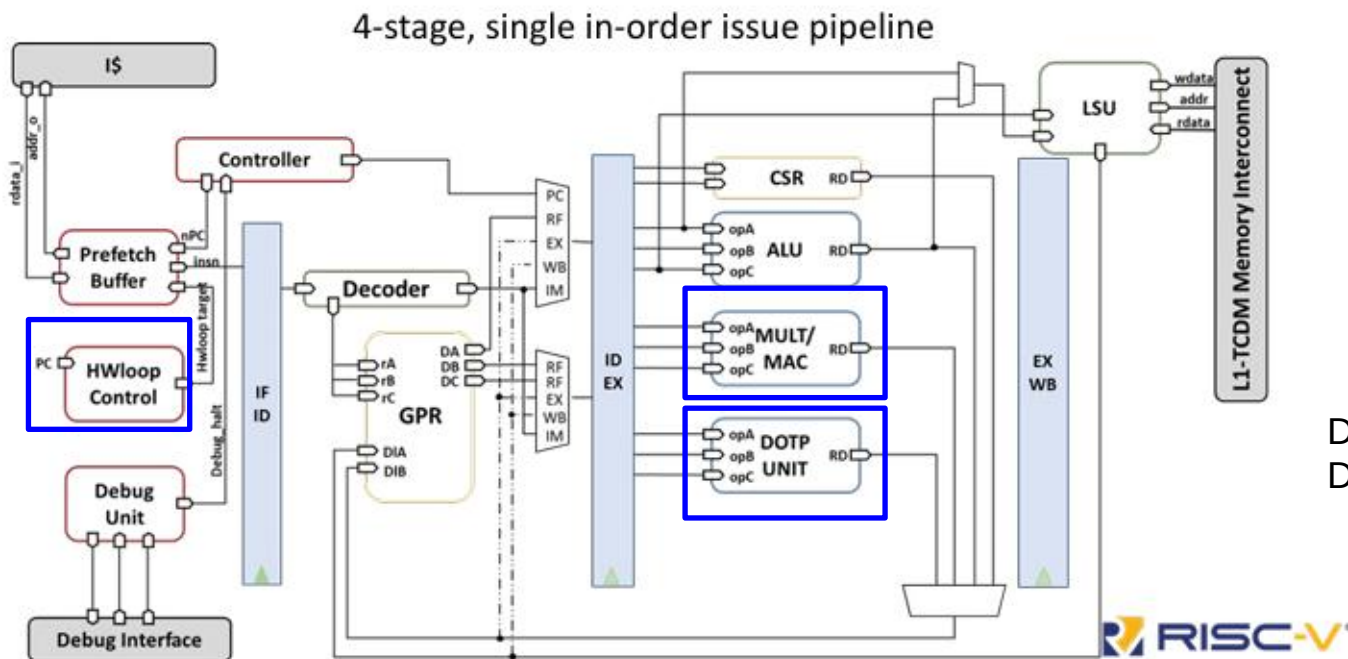
Perform 4 MAC ops on SIMD vectors
Each SIMD vector contains 4 8-bit
elements

per iteration:
1 MAC **SIMD** Operation
3 instructions
efficiency = $1/3 = 0.33$ ☺

But higher throughput compared to
non-SIMD case!!



