



EFCL Winter School – Track 2

Customizing RISC-V Based Microcontrollers

Lecture 2 – Extending RISC-V cores with
custom instructions

Yvan Tortorella - yvan.tortorella@chips.it

Overview of the course



- > **PULP platform & PULPissimo microcontroller architecture**
- > **Extending RISC-V cores with dedicated custom instructions**
 - > *Lecture*: In-depth RISC-V ISA, microarchitecture of CV32E40X, and CV-XIF specification
 - > *Hands-on*: Extending the RISC-V ISA with new instructions for FIR Filter acceleration
- > **Integrating cooperative HW Processing Engines / HWPEs**
- > **Testing extended PULPissimo on FPGA**

RISC-V: all in one page



Free & Open **RISC-V** Reference Card

Base Integer Instructions: RV32I, RV64I, and RV128I				RV Privileged Instructions				
Category	Name	Format	RV32I Base	RV64I Base	RV128I Base	Name	RV Privilege	
Loads	Load Byte	I	LB	rd, rs1, imm		Atomic R/W	CBRRW	rd, csr, rs1
	Load Halfword	I	LH	rd, rs1, imm		Atomic Read & Set	CBRRS	rd, csr, rs1
	Load Word	I	LW	rd, rs1, imm	LD(0)	Atomic Read & Clear	CBRAC	rd, csr, rs1
	Load Byte Unsigned	I	LBU	rd, rs1, imm		Atomic Read & Set	CBRRS	rd, csr, imm
	Load Halfword Unsigned	I	LHU	rd, rs1, imm	LD(0)U	Atomic Read & Clear	CBRAC	rd, csr, imm
Stores	Store Byte	S	SB	rs1, rs2, imm		Change Level	ENVCALL	
	Store Halfword	S	SH	rs1, rs2, imm		Environment	ENVINFO	
	Store Word	S	SW	rs1, rs2, imm	SD(0)	Trap Redirect	TRAPRED	
Shifts	Shift Left	R	SLL	rd, rs1, rs2	SLL(W)	Redirect to Supervisor	REDTOSUP	
	Shift Left Immediate	I	SLLI	rd, rs1, shamt	SLLI(W)	Redirect Top to Supervisor	REDTTOSUP	
	Shift Right	R	SRL	rd, rs1, rs2	SRL(W)	Interrupt: Wait for Interrupt	INTPT	
Arithmetic	ADD	I	ADD	rd, rs1, rs2	ADD(W)	MMU Supervisor	FPENC	rs1
	ADD Immediate	I	ADDI	rd, rs1, imm				
	SUBTRACT	I	SUB	rd, rs1, rs2				
Logical	XOR	I	XOR	rd, rs1, rs2				
	XOR Immediate	I	XORI	rd, rs1, imm				
	OR	I	OR	rd, rs1, rs2				
Compare	Set <	I	SLT	rd, rs1, rs2				
	Set < Immediate	I	SLTI	rd, rs1, imm				
	Set < Unsigned	I	SLTU	rd, rs1, rs2				
Branches	Branch =	I	BEQ	rs1, rs2, imm				
	Branch ≠	I	BNE	rs1, rs2, imm				
	Branch ≥	I	BGE	rs1, rs2, imm				
Jump & Link	Jump	J	JAL	rd, rs1, imm				
	Jump & Link Register	I	JALR	rd, rs1, imm				
	Synch	I	FENCE					
System	System CRR1	I	SCALL					
	System BRK1	I	SBREAK					
	Counters	I	RDYCYCLER	rd				
Read CYCLE	Read CYCLE upper Half	I	RDYCYCLERH	rd				
	Read TIME	I	RDYTIME	rd				
	Read TIME upper Half	I	RDYTIMEH	rd				
Jump & Link Register	Jump Register	I	JALR	rd, rs1, imm				
	Jump & Link Register	I	JALR	rd, rs1, imm				
	System Env. BRK1	I	SBREAK					

Optional Compressed (16-bit) Instruction Extension: RVC			
Category	Name	Format	RVC
Loads	Load Word SP	CI	CLWSP
	Load Double	CI	CLD
	Load Quad SP	CI	CLQSP
Stores	Store Word	CS	CSW
	Store Word SP	CS	CSWSP
	Store Double SP	CS	CSDSP
Arithmetic	ADD	CR	CRADD
	ADD Immediate	CR	CRADDI
	ADD Word	CR	CRADDW
Shifts	Shift Left Imm	CR	CRSLLI
	Branches	CB	CRBEQ, CRBNE, CRBGE, CRBLT
	Jump	CJ	CRJAL, CRJALR

Optional Floating-Point Instruction Extensions: RV32F, RV64F, RV128F						
Category	Name	Format	RV32F (Single)	RV64F (Double)	RV128F (Quad)	
Multiply/Divide (M)	Multiply	R	MUL	rd, rs1, rs2	MUL(W)	rd, rs1, rs2
	Multiply upper Half	R	MULH	rd, rs1, rs2		
	Multiply upper	R	MULHU	rd, rs1, rs2		
Divide	Divide	R	DIV	rd, rs1, rs2	DIV(W)	rd, rs1, rs2
	Divide Unsigned	R	DIVU	rd, rs1, rs2		
	Remainder	R	REM	rd, rs1, rs2	REM(W)	rd, rs1, rs2
Atomic Extensions (A)	Atomic Load	R	LR	rd, rs1, rs2		
	Atomic Store	R	SR	rd, rs1, rs2		
	Atomic Swap	R	SWAP	rd, rs1, rs2		
Floating-Point Extensions	Single Precision Add	R	FMADD.S	rd, rs1, rs2, rs3		
	Double Precision Add	R	FMADD.D	rd, rs1, rs2, rs3		
	Quad Precision Add	R	FMADD.Q	rd, rs1, rs2, rs3		

RISC-V Calling Convention			
Register	ABI Name	Saver	Description
\$0	zero	---	Hard-wired zero
\$1	ra	Caller	Return address
\$2	sp	Caller	Stack pointer
\$3	gp	---	Global pointer
\$4	tp	---	Thread pointer
\$5-7	ra-2	Caller	Temporaries
\$8	ra+sp	Caller	Saved register/frame pointer
\$9	fp	Caller	Saved registers
\$10-11	a0-a1	Caller	Function arguments/return values
\$12-17	a2-a7	Caller	Function arguments
\$18-27	s0-s11	Caller	Saved registers
\$28-30	t0-t2	Caller	Temporaries
\$31	ra+2	Caller	FP Temporaries
\$32-33	ra+3	Caller	FP saved registers
\$34-37	ra+4	Caller	FP arguments/return values
\$38-39	ra+5	Caller	FP saved registers
\$40-43	ra+6	Caller	FP saved registers

32-bit Instruction Format																			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16				
R	func3	rs2	rs1	func3	rd	opcode										CS	rs1	rd	op
I	imm(16)	rs2	rs1	func3	rd	opcode										CS	rs1	rd	op
SB	imm(16)	rs2	rs1	func3	imm(4)	opcode										CS	rs1	rd	op
U	imm(2)	imm(14)	rs1	func3	rd	opcode										CS	rs1	rd	op
UJ	imm(2)	imm(10)	imm(1)	imm(12)	rd	opcode										CS	rs1	rd	op

RV128I (128-bit) Instruction Format																			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16				
R	func3	rs2	rs1	func3	rd	opcode										CS	rs1	rd	op
I	imm(16)	rs2	rs1	func3	rd	opcode										CS	rs1	rd	op
SB	imm(16)	rs2	rs1	func3	imm(4)	opcode										CS	rs1	rd	op
U	imm(2)	imm(14)	rs1	func3	rd	opcode										CS	rs1	rd	op
UJ	imm(2)	imm(10)	imm(1)	imm(12)	rd	opcode										CS	rs1	rd	op

RISC-V Integer Base (RV32I/RV64I/128I), privileged, and optional compressed extension (RVC). Registers $x1-x31$ and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (50-0). RV64I/128I add 10 instructions for the vd formats. The RVI base of <30 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See riscv.org. (R32I15 revision)

RISC-V calling convention and five optional extensions: 1) multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers $f0-f31$, whose width matches the widest precision, and a floating-point control and status register $fsxr$. Each larger address adds some instructions: 4 for RV32M, 11 for RV32A, and 6 each for RV32F/Q. Using regex notation, (1) means set, so $L(0)(0)$ is both LD and LQ. See riscv.org. (R32I15 revision)

Lecture 2 – Extending RISC-V cores with custom instructions

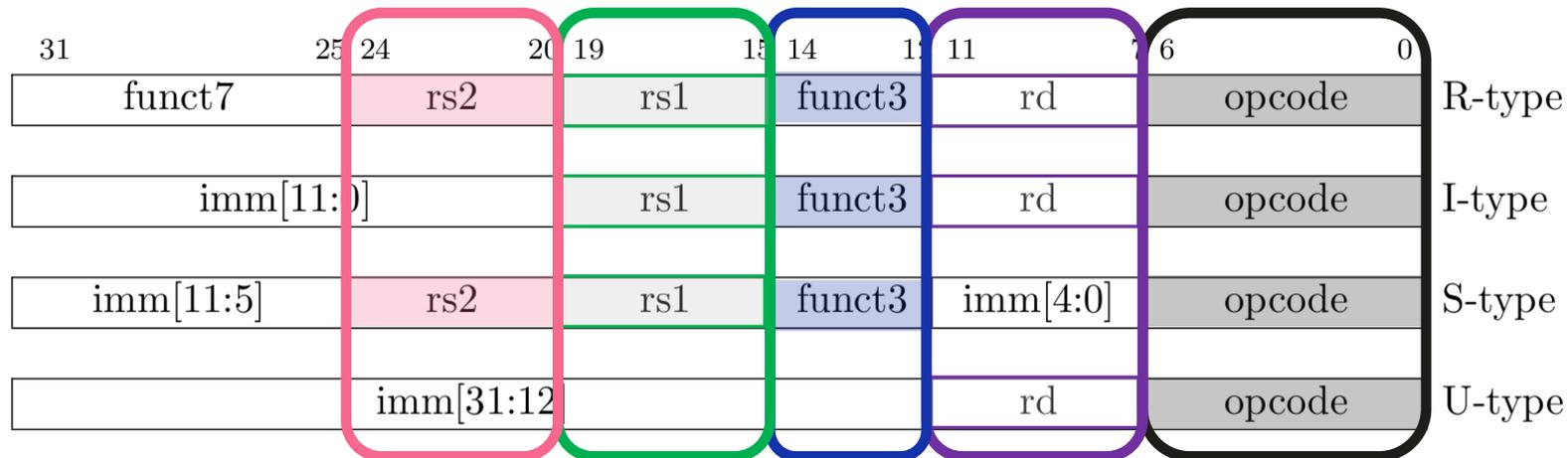
RISC-V Architectural State



- There are 32 registers, each 32 / 64 / 128 bits long
 - Named x0 to x31
 - x0 is hard wired to zero
 - There is a standard 'E' extension that uses only 16 registers (RV32E)
- In addition, one program counter (PC)
 - Byte based addressing, program counter increments by 4/8/16
- For floating point operation 32 additional FP registers
- Additional Control Status Registers (CSRs)
 - Encoding for up to 4'096 registers are reserved. Not all are used.

RISC-V Instructions four basic types

- > **R** register to register operations
- > **I** operations with **i**mmediate/constant values
- > **S/B** operations with two **s**ource registers
- > **U/J** operations with large immediate/constant value



Instruction Encoding: main groups



- Reserved opcodes for standard extensions
- Rest of opcodes free for **custom** implementations
- Standard extensions will be frozen/not change in the future

inst[4:2]	000	001	010	011	100	101	110	111
inst[6:5]								(> 32b)
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

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
-

Constants (Immediates) in Instructions

- > In 32bit instructions, not possible to have 32b constants
 - > Constants are distributed in instructions, and then sign extended
 - > The load **U**pper **I**mmEDIATE (**lui**) instruction to assemble/push constants
- > Instruction types according to immediate encoding

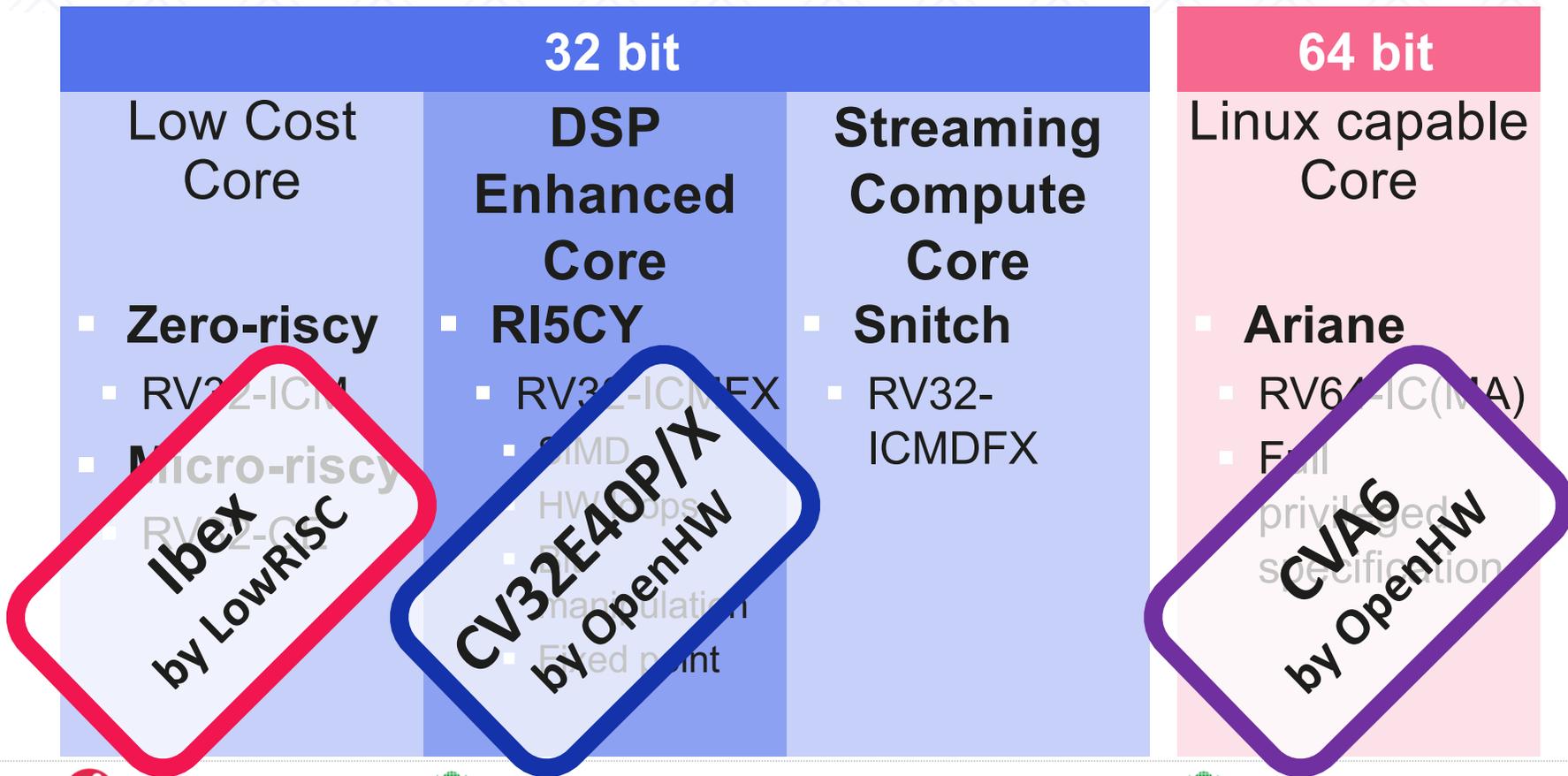
31	30	25	24	21	20	19	15	14	12	11	8	7	6	0	
funct7			rs2			rs1	funct3			rd		opcode		R-type	
imm[11:0]						rs1	funct3			rd		opcode		I-type	
imm[11:5]			rs2			rs1	funct3			imm[4:0]		opcode		S-type	
imm[12]		imm[10:5]			rs2			rs1	funct3			imm[4:1]	imm[11]	opcode	B-type
imm[31:12]									rd			opcode		U-type	
imm[20]		imm[10:1]			imm[11]		imm[19:12]			rd		opcode		J-type	

So, how to build RISC-V cores?



- **RISC-V ISA tells you the function**
 - You know which instructions are supported
 - How they are encoded
 - What they are supposed to do
 - **It does **not** tell you any implementation details**
 - Pipeline stages, memory hierarchy, computation units, 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**
 - Determine which standard extensions are supported, how
 - Choose a micro-architecture that fits performance requirements
-

RISC-V cores developed in the PULP Platform



RI5CY / CV32E40P / CV32E40X



- Tuned for energy efficiency
 - Not necessarily lowest power
- Make use of *custom extensions* in RI5CY & CV32E40P
 - The Xpulp extensions enhance the capabilities
 - Several Xpulp extensions in discussions for ratification
- CV32E40X is developed by OpenHWGroup
 - Started as a fork of CV32E40P
 - X stands for “extensible”: CV32E40X introduces the CV-XIF (eXtension InterFace)
 - CV-XIF is not yet a ratified standard: we consider the flavour supported in CV32E40X!

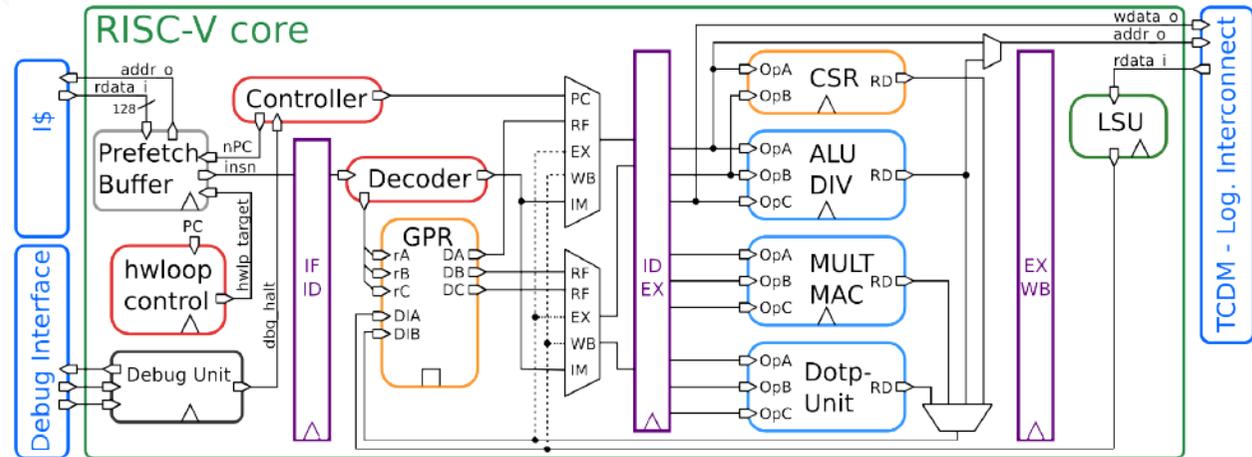


OPENHW^{GROUP}[®]
— PROVEN PROCESSOR IP —

RI5CY/CV32E40P: Our 32-bit workhorse



- > 4-stage pipeline
 - > 41 kGE
 - > Coremark/MHz 3.19
- > Includes Xpulp extensions
 - > SIMD
 - > Fixed point
 - > Bit manipulations
 - > HW loops

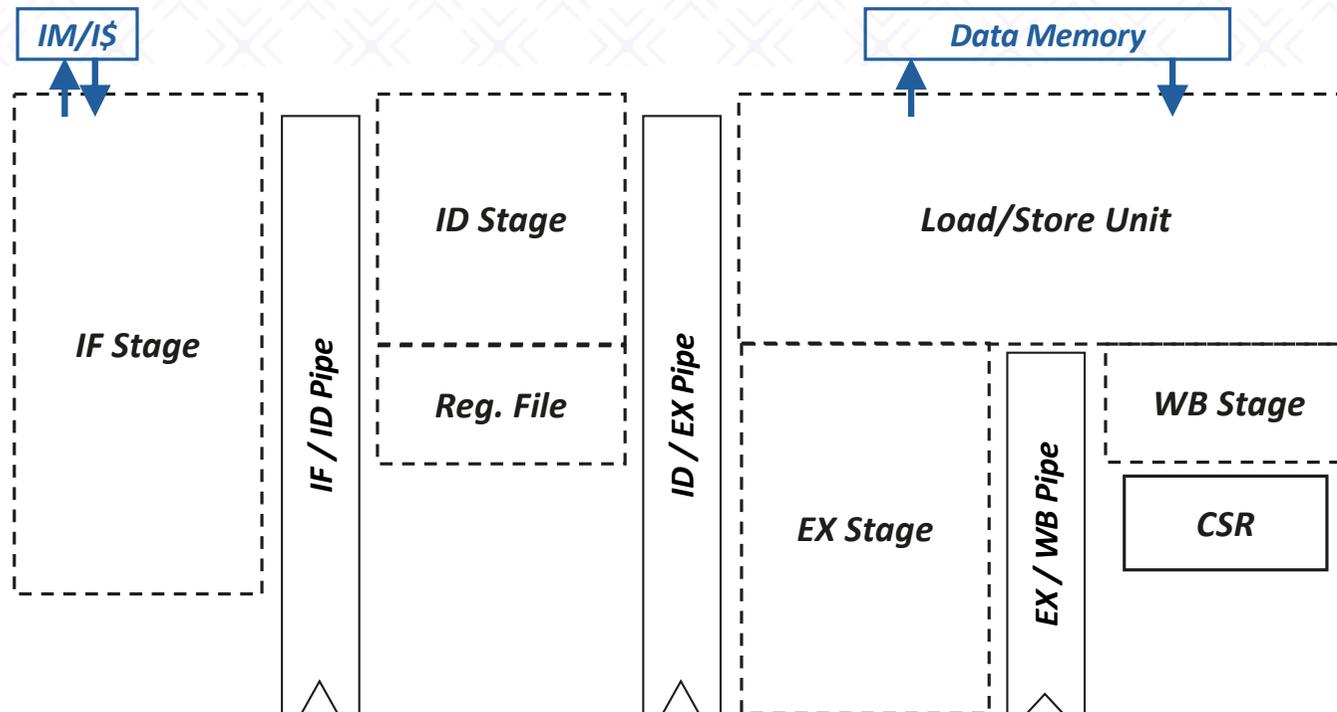


■ Different Options:

- **FPU**: IEEE 754 single precision
 - Including hardware support for FDIV, FSQRT, FMAC, FMUL
- **Privilege support**:
 - Supports privilege mode **M** and **U**

M. Gautschi *et al.*, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700-2713, Oct. 2017.

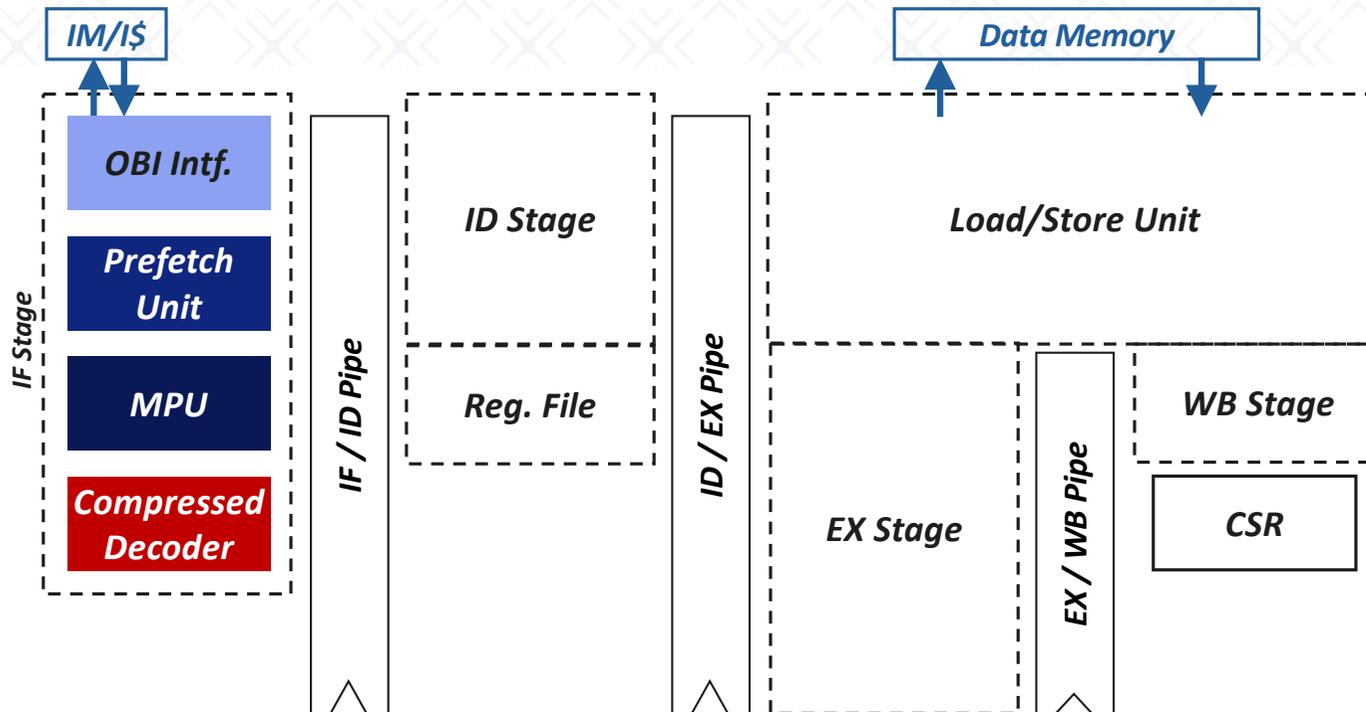
CV32E40X architecture



4-stage pipeline

- > IF/ID/EX/WB
- > EX includes EX+MEM from “classic” Hennessy Patterson 5-stage architecture
- > The 4 stages used a Valid/Ready handshake on the pipeline registers
- > The RTL description follows quite accurately the organization in this diagram
- > A fork of CV32E40P