RI5CY Implements the Xpulp ISA

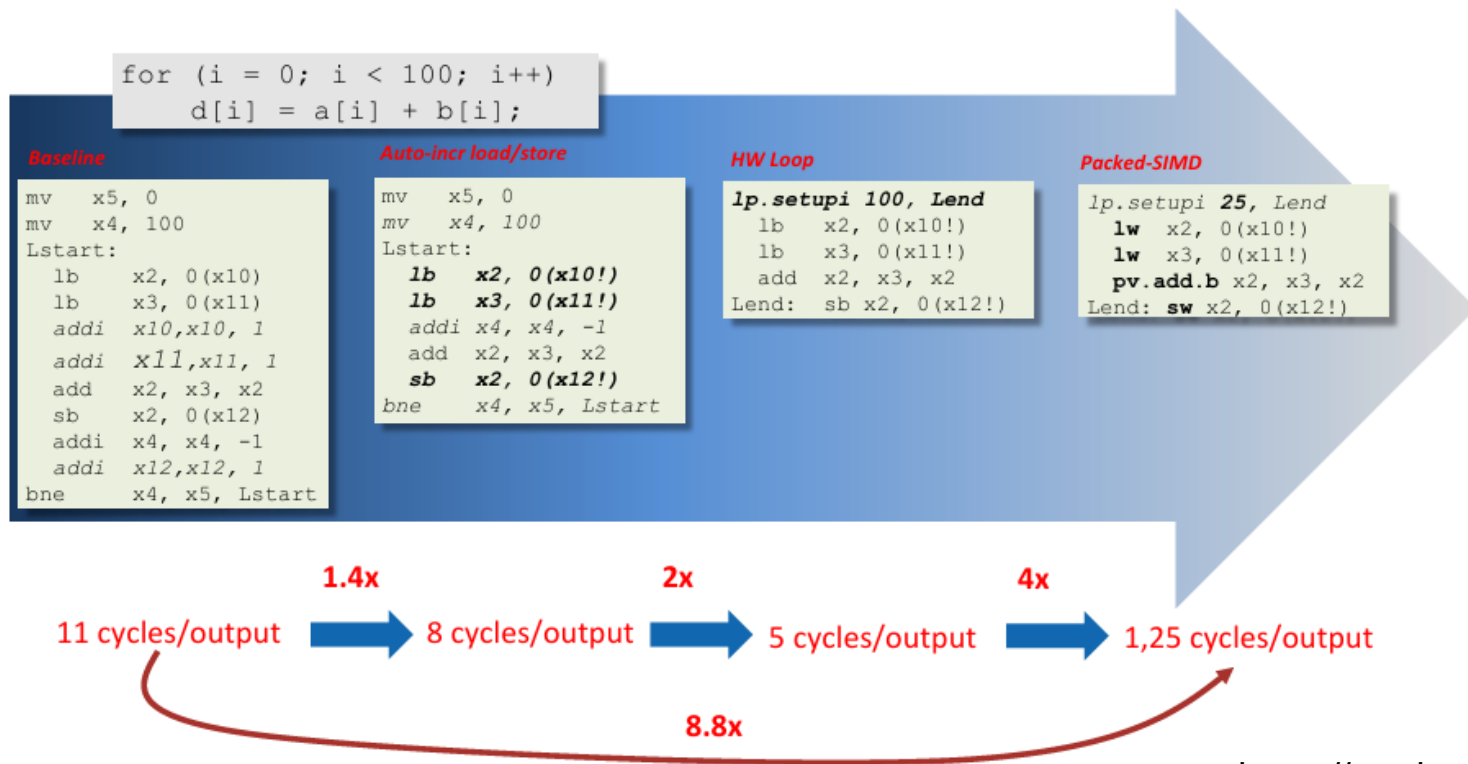


DOTP:
Dot-Product

- ❑ [Baseline] RV32IMC: not good for machine learning workload (~44kGE)
- ❑ [DSP] Xpulp: 16/8-bit SIMD, HW loops, post-modified LD/ST & bit-manip. insns (~55kGE)