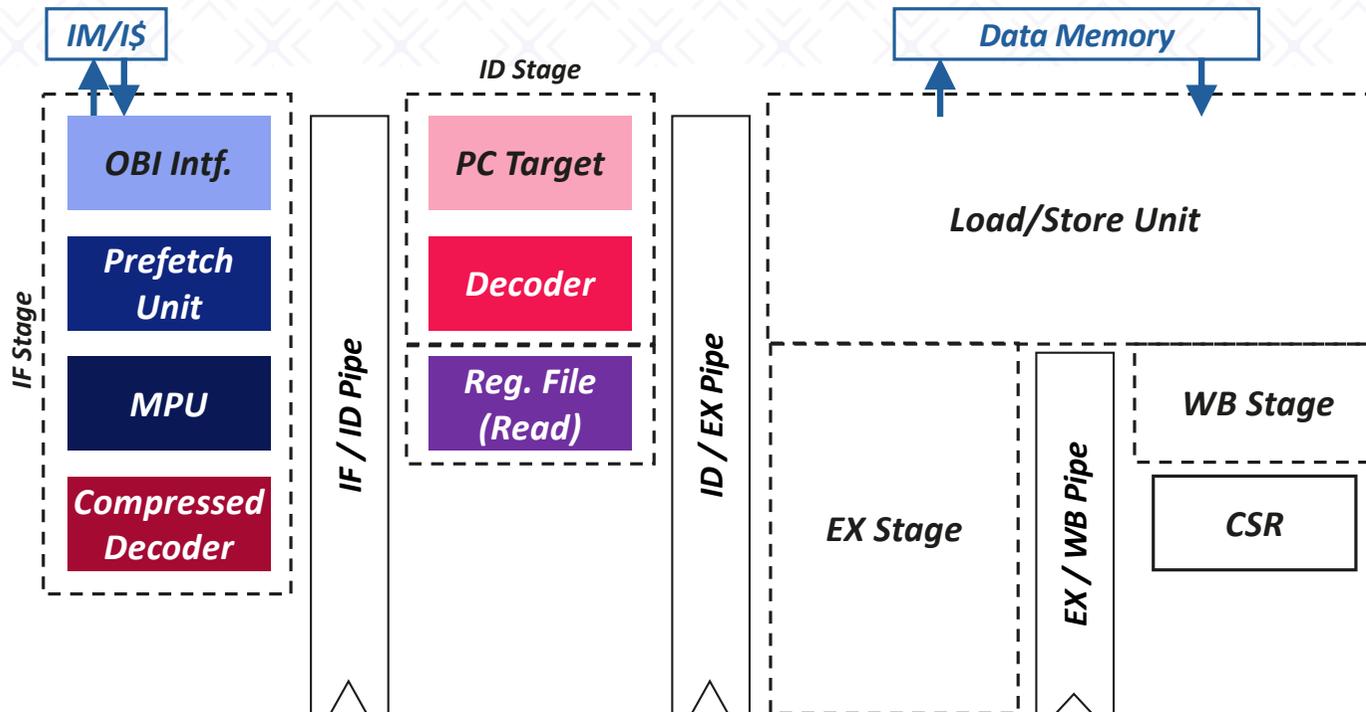
CV32E40X architecture



Instruction Fetch

- > Includes a prefetcher unit and a decoder for compressed instructions (16-bit)
- > Uses the OBI interface, but without support for all signals
 - > in practice, the TCDM interface discussed yesterday

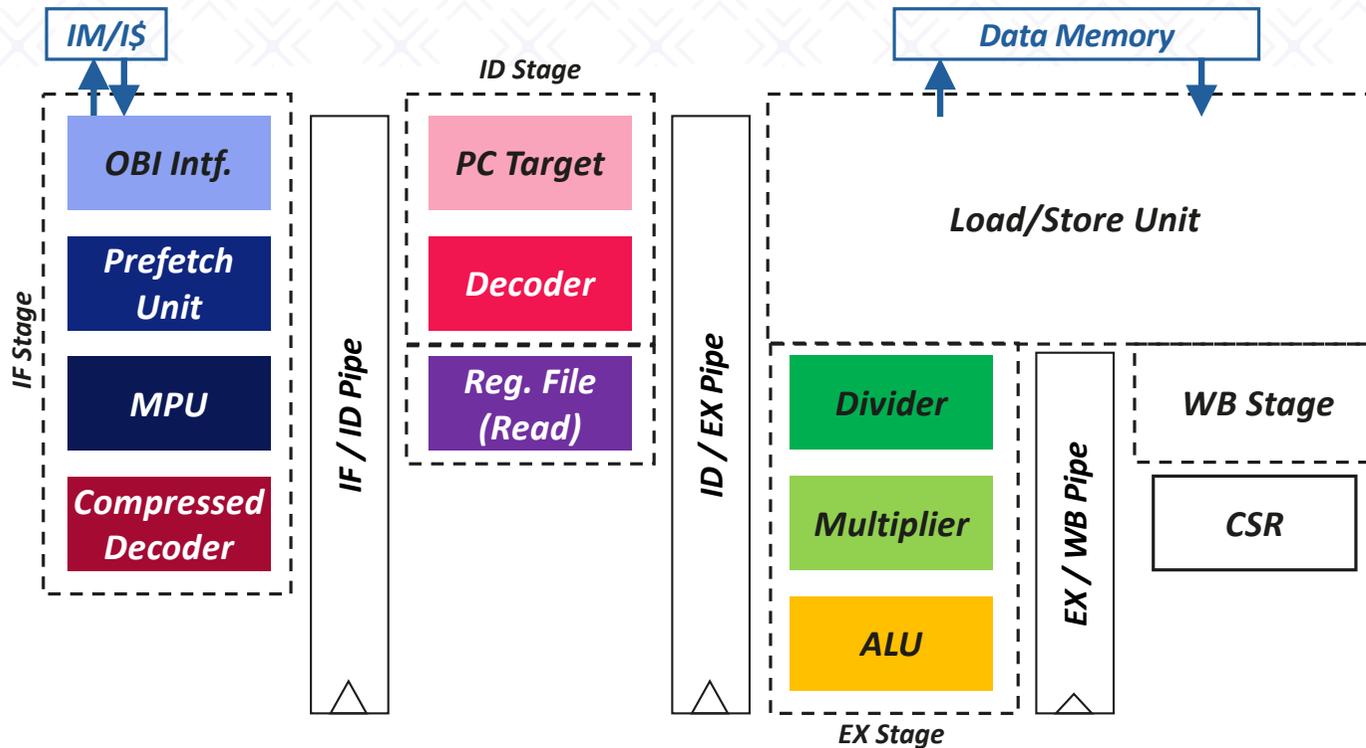
CV32E40X architecture



Instruction Decode

- > Includes a Decoder for decompressed instructions (32-bit)
- > The Register File (for reading) is in the ID Stage!
 - > Configurable for 2 read ports (RV32IMC) or 3 (for X extensions)

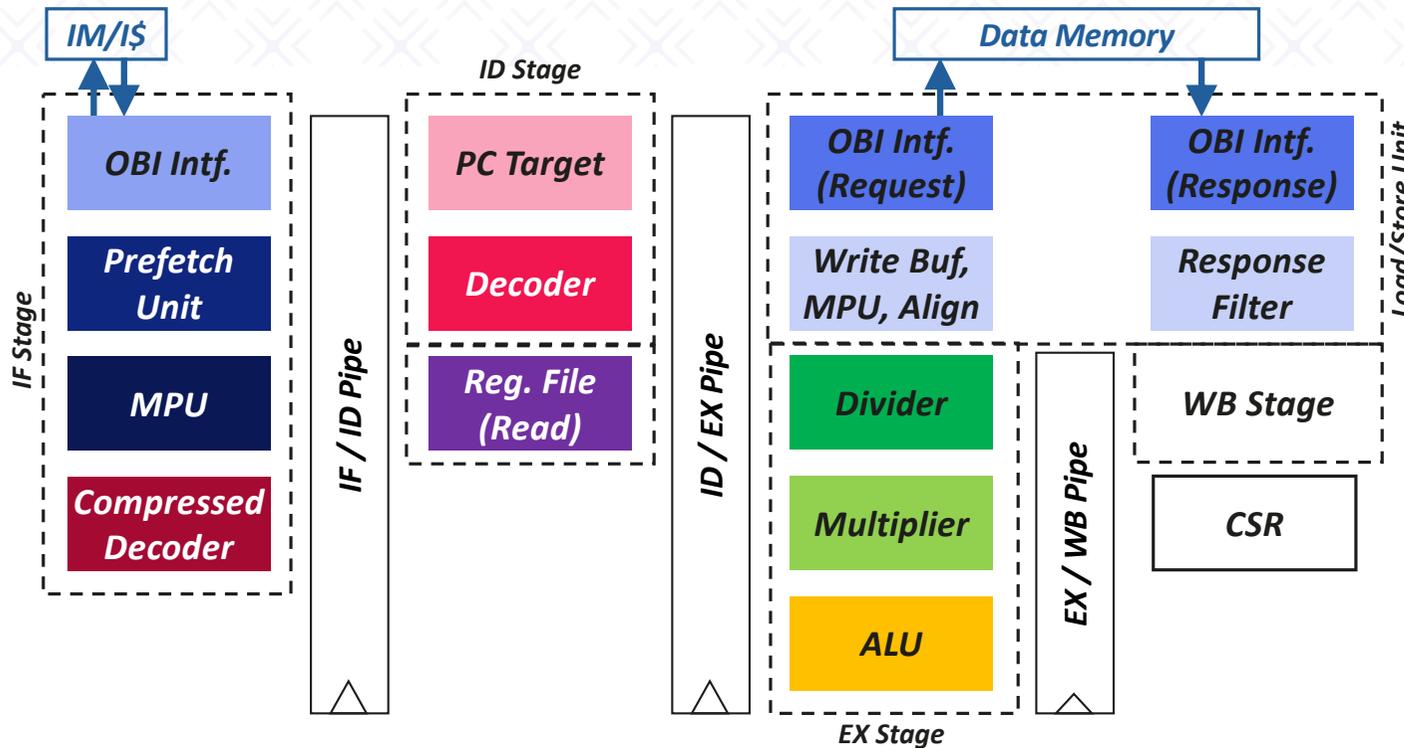
CV32E40X architecture



Execute (arithmetic)

- > The Execute stage contains all arithmetic units
- > 1-cycle ALU
- > 1-cycle Multiplier
- > Multi-cycle (3-35) Divider

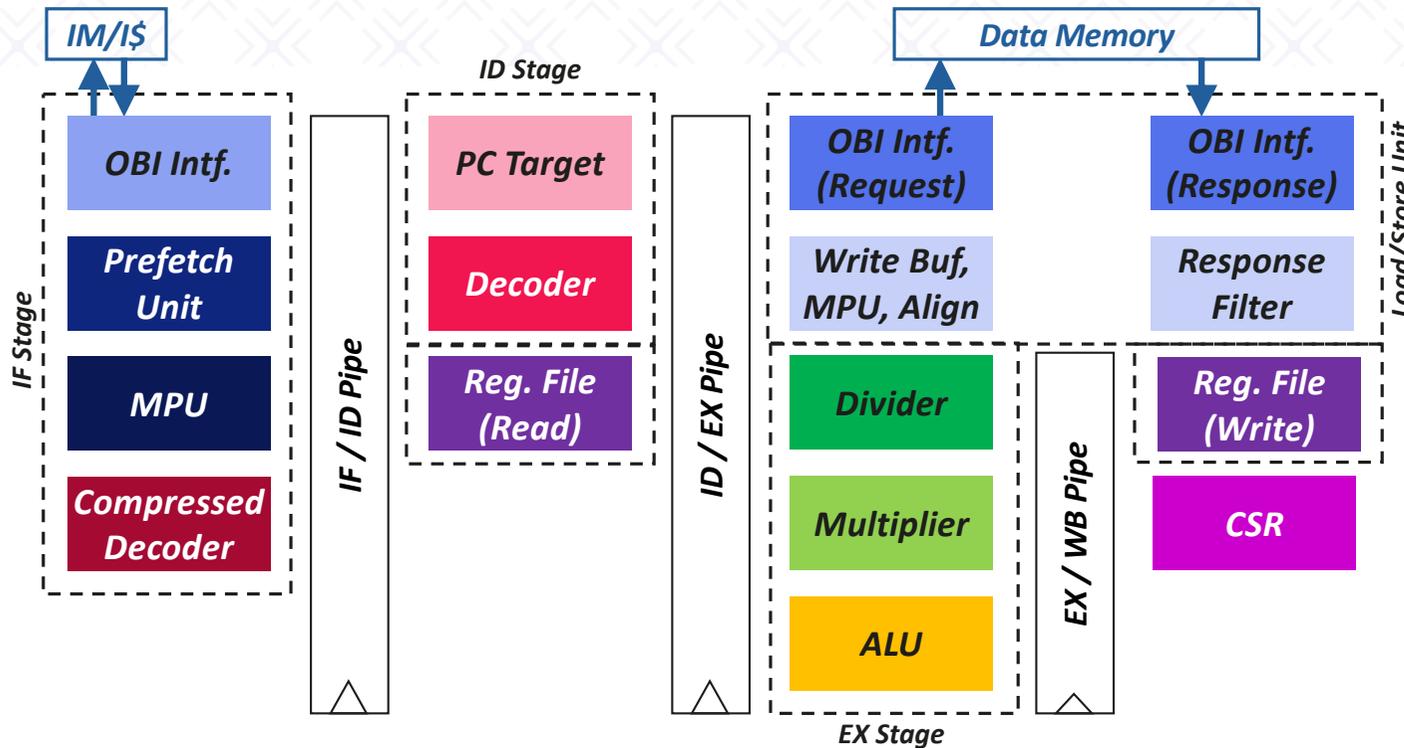
CV32E40X architecture



Execute (memory)

- > Memory requests are performed in the **EX** stage as well
- > Also uses “restricted OBI”
- > Memory responses arrive in the **WB** stage (assuming no conflict, else -> stall)

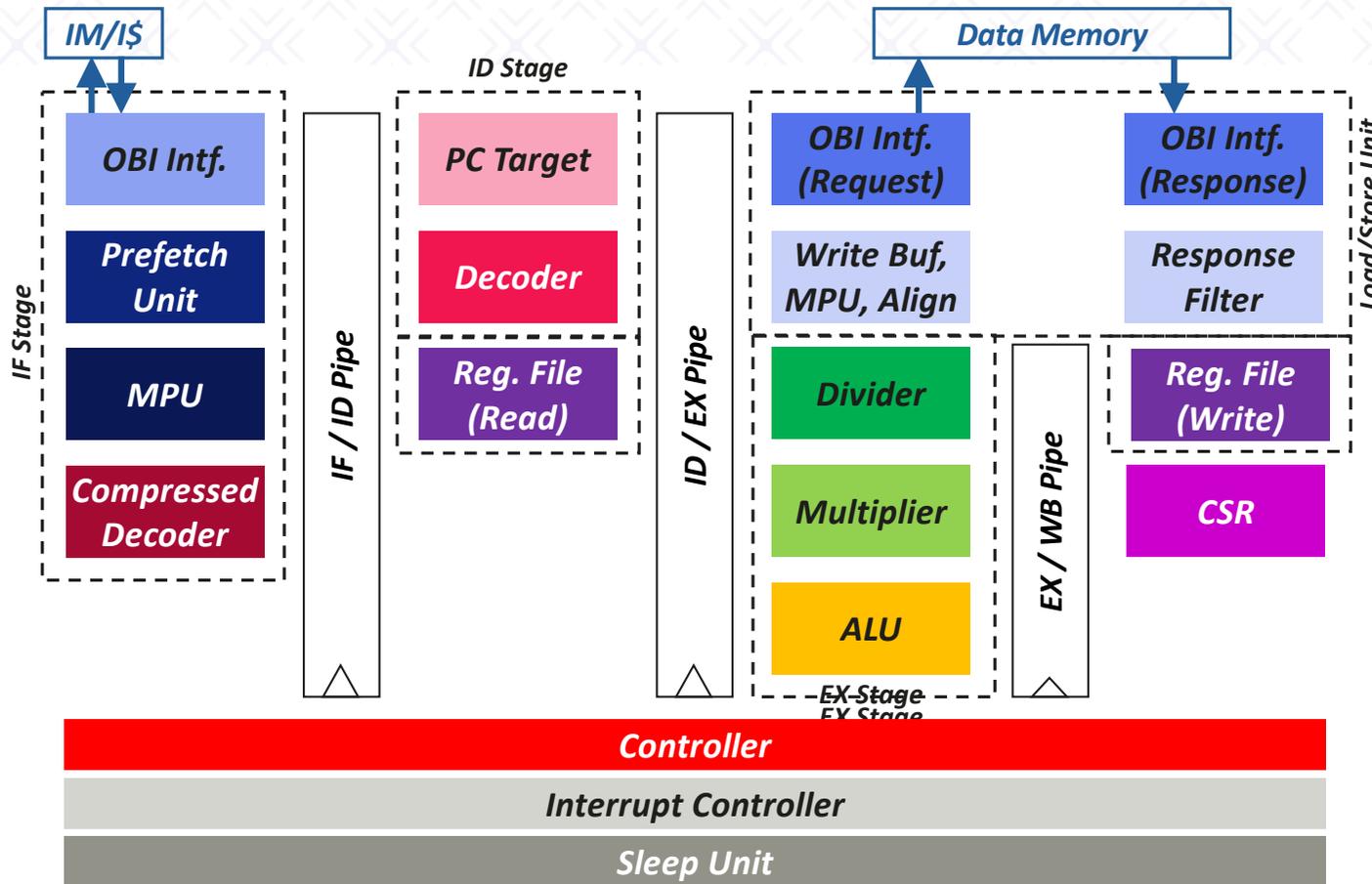
CV32E40X architecture



Write-back

- > Register file writes are performed in **WB** stage
- > Same for Control & Status Registers

CV32E40X architecture



Outside pipeline

- > A controller is used for instruction forwarding, scoreboarding
- > Integrated interrupt controller
- > Sleep unit controls clock-gating in case of *wfi*, *wfe* instructions

RISC-V has space for custom instructions (X)



- There is a reserved decoding space for custom instructions
 - Allows **everyone** to add new instructions to the core
 - The address decoding space is **reserved**, it will not be used by future extensions
 - Implementations supporting custom instructions will be compatible with standard ISA
 - Code compiled for standard RISC-V will run without issues
 - The user has to provide support to take advantage of the additional instructions
 - Compiler that generates code for the custom instructions
 - We use a lot this degree of freedom
 - Great tool for exploring
 - The goal is to help ratify these extensions as standards through working group
-

Is it fast enough?



Take as example our FIR inner loop:

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

ANSI C

per iteration:

1 MAC (multiply-accumulate) operation

7 instructions

efficiency = 1/7 = 0.14 ☹

```
Lstart:  
lh t2, 0x0(t1)  
lh a4, 0x0(t3)  
mul a4, a4, t2  
add t0, t0, a4  
addi t3, t3, 0x2  
addi t1, t1, 0x2  
bne t3, a7, Lstart
```

RV32IM ASM

Is it fast enough?

Take as example our FIR inner loop:

```
for(int j=0; j<coeff_len; j++) {  
    int arr_int = arr[i+j];  
    int 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

efficiency = 1/7 = 0.14 ☹

```
Lstart:  
lh t2, 0x0(t1)  
lh a4, 0x0(t3)  
mul a4, a4, t2  
add t0, t0, a4  
addi t3, t3, 0x2  
addi t1, t1, 0x2  
bne t3, a7, Lstart
```

RV32IM ASM

Make it a bit faster: MAC instruction



Take as example our FIR inner loop:

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

ANSI C

per iteration:

1 MAC (multiply-accumulate) operation
6 instructions

efficiency = $1/6 = 0.17$ ☺

```
Lstart:  
lh t2, 0x0(t1)  
lh a4, 0x0(t3)  
p.mac t0, a4, t2  
addi t3, t3, 0x2  
addi t1, t1, 0x2  
bne t3, a7, Lstart
```

RV32IM ASM

Requires three ports in the register file!

Make it a bit faster, again: pointer post-increment



Take as example our FIR inner loop:

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

ANSI C

per iteration:

1 MAC (multiply-accumulate) operation

4 instructions

efficiency = 1/4 = 0.25 ☺

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

RV32IM ASM

Make it a bit faster, again²: HW loops



Take as example our FIR inner loop:

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

ANSI C

per iteration:

1 MAC (multiply-accumulate) operation

3 instructions

efficiency = 1/3 = 0.33 ☺☺

```
lp.setup t3, a7, Lend  
Lstart:  
p.lh t2, 0x0(t1!)  
p.lh a4, 0x0(t3!)  
Lend:  
p.mac t0, a4, t2
```

RV32IM ASM

How to get even more performance?

Look at the instruction trace

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

ANSI C

```
lp.setup t3,a7,Lend  
Lstart:  
p.lh t2, 0x0(t1!)  
p.lh a4, 0x0(t3!)  
Lend:  
p.mac t0, a4, t2
```

RV32IMXpulp ASM

```
lp.setup t3,a7,Lend  
p.lh t2,0x0(t1!)  
p.lh a4,0x0(t3!)  
p.mac t0,a4,t2  
p.lh t2,0x0(t1!)  
p.lh a4,0x0(t3!)  
p.mac t0,a4,t2  
p.lh t2,0x0(t1!)  
p.lh a4,0x0(t3!)  
p.mac t0,a4,t2  
// ...  
p.lh t2,0x0(t1!)  
p.lh a4,0x0(t3!)  
p.mac t0,a4,t2
```

coeff_len times

How to get even more performance?

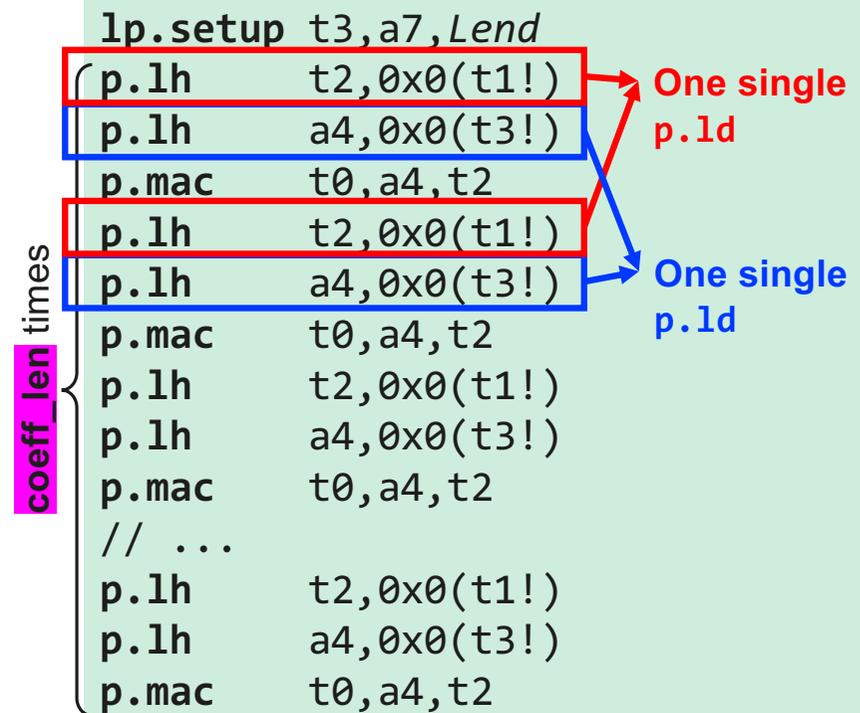
Look at the instruction trace

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

ANSI C

```
lp.setup t3,a7,Lend  
Lstart:  
p.lh t2, 0x0(t1!)  
p.lh a4, 0x0(t3!)  
Lend:  
p.mac t0, a4, t2
```

RV32IMXpulp ASM



How to get even more performance?

Look at the instruction trace

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

ANSI C

```
lp.setup t3,a7,Lend  
Lstart:  
p.lh t2, 0x0(t1!)  
p.lh a4, 0x0(t3!)  
Lend:  
p.mac t0, a4, t2
```

RV32IMXpulp ASM

```
lp.setup t3,a7,Lend  
p.lh t2,0x0(t1!)  
p.lh a4,0x0(t3!)  
p.mac t0,a4,t2  
p.lh t2,0x0(t1!)  
p.lh a4,0x0(t3!)  
p.mac t0,a4,t2  
p.lh t2,0x0(t1!)  
p.lh a4,0x0(t3!)  
p.mac t0,a4,t2  
// ...  
p.lh t2,0x0(t1!)  
p.lh a4,0x0(t3!)  
p.mac t0,a4,t2
```

coeff_len times

One single pv.sdotp

Instruction trace

Loop stripmining and vectorization



Single Instruction Multiple Data (SIMD) Vectorization → do all MACs in one instruction!

coeff_len/2 times

```
p.ld    t2,0(t1!)
p.ld    a4,0(t3!)
pv.sdotp t0,a4,t2
// ...
```

coeff_len%2 times

```
p.lh   t2,0(t1!)
p.lbu  a4,0(t3!)
p.mac  t0,a4,t2
```

Instruction trace (RVIM32Xpulp2)

```
for(j=0; j<coeff_len/2; j++) {
    v2u coef_v = *(v4u *) coeff++; // coeff is an int*
    v2u arr_v  = *(v4u *) arr++;   // arr  is an int*
    sum = sum + (int) __builtin_pulp_sdotsp4(coef_v, arr_v, sum);
}
vectorized
```

```
for(j=0; j<coeff_len%2; j++) {
    sum = sum + (int)(coeff[j]*arr[j]);
}
leftover
```

ANSI C

per iteration (unrolled loop):

2 MAC (multiply-accumulate) operations

3 instructions

efficiency = $2/3 = 0.67$ 🍷 (vectorized loop is 4.8x better than original scalar code!)

Can we still improve more?



Remember: this is part of a larger piece of code

```
for (int i=0; i<arr_len; i++) {  
    int32_t sum=0;  
    for (int j=0; j<coeff_len; j++) {  
        sum += arr[i+j] * coeff[j];  
    }  
    // shift accumulator and fit it into 16 bits  
    output[i] = (int16_t) ((sum >> right_shift) & 0x0000ffff);  
}
```

ANSI C

Can we still improve more?

Remember: this is part of a larger piece of code

```
for (int i=0; i<arr_len; i++) {  
    int32_t sum=0;  
    for (int j=0; j<coeff_len; j++) {  
        sum += arr[i+j] * coeff[j];  
    }  
    // shift accumulator and fit it into 16 bits  
    output[i] = (int16_t) ((sum >> right_shift) & 0x0000ffff);  
}
```

ANSI C

The same `coeff[j]` get used for many iterations of the outermost loop

- `i=0, 1, ...`
 - Can be *reused!*
-

Further restructuring with loop unrolling (factor=2)



```
p.ld    t2,0(t1!)
p.ld    a4,0(t3!)
p.ld    a5,0(t4!)
pv.sdotp t0,a4,t2
pv.sdotp t5,a5,t2
// ...
```

```
for (int i=0; i<arr_len; i+=2) {
    int32_t sum0=0, sum1=0;
    for (int j=0; j<coeff_len; j++) {
        sum0 += arr[i+0+j] * coeff[j];
        sum1 += arr[i+1+j] * coeff[j];
    }
    // shift accumulator and fit it into 16 bits
    output0[i] = (int16_t) ((sum0 >> right_shift) & 0xffff);
    output1[i] = (int16_t) ((sum1 >> right_shift) & 0xffff);
}
```

ANSI C

per iteration (unrolled loop):

4 MAC (multiply-accumulate) operations

5 instructions

efficiency = $4/5 = 0.8$ 🍷 (vectorized loop is 5.7x better than original scalar code!)

Further restructuring with loop unrolling (factor=4)



```
p.ld    t2,0(t1!)
p.ld    a4,0(t3!)
p.ld    a5,0(t4!)
p.ld    a6,0(t6!)
p.ld    a7,0(t7!)
pv.sdotp t0,a4,t2
pv.sdotp t5,a5,t2
pv.sdotp t8,a6,t2
pv.sdotp t9,a7,t2
// ...
```

per iteration (unrolled loop):
8 MAC (multiply-accumulate) operations
9 instructions