RI5CY Implements the Xpulp ISA





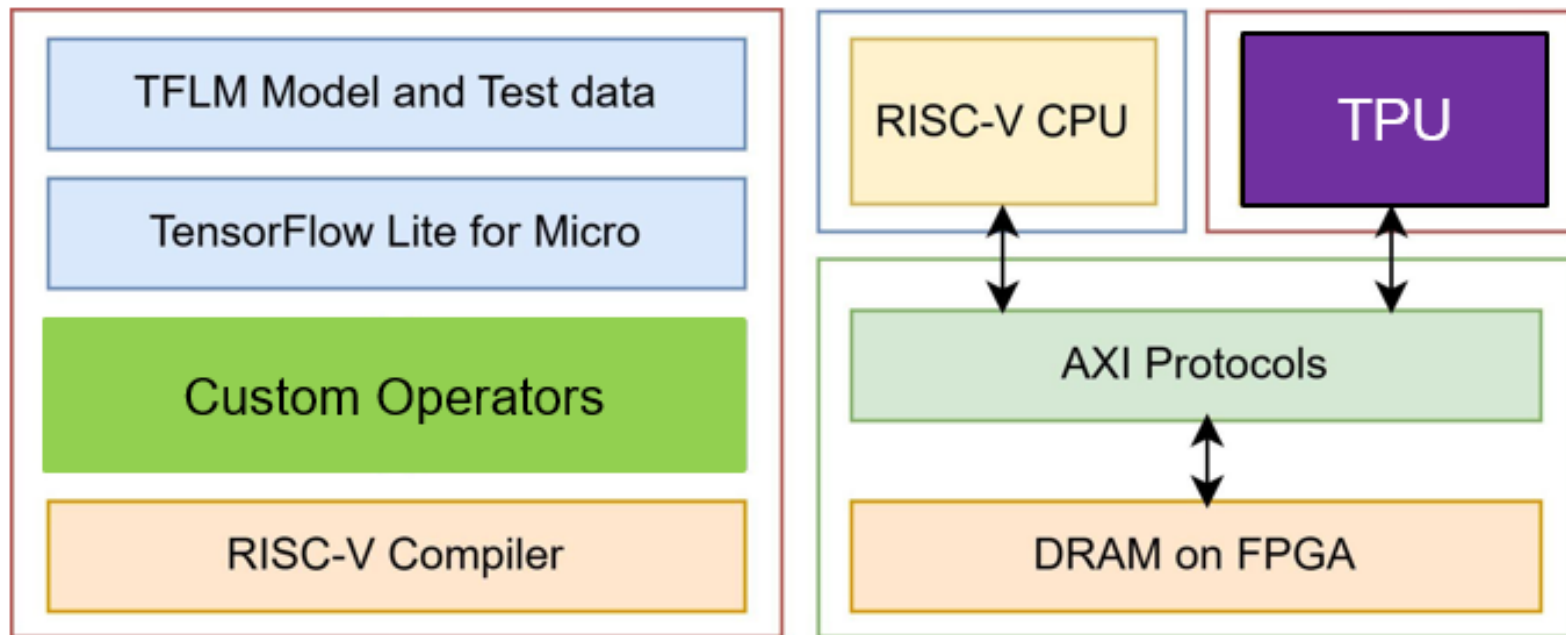
How to Add an ISA Extension

- **Analyze application and identify bottlenecks**
- **Design the instruction (instruction format, encoding, src-dst regs)**
- **Describe the instruction in RISC-V compiler (LLVM or GCC)**
 - Back-end (binutils, disassembler, etc) for debugging & assembly-level or intrinsics usage
 - Front-end (compiler optimization steps) for C to assembly translation and optimization
- **(Optional) Test new instruction into an instruction set simulator**
- **Describe the micro-architecture (extend the core)**
- **Evaluate the power, performance, area cost of the micro-archi**



TinyRISC-V SoC Platform

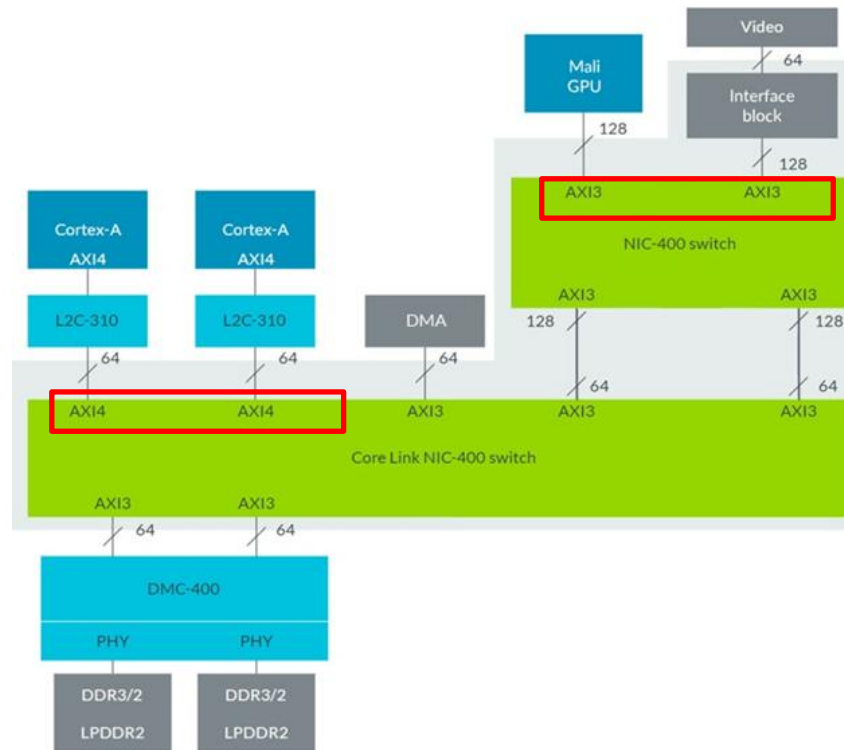
- Include CPU + Systolic Array interconnected with AXI protocol





What is AMBA and why use it?

- **What is AMBA?**
 - The Arm Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, on-chip interconnect specification
 - The connection and management of functional blocks in system-on-chip (SoC) design
- AMBA protocols like AXI4 and AXI3





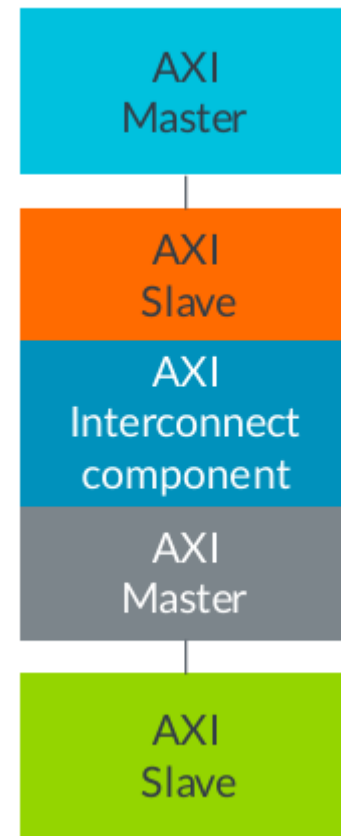
What is AMBA and why use it?