```
for (int i=0; i<arr_len; i+=4) {
    int32_t sum0=0, sum1=0, sum2=0, sum3=0;
    for (int j=0; j<coeff_len; j++) {
        sum0 += arr[i+0+j] * coeff[j];
        sum1 += arr[i+1+j] * coeff[j];
        sum2 += arr[i+2+j] * coeff[j];
        sum3 += arr[i+3+j] * coeff[j];
    }
    // shift accumulator and fit it into 16 bits
    output0[i] = (int16_t) ((sum0 >> right_shift) & 0xffff);
    output1[i] = (int16_t) ((sum1 >> right_shift) & 0xffff);
    output2[i] = (int16_t) ((sum2 >> right_shift) & 0xffff);
    output3[i] = (int16_t) ((sum3 >> right_shift) & 0xffff);
}
```

ANSI C

efficiency = $8/9 = 0.89$ 🍷 (vectorized loop is 6.3x better than original scalar code!)

Further restructuring with loop unrolling (factor=4)



```
p.ld    t2,0(t1!)
p.ld    a4,0(t3!)
p.ld    a5,0(t4!)
p.ld    a6,0(t6!)
p.ld    a7,0(t7!)
pv.sdotp t0,a4,t2
pv.sdotp t5,a5,t2
pv.sdotp t8,a6,t2
pv.sdotp t9,a7,t2
// ...
```

per iteration (unrolled loop):
8 MAC (multiply-accumulate) operations
9 instructions

```
for (int i=0; i<arr_len; i+=4) {
    int32_t sum0=0, sum1=0, sum2=0, sum3=0;
    for (int j=0; j<coeff_len; j++) {
        sum0 += arr[i+0+j] * coeff[j];
```

Unfortunately, these registers do not exist...
We are limited by the internal register file! → spilling to stack

```
    }
    // shift accumulator and fit it into 16 bits
    output0[i] = (int16_t) ((sum0 >> right_shift) & 0xffff);
    output1[i] = (int16_t) ((sum1 >> right_shift) & 0xffff);
    output2[i] = (int16_t) ((sum2 >> right_shift) & 0xffff);
    output3[i] = (int16_t) ((sum3 >> right_shift) & 0xffff);
}
```

ANSI C

efficiency = $8/9 = 0.89$ 🍷 (vectorized loop is 6.3x better than original scalar code!)... **but only in theory**

A bit more flexibility for our FIR application?



Would be nice to be able to add more flexible extensions suited to our tasks

- In our FIR example Xpulp would work ok for vectorization, but we do not have enough register file space to really exploit unrolling

Add a separate register file in the core?

- Scary...
- Requires changing a lot of things in the core!

Is there an alternative?



CORE-V eXtension InterFace (XIF)



It often makes sense to integrate extensions “deep” within a core

- See the past few slides for Xpulp extensions!

However, changing “deeply” the core RTL is extremely time-consuming

- A simpler approach is to provide standard “hooks” to couple co-processors implementing an ISA extension
- CV-XIF provides low latency (tightly integrated) read and write access to the CV32E40X register file.
- CV32E40X offloads “invalid” instructions to CV-XIF co-processors

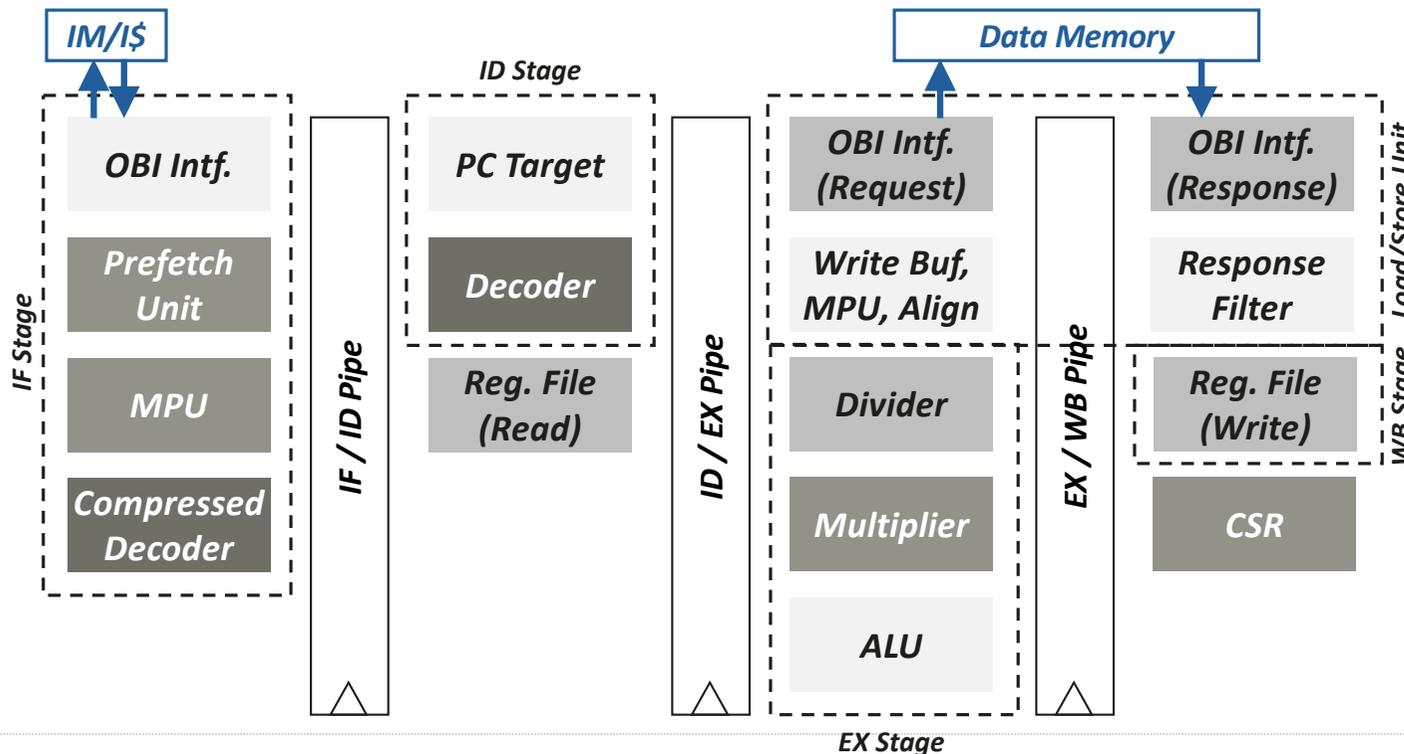
The eXtension interface enables extension of CV32E40X with:

- Custom ALU type instructions.
- Custom load/store type instructions.
- Custom CSRs and related instructions.
- Control-Transfer instructions (e.g. branches/jumps) are not supported via CV-XIF



CV-XIF in CV32E40X

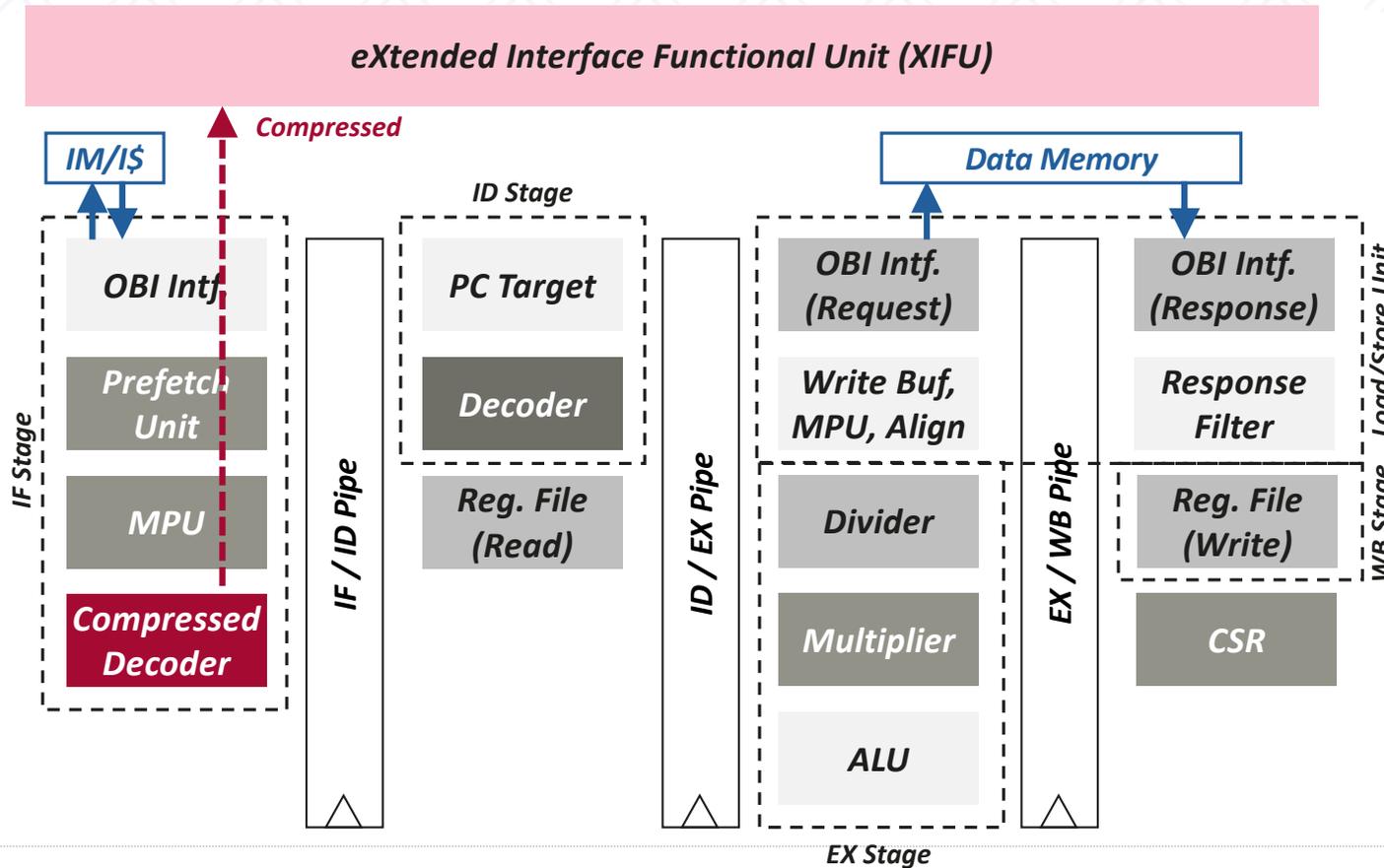
eXtended Interface Functional Unit (XIFU)



How to couple a coprocessor with CV32E40X?

- > Let us call it a XIF Functional Unit (**XIFU**)
- > The XIFU can be organized arbitrarily internally, but the interface is based on the same organization of CV32E40X
- > XIF uses bundles of signals (interfaces) btw core and co-processor
- > All with valid/ready HS

CV-XIF in CV32E40X



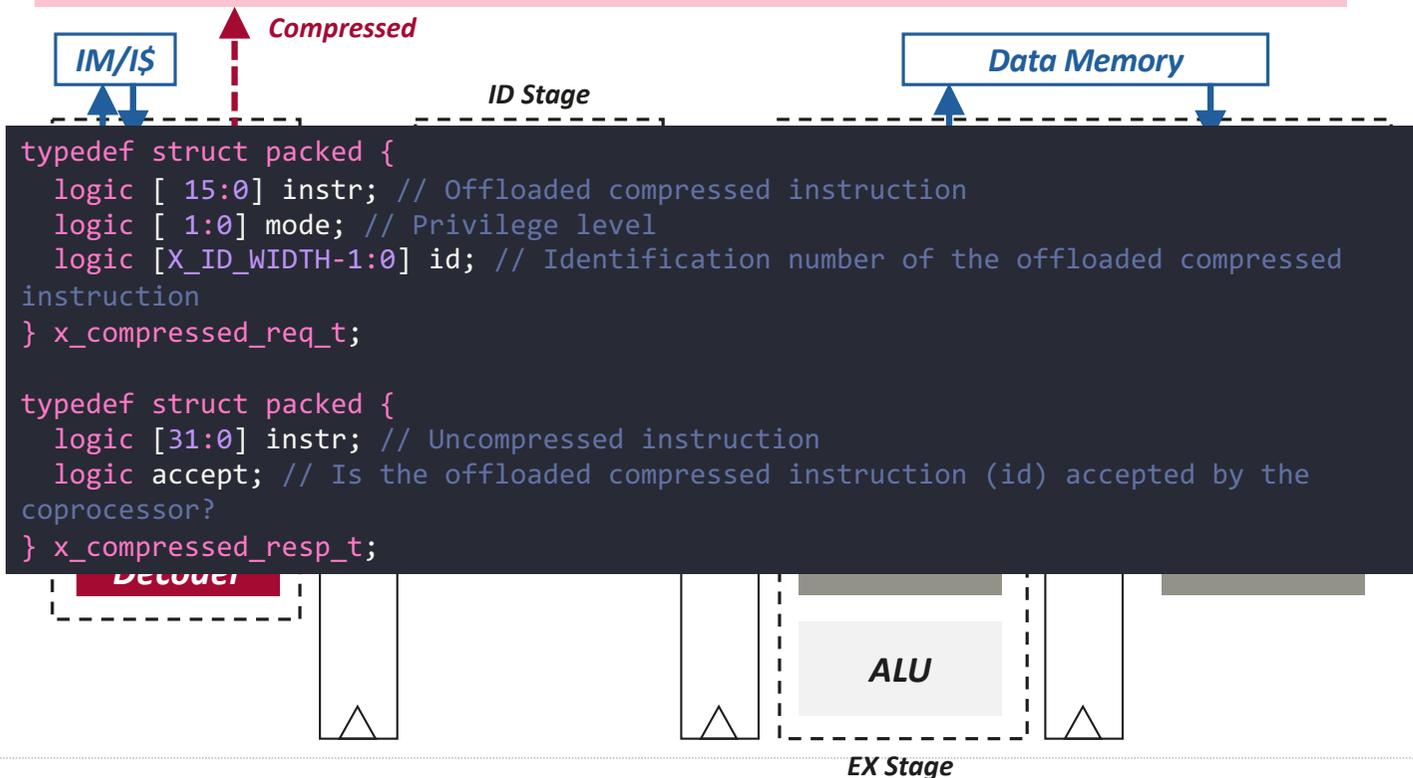
Compressed interf.

- > Basically correspondent to CV32E40X IF
- > Illegal compressed instructions (lsb != 2'b11) get offloaded to XIFU for decompression
- > Within the same cycle, XIFU responds with a decompressed instruction (or does not accept the instruction)
- > Currently not implemented in CV32E40X

CV-XIF in CV32E40X



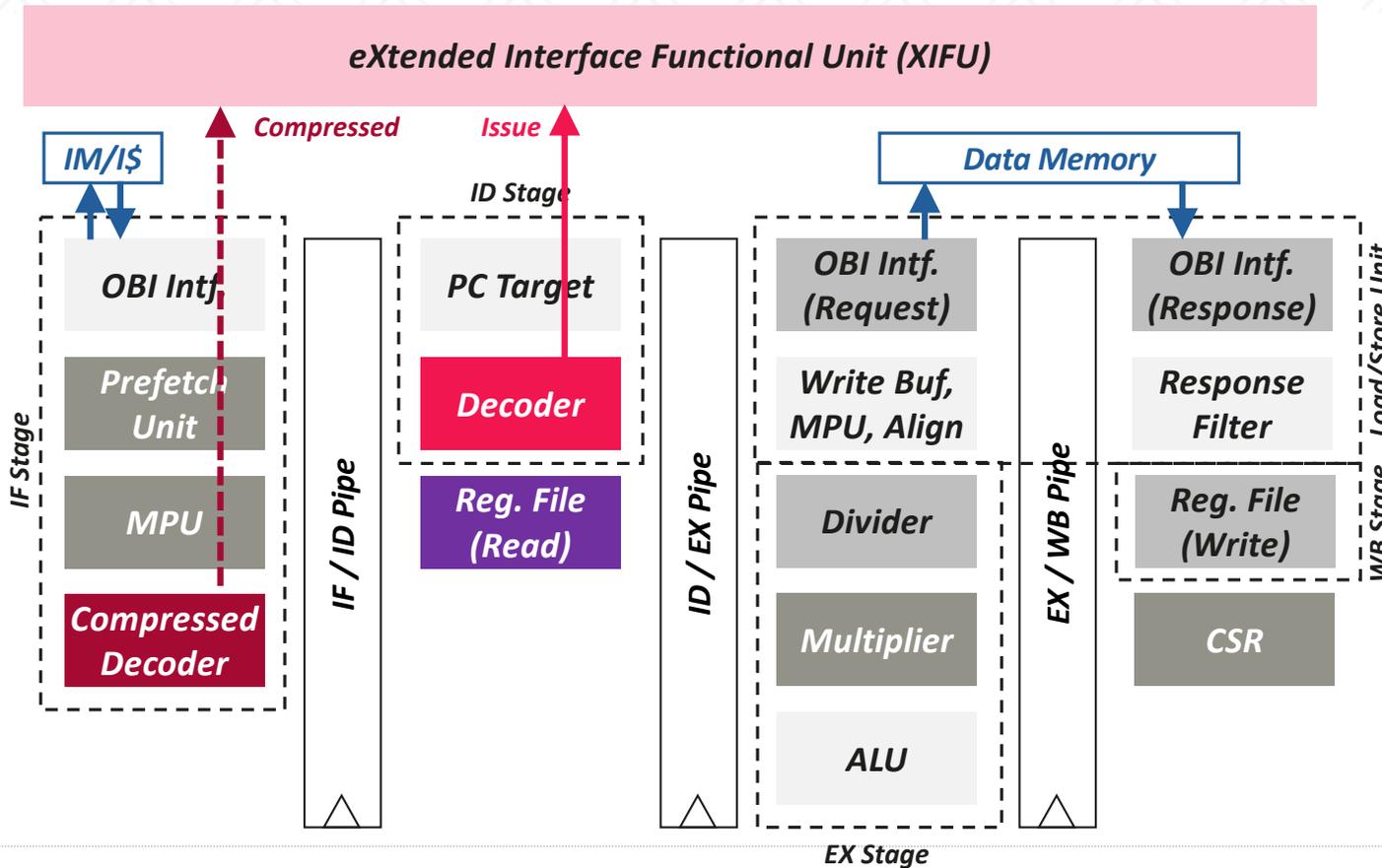
eXtended Interface Functional Unit (XIFU)



Compressed interf.

- > Basically correspondent to CV32E40X IF
- > Illegal compressed instructions (lsb != 2'b11) get offloaded to XIFU for decompression
- > Within the same cycle, XIFU responds with a decompressed instruction (or does not accept the instruction)
- > Currently not implemented in CV32E40X

CV-XIF in CV32E40X



Issue interf.

- > Basically correspondent to CV32E40X ID
- > Illegal instructions get offloaded to XIFU
- > Issue also contains register file operands from reg. file
- > Response accepts/denies instruction and provides info to CV32 controller

CV-XIF in CV32E40X



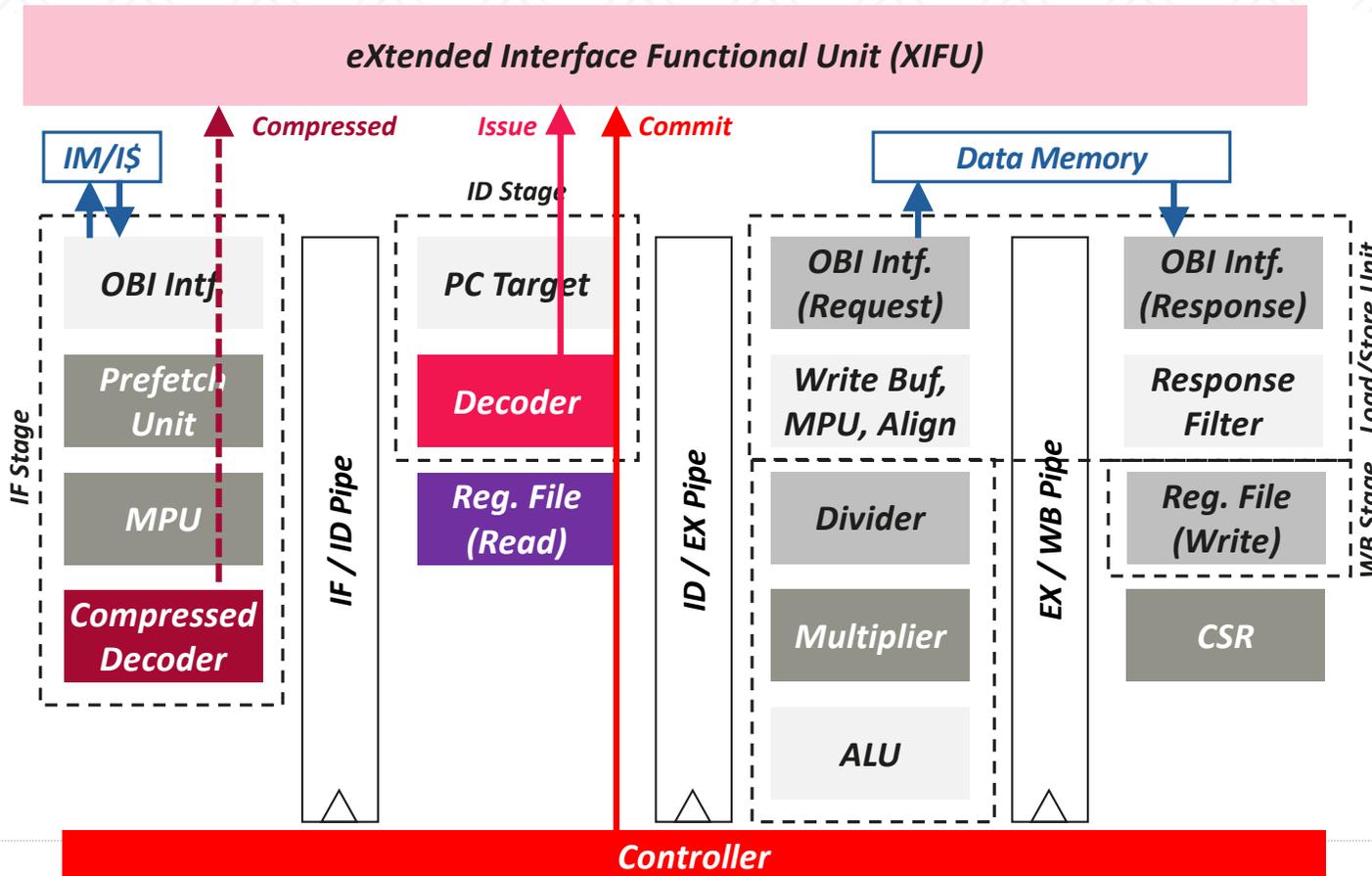
```
typedef struct packed {
    logic [ 31:0] instr; // Offloaded instruction
    logic [ 1:0] mode; // Privilege level
    logic [X_ID_WIDTH-1:0] id; // Identification of the offloaded instruction
    logic [X_NUM_RS -1:0][X_RFR_WIDTH-1:0] rs; // Register file source operands for the
offloaded instruction
    logic [X_NUM_RS -1:0] rs_valid; // Validity of the register file source operand(s)
    logic [ 5:0] ecs; // Extension Context Status ({mstatus.xs, mstatus.fs,
mstatus.vs})
    logic ecs_valid; // Validity of the Extension Context Status
} x_issue_req_t;

typedef struct packed {
    logic accept; // Is the offloaded instruction (id) accepted by the coprocessor?
    logic writeback; // Will the coprocessor perform a writeback in the core to rd?
    logic dualwrite; // Will the coprocessor perform a dual writeback in the core to rd
and rd+1?
    logic [2:0] dualread; // Will the coprocessor require dual reads from rs1\rs2\rs3
and rs1+1\rs2+1\rs3+1?
    logic loadstore; // Is the offloaded instruction a load/store instruction?
    logic ecswrite ; // Will the coprocessor write the Extension Context Status in
mstatus?
    logic exc; // Can the offloaded instruction possibly cause a synchronous exception
in the coprocessor itself?
} x_issue_resp_t;
```

Issue interf.

- > Basically correspondent to CV32E40X ID
- > Illegal instructions get offloaded to XIFU
- > Issue also contains register file operands from reg. file
- > Response accepts/denies instruction and provides info to CV32 controller

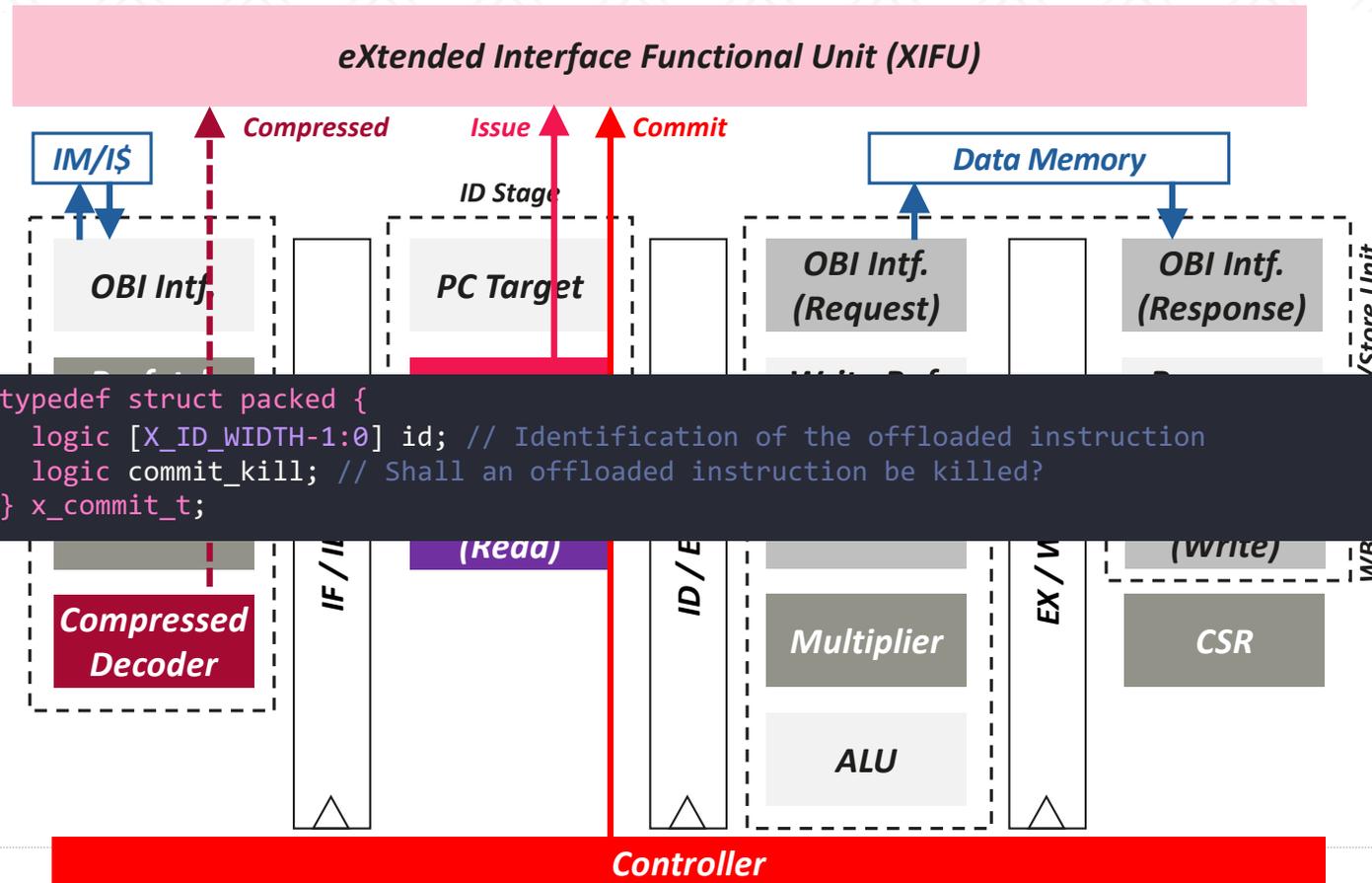
CV-XIF in CV32E40X



Commit interf.

- > Basically correspondent to CV32E40X **Controller**
- > Commits or Kills instructions (e.g., in case of an exception on earlier ones)
- > XIFU does not respond, only complies!

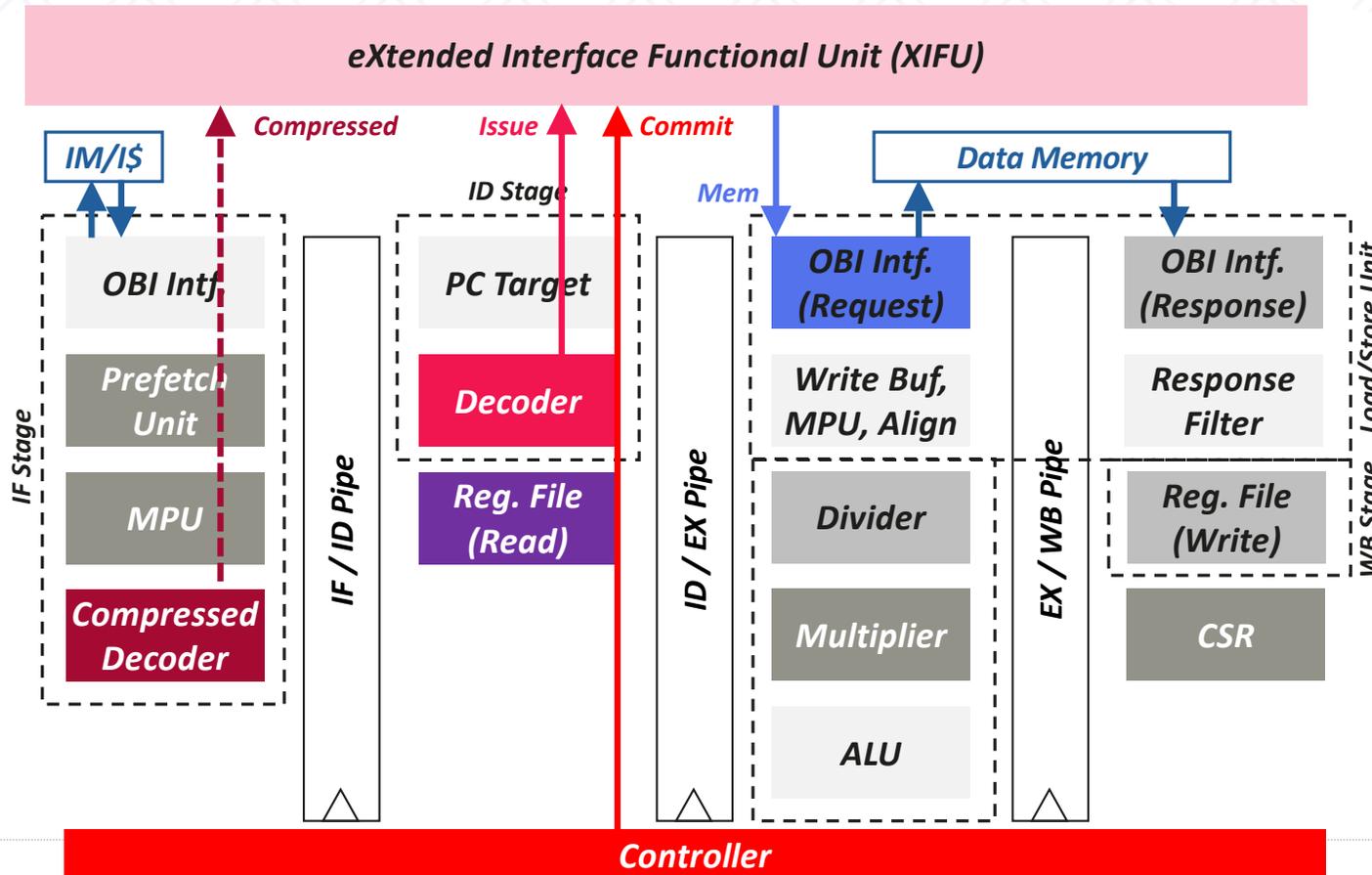
CV-XIF in CV32E40X



Commit interf.

- > Basically correspondent to CV32E40X **Controller**
- > Commits or Kills instructions (e.g., in case of an exception on earlier ones)
- > XIFU does not respond, only complies!

CV-XIF in CV32E40X



Mem interf.

- > Basically correspondent to CV32E40X EX **(memory request)**
- > XIFU can issue memory requests (one or more!) through the CV32E40X LSU
- > Only aligned to word boundary (0x0, 0x4, 0x8, 0xC but not 0x1, 0x2...)

CV-XIF in CV32E40X



eXtended Interface Functional Unit (XIFU)

Compressed Issue Commit