- **Where is AMBA used?**
 - AMBA simplifies the development of designs with multiple processors and large numbers of controllers and peripherals
 - AMBA is widely used in a range of ASIC and SoC parts
- **Why use AMBA?**
 - Efficient IP reuse
 - Provide the interface standard that enables IP reuse
 - Flexibility
 - Offers the flexibility to work with a range of SoCs



AXI Protocol Overview

- **What is AXI protocol?**
 - A **point-to-point specification**, not a bus specification
 - Describes only the signals and timing between interface
 - All AXI connections are between master interfaces and slave interfaces
 - AXI interconnect interfaces contain the same signals
 - Makes integration of different IP simple
 - The direct connection gives maximum bandwidth between the master and slave components with no extra logic





Variations of AXI4

- **There exists three types of AXI interface**
 - **AXI4:**
 - The high-performance interface, suited for memory mapped communication
 - AxLEN[7:0] has 8 bits, allowing bursts of up to 256 data transfers in a transaction
 - **AXI4-Lite:**
 - A light-weight variant of the interface, used for memory mapped single transactions
 - This variation does not support burst data, only provides a single data transfer per transaction



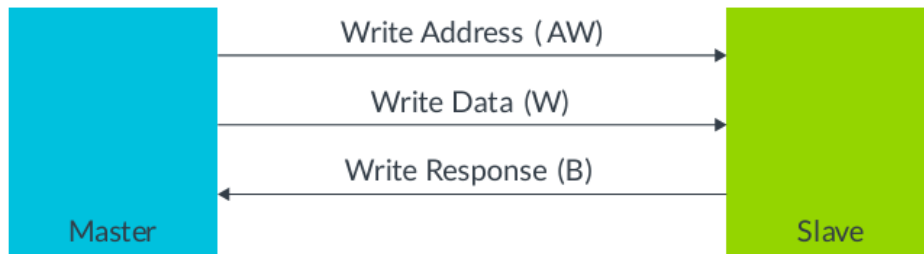
Variations of AXI4

- **There exists three types of AXI interface**
 - **AXI4-Stream:**
 - Allows for an unlimited data burst size
 - A single channel is defined for the transmission of streaming data
 - Connection is from master to slave only
 - If bidirectional transfers are required both peripherals must be of type master/slave



AXI Channels

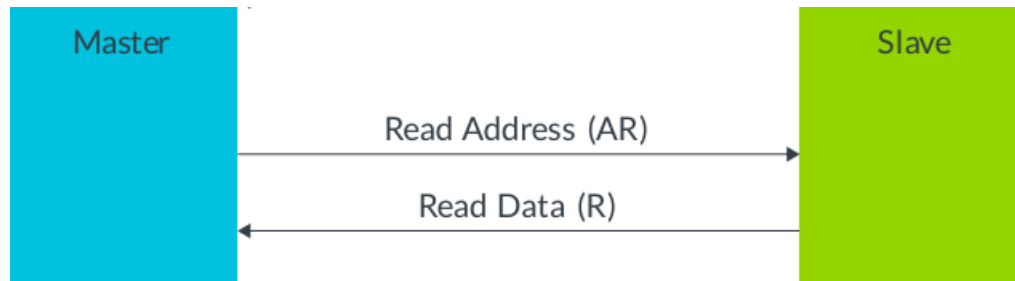
- **Write operations use the following channels**
 - The master sends an address on the **Write Address (AW)** channel
 - Transfer data on the **Write Data (W)** channel to the slave
 - The slave writes the received data to the specified address
 - Once the slave has complete the write operation
 - Responds a message to master on **Write Response (B)** channel
 - Each channel is unidirectional -> a separate B channel is needed to pass responses back to the master





AXI Channels

- **Read operations use the following channels**
 - The master sends an addr. it wants to read on **Read Address (AR)** channel
 - The slave sends the data from the requested address to the master on the **Read Data (R)** channel
 - The slave can also return an error message on **Read Data (R)** channel
 - No need for a Read Response (RR) channel, RR is passed as part of the Read Data channel





Main AXI Features

- **Independent read and write channels**
 - AXI supports two different sets of channels (write and read)
 - Having two independent sets of channel helps to improve bandwidth performance of the interface
 - Read and write operations can happen at the same time
- **Multiple outstanding addresses**
 - A master can issue transactions without waiting for earlier transaction to complete
- **No strict timing relationship between address and data oper.**
 - There is no timing requirement for when the master has to provide the corresponding data to write on the Write Data channel



Main AXI Features

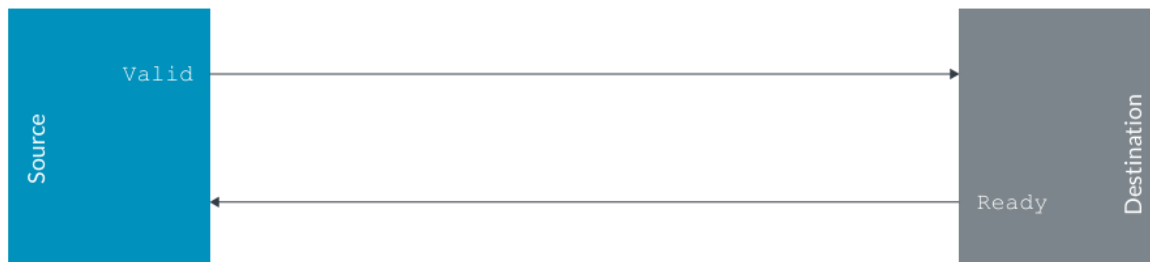
- **Support for unaligned data transfer**
 - A 32-bit data packet that starts at a byte address of 0x1002 is not aligned to the natural 32-bit address boundary
- **Out-of-order transaction completion**
 - There is no restriction on the completion of transactions
 - A single physical port can support out-of-order transactions by acting as server logical ports
- **Burst transactions based on start address**
 - AXI masters only issue the starting address for the first transfer
 - Any following transfer, the slave will calculate the next transfer address based on the burst type