```
typedef struct packed {
    logic [X_ID_WIDTH -1:0] id; // Identification of the offloaded instruction
    logic [ 31:0] addr; // Virtual address of the memory transaction
    logic [ 1:0] mode; // Privilege level
    logic we; // Write enable of the memory transaction
    logic [ 2:0] size; // Size of the memory transaction
    logic [X_MEM_WIDTH/8-1:0] be; // Byte enables for memory transaction
    logic [ 1:0] attr; // Memory transaction attributes
    logic [X_MEM_WIDTH -1:0] wdata; // Write data of a store memory transaction
    logic last; // Is this the last memory transaction for the offloaded instruction?
    logic spec; // Is the memory transaction speculative?
} x_mem_req_t;

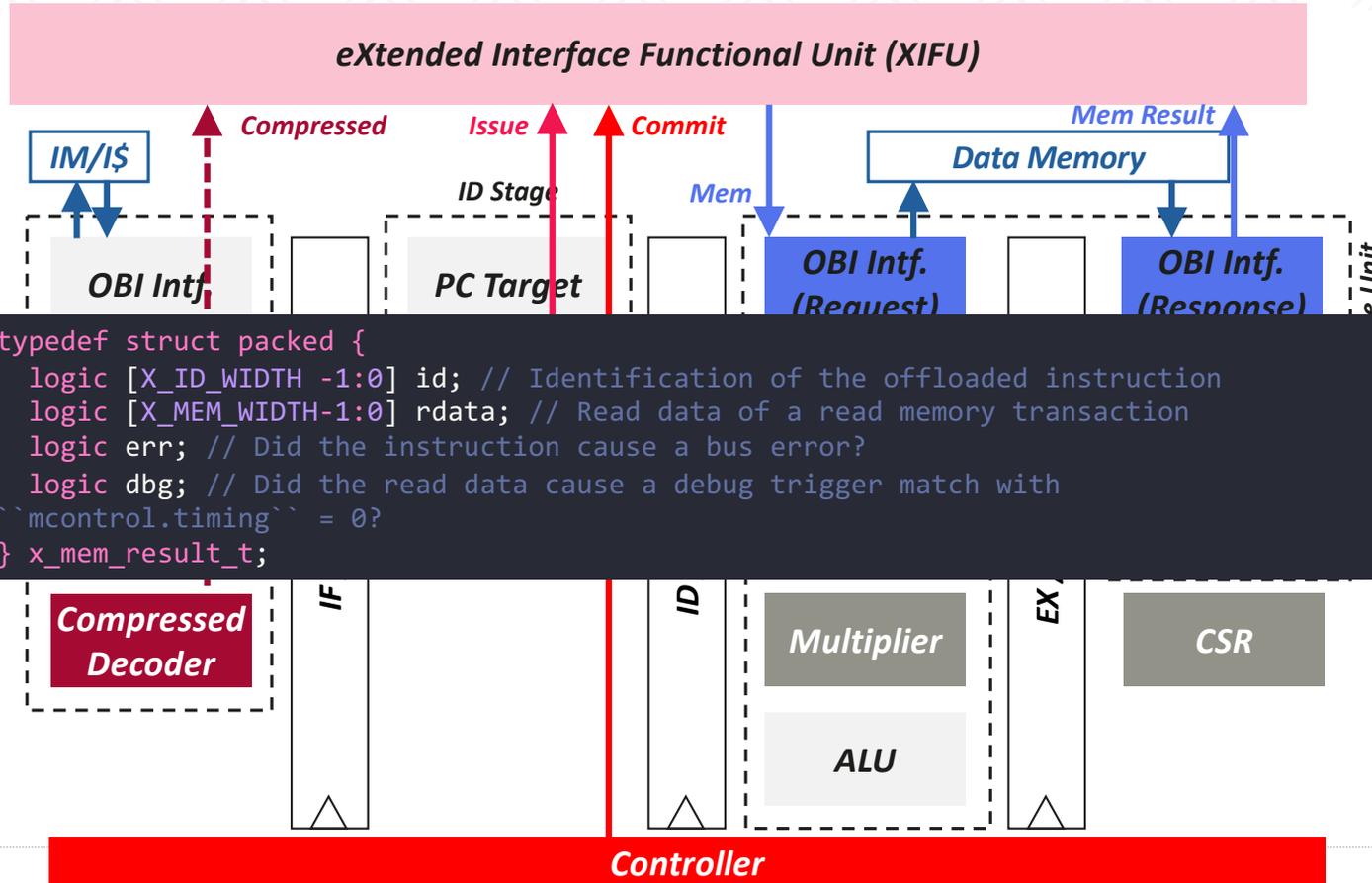
typedef struct packed {
    logic exc; // Did the memory request cause a synchronous exception?
    logic [5:0] exccode; // Exception code
    logic dbg; // Did the memory request cause a debug trigger match with
`mcontrol.timing` = 0?
} x_mem_resp_t;
```

Controller

Mem interf.

- > Basically correspondent to CV32E40X **EX (memory request)**
- > XIFU can issue memory requests (one or more!) through the CV32E40X LSU
- > Only aligned to word boundary (0x0, 0x4, 0x8, 0xC but not 0x1, 0x2...)

CV-XIF in CV32E40X

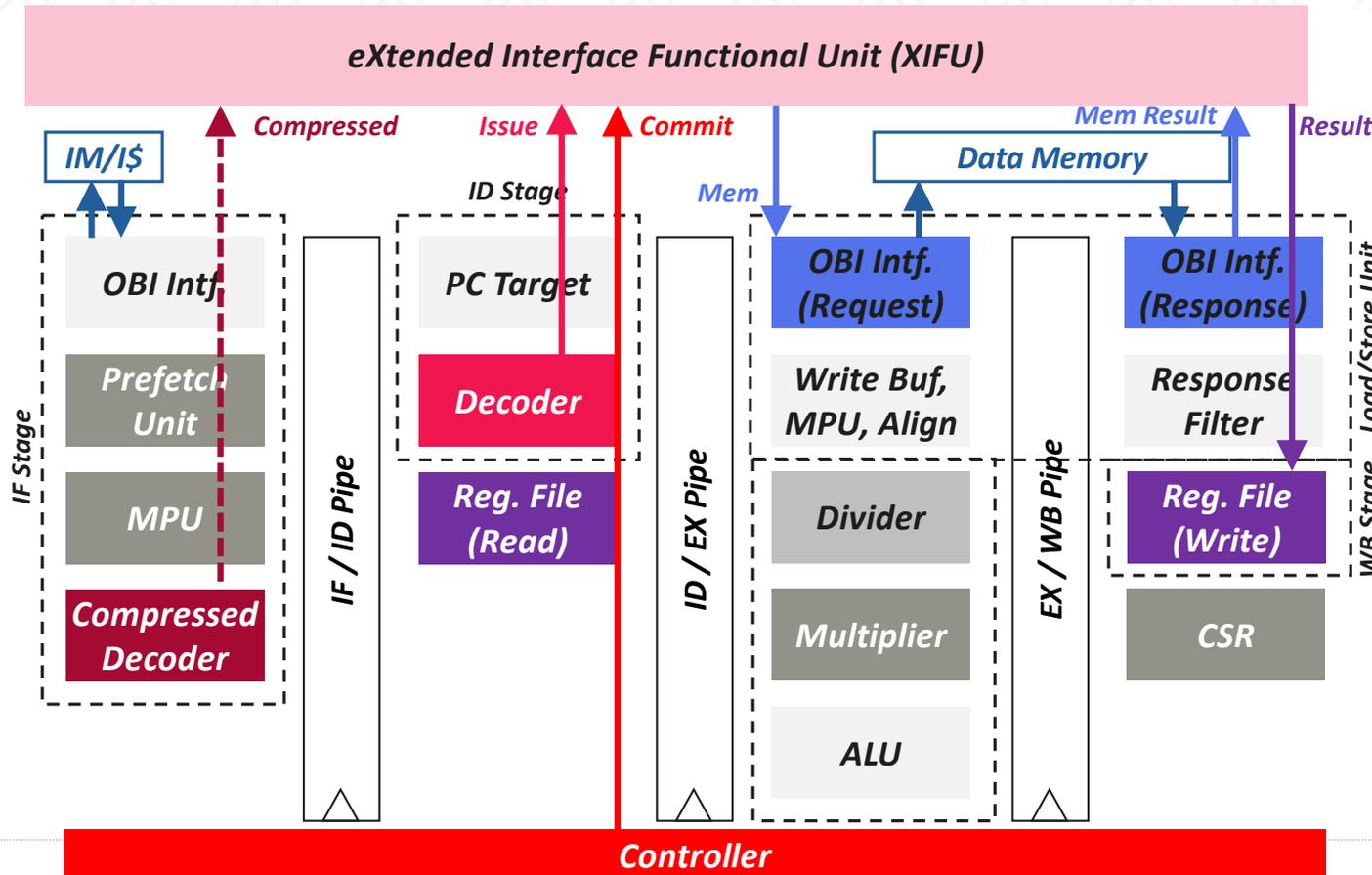


Mem Result interf.

- > Basically correspondent to CV32E40X **WB (memory response)**
- > This interface carries the responses to the memory requests issued in the previous interf.

```
typedef struct packed {
    logic [X_ID_WIDTH-1:0] id; // Identification of the offloaded instruction
    logic [X_MEM_WIDTH-1:0] rdata; // Read data of a read memory transaction
    logic err; // Did the instruction cause a bus error?
    logic dbg; // Did the read data cause a debug trigger match with
    `mcontrol.timing` = 0?
} x_mem_result_t;
```

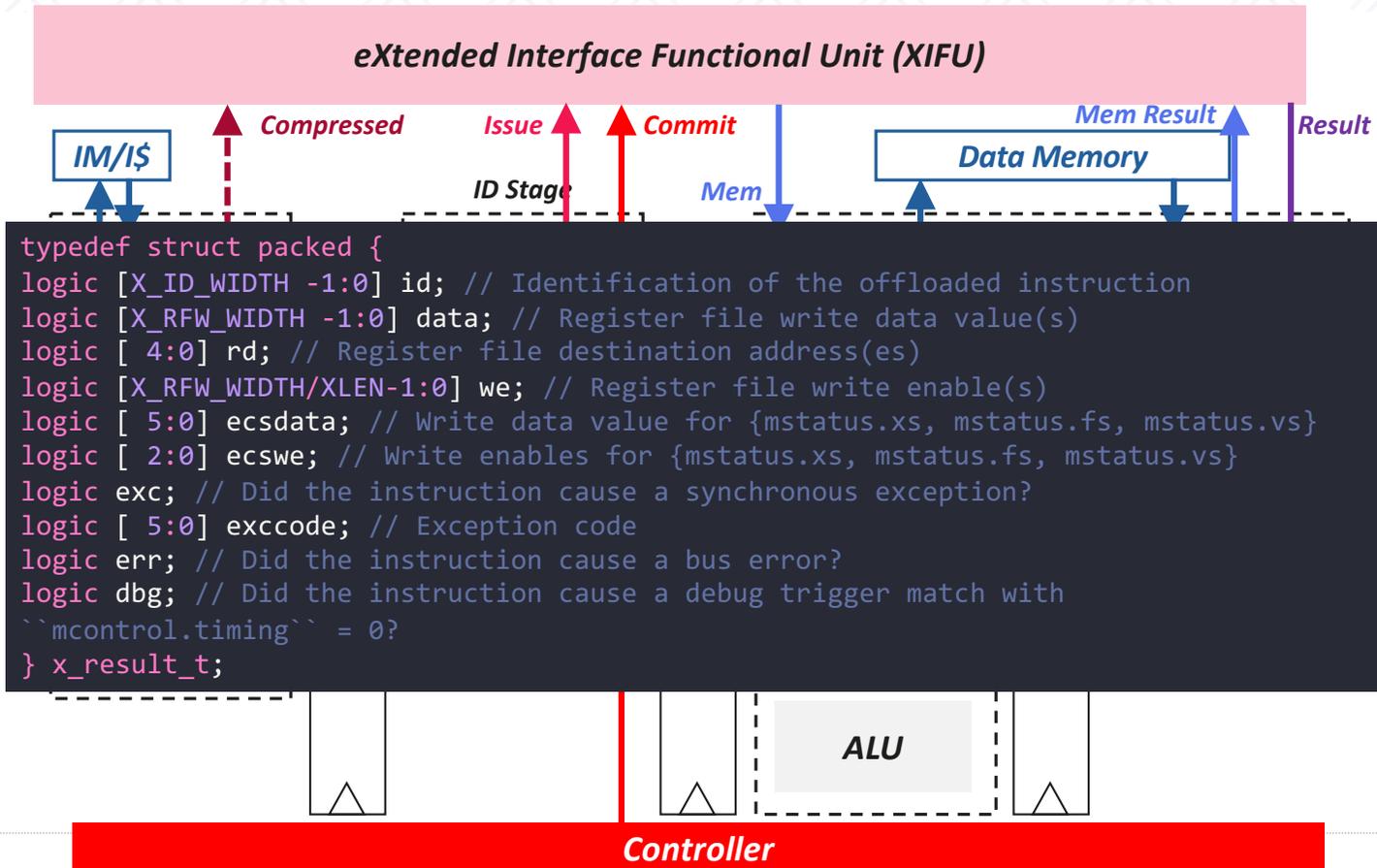
CV-XIF in CV32E40X



Result interf.

- > Basically correspondent to CV32E40X **WB**
- > This interface carries write-back information to the CV32E40X register file

CV-XIF in CV32E40X

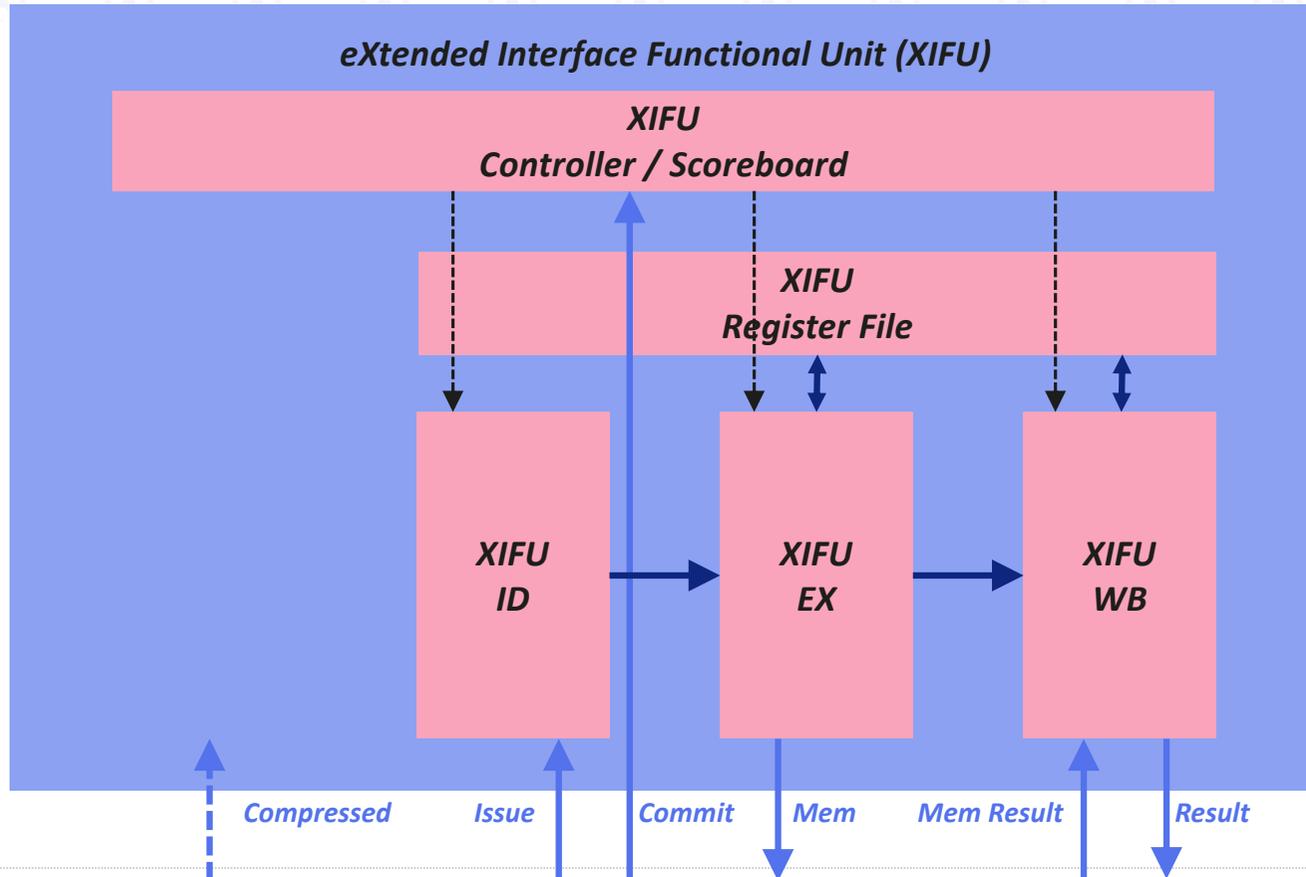


Result interf.

- > Basically correspondent to CV32E40X **WB**
- > This interface carries write-back information to the CV32E40X register file

```
typedef struct packed {  
    logic [X_ID_WIDTH -1:0] id; // Identification of the offloaded instruction  
    logic [X_RFW_WIDTH -1:0] data; // Register file write data value(s)  
    logic [ 4:0] rd; // Register file destination address(es)  
    logic [X_RFW_WIDTH/XLEN-1:0] we; // Register file write enable(s)  
    logic [ 5:0] ecsdata; // Write data value for {mstatus.xs, mstatus.fs, mstatus.vs}  
    logic [ 2:0] ecswe; // Write enables for {mstatus.xs, mstatus.fs, mstatus.vs}  
    logic exc; // Did the instruction cause a synchronous exception?  
    logic [ 5:0] exccode; // Exception code  
    logic err; // Did the instruction cause a bus error?  
    logic dbg; // Did the instruction cause a debug trigger match with  
    ``mcontrol.timing`` = 0?  
} x_result_t;
```

CV-XIF in CV32E40X



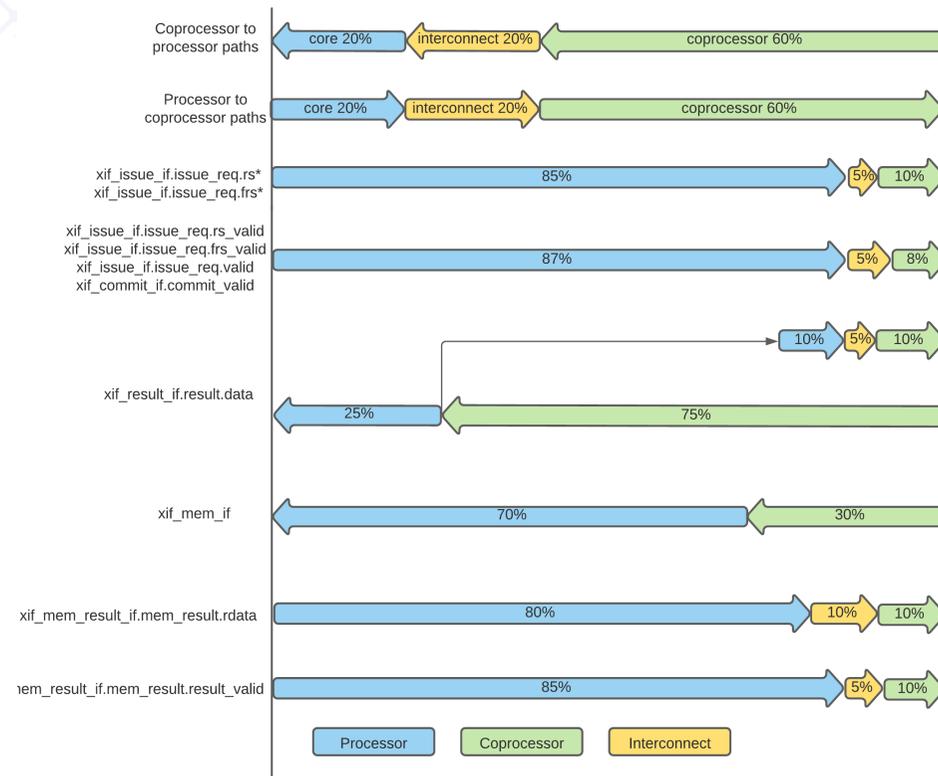
XIFU architecture

- > In theory, arbitrary!
- > In practice depends on *what* we want to implement in the XIFU
 - > **a new instruction?** probably similar to CV32E40X architecture
 - > **just a control interface?** can probably be simpler
 - > **a vector co-processor?** will likely be more complex than the processor itself

CV-XIF timing constraints

Tight integration between core & coprocessor

- > Not a lot of stuff can happen in the coprocessor combinationaly!
- > In practice, there are tight constraints to respect if we want to keep the frequency similar to baseline CV32E40X
 - > Issue interface is particularly tight!



Hands-on time again!



This time we will dive into the design of a XIFU for FIR... a toy example, but complex enough to show you many of these concepts

- Design the XIFU unit in SystemVerilog
 - We provide you with a XIFU template following the same structure as a few slides back...
 - You will implement three special instructions: **xfirlw**, **xfirdotp**, **xfirsw**.
 - You need to decide the instruction encoding... without hitting other RV32IMC instructions!
 - Test the instructions on the baseline RV32IMC compiler
 - This will be very brutal 😊 but a good first step
 - Basically handwritten binary code
 - Adapt the FIR filter application and check the speedup
 - You can start from yesterday's work and build up from that, using your own builtins
-

Thank you