Channel Transfers and Transactions

- **Channel handshake**

- The AXI4 protocol defines five different channels
- All of these channels share the same handshake mechanism that is based on the VALID and READY signals
- The source uses the VALID signal to indicate when valid information is available
 - The VALID signal must remain asserted until the destination accepts

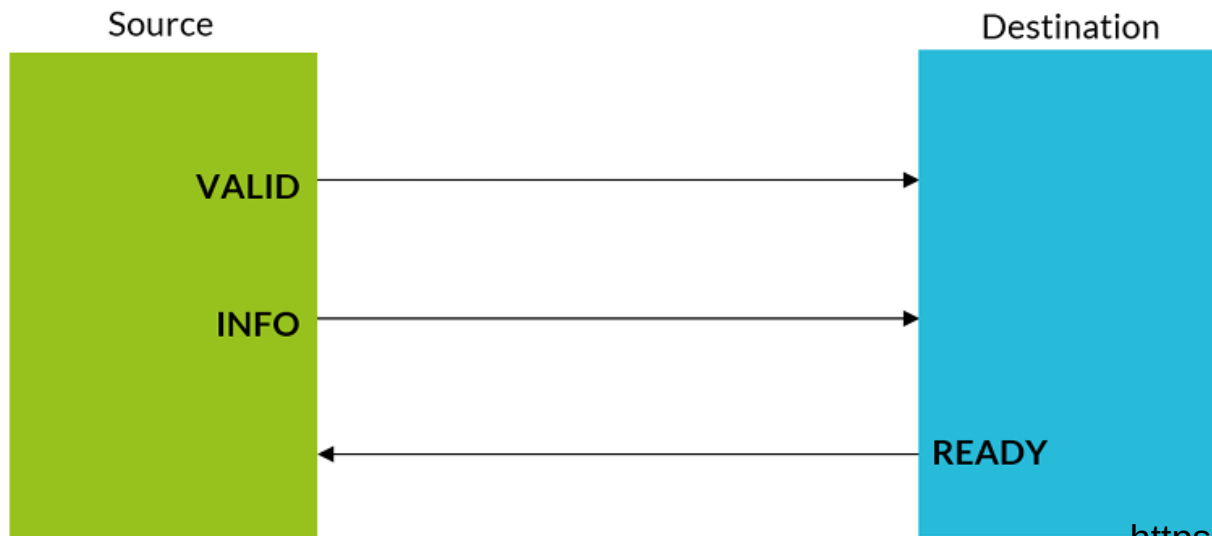




Channel Transfers and Transactions

- **Transfer**

- A transfer is a single exchange of information
- With one VALID and READY handshake

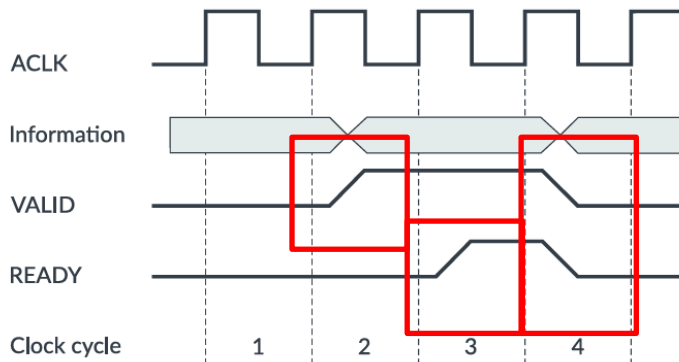




Channel Transfers and Transactions

- **Channel transfer examples**

- In clock cycle 2, the VALID signal is asserted, indicating that the data on the information channel is valid
- In clock cycle 3, the following clock cycle, the READ signal is asserted
- The handshake completes on the rising edge of clock cycle 4
 - Both READY and VALID signals are asserted

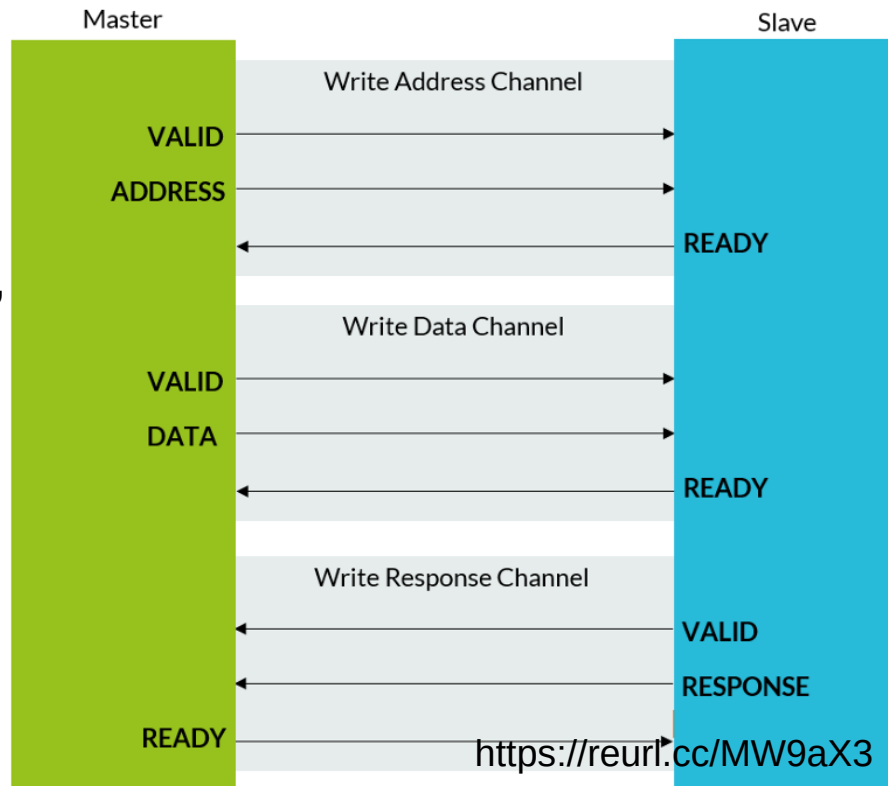




Channel Transfers and Transactions

- **Transaction**

- A transaction is an entire burst of transfer
- **In a write transaction:**
Containing an address transfer,
one or more data transfers,
for write sequences, a
response transfer

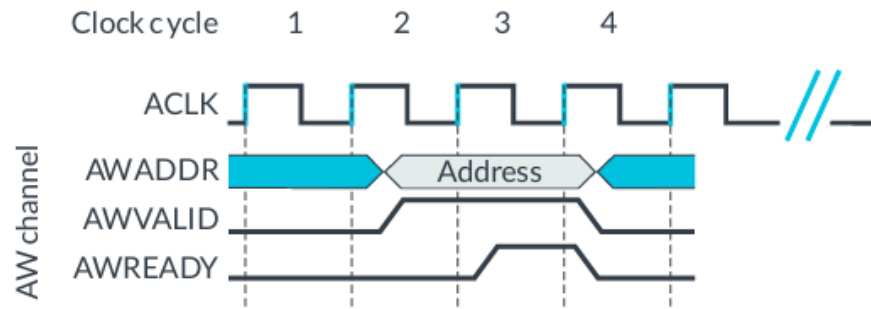




Channel Transfers and Transactions

- **Write transaction: single data item**

- The write transaction involves the following channels
 - Write Address (AW)
 - Write (W)
 - Write Response (B)
- The handshake is where the master communicates the address of the write to the slave

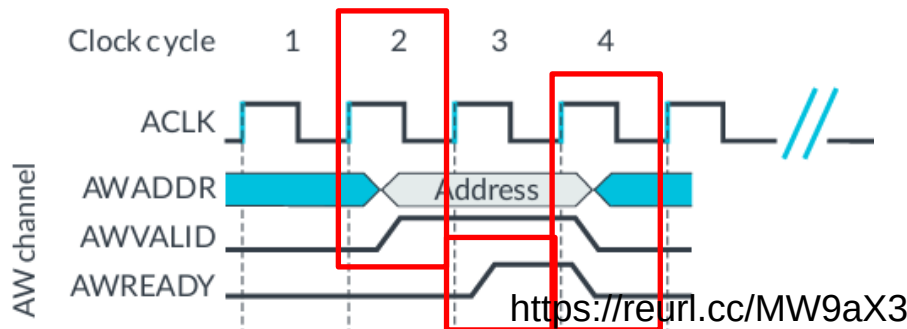




Channel Transfers and Transactions

- **Write transaction: single data item**

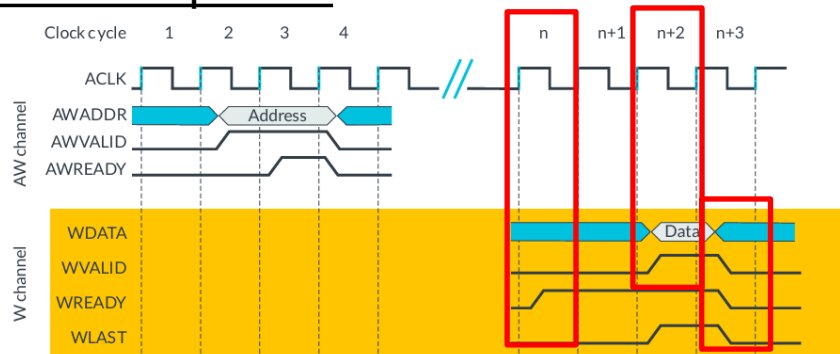
- The master puts the address on AWADDR and asserts AWVALID in clock cycle 2
- The slave asserts AWREADY in clock cycle 3 to indicate its ability to receive the address value
- The handshake completes on the rising edge of clock cycle 4
- After this first handshake, the master transfers the data to the slave on the Write channel





Channel Transfers and Transactions

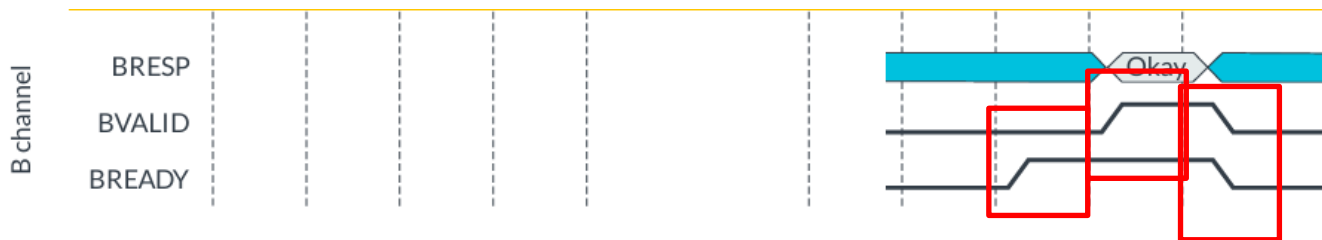
- **The data transfer has the following sequence of event**
 - The slave is waiting for data with WREADY set to high in clock cycle n
 - The master puts the data on the WDATA bus and asserts WVALID in clock cycle $n + 2$
 - The handshake completes on the rising edge of clock cycle $n+3$
 - The slave use Write Response (B) channel to confirm that the write transaction has completed once all WDATA has been received





Channel Transfers and Transactions

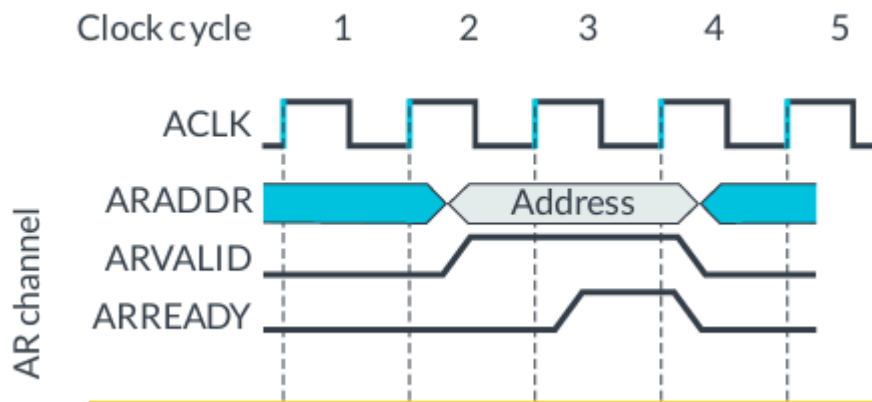
- **The write response has the following sequence of events**
 - The master asserts BREADY
 - The slave drives BRESP to indicate success or failure of the write transaction, and asserts BVALID
 - The handshake completes on the rising edge of clock cycle $n+3$





Channel Transfers and Transactions

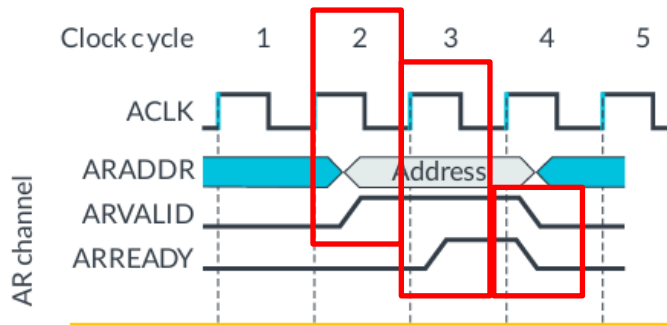
- **Read transaction: single data item**
 - The read transaction involves the following channels
 - Read Address (AR)
 - Read (R)
 - First, there is a handshake on the Read Address (AR) channel





Channel Transfers and Transactions

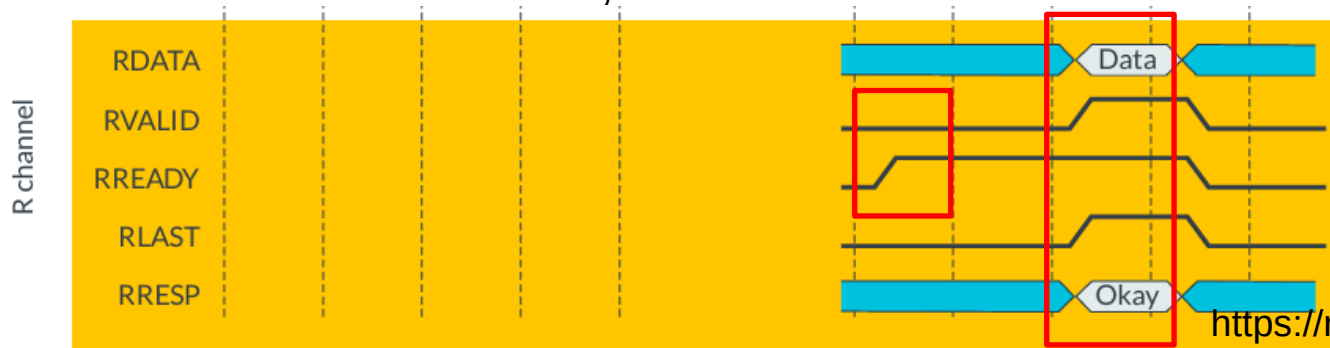
- **The handshake has the following sequence of events**
 - In clock cycle 2, the master communicates the address of the read to the slave on ARADDR and asserts ARVALID
 - In clock cycle 3, the slave asserts ARREADY to indicate that it is ready to receive the address value
 - The handshake completes on the rising edge of clock cycle 4
 - Next, on the Read (R) channel, the slave transfers the data to the master





Channel Transfers and Transactions

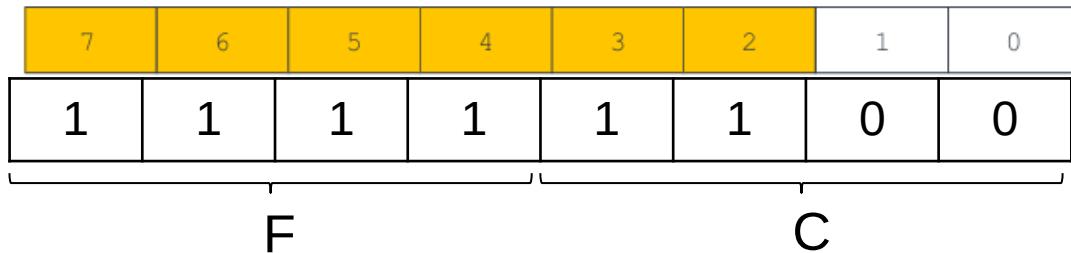
- **The data transfer handshake has the following events**
 - In clock cycle n, the master indicates that it is waiting to receive the data by asserting RREADY
 - The slave retrieves the data and places it on RDATA in clock cycle n+2
 - In a single data transaction, the slave sets the RLAST signal to high
 - The slave uses RRESP to indicate the success or failure of the read transaction to the master, and asserts RVALID





Write Data Strobes

- **What is write data strobe signal?**
 - Used by a master to tell a slave which bytes of the data bus are required
- **When to use write data strobes?**
 - Write data strobes are useful for cache accesses for efficient movement of sparse data arrays
 - Optimize data transfers using unaligned start addresses
 - E.g. Suppose that the valid data are only in the top six significant bytes of the data bus, from byte 7 to byte 2



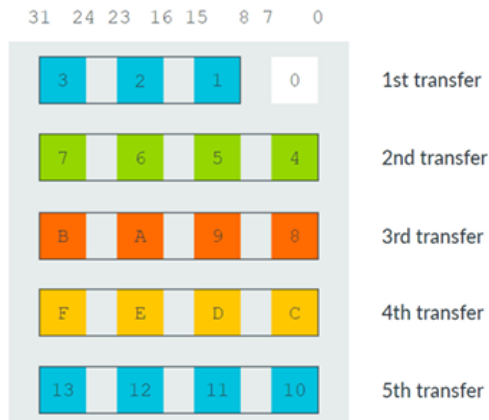
WSTRB = 0xFC



Unaligned Transfer Start Address

- The AXI protocol supports transactions with an **unaligned start address** that only affects the first transfer in a transaction
 - AXI supports unaligned transfer using the strobe signals
 - After the first transfer in a transaction, all other transfers are aligned
 - The example shows a 5-beat 32-bit transfer starting at an unaligned address of 0x01
 - The first transfer contains three bytes
 - All the following transfers in the burst are aligned with the bus width

AxADDR: 0x01
AxSIZE: 32 bits
AxBURST: INCR
AxLEN: 5 transfers





Takeaway Questions

- What are correct descriptions for RISC-V?
 - (A) RISC-V tells us implementation details of the CPU
 - (B) All operations are on internal registers
 - (C) Fused MAC instruction reduces instruction count
- What are benefits of the “Packed-SIMD” ?
 - (A) Reduce the number of LD/ST instructions
 - (B) Improve the L1 cache hit rate
 - (C) Parallel operations on high-precision data



Takeaway Questions

- What is the correct encoding value of WSTRB?

7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---

- (A) 0x81
- (B) 0x3C
- (C) 0xE8