ETH zürich

OpenSource RISC Processors: An introduction into OpenRISC and RISC-V

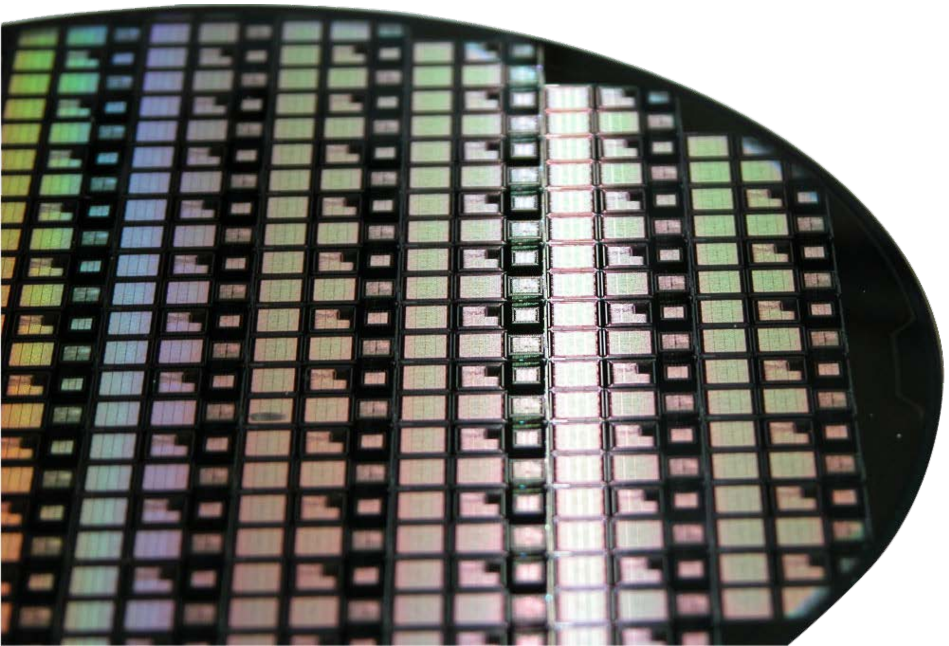
Advanced System-on-Chip Design

Lecture 6

05.04.2016

Michael Gautschi IIS-ETHZ

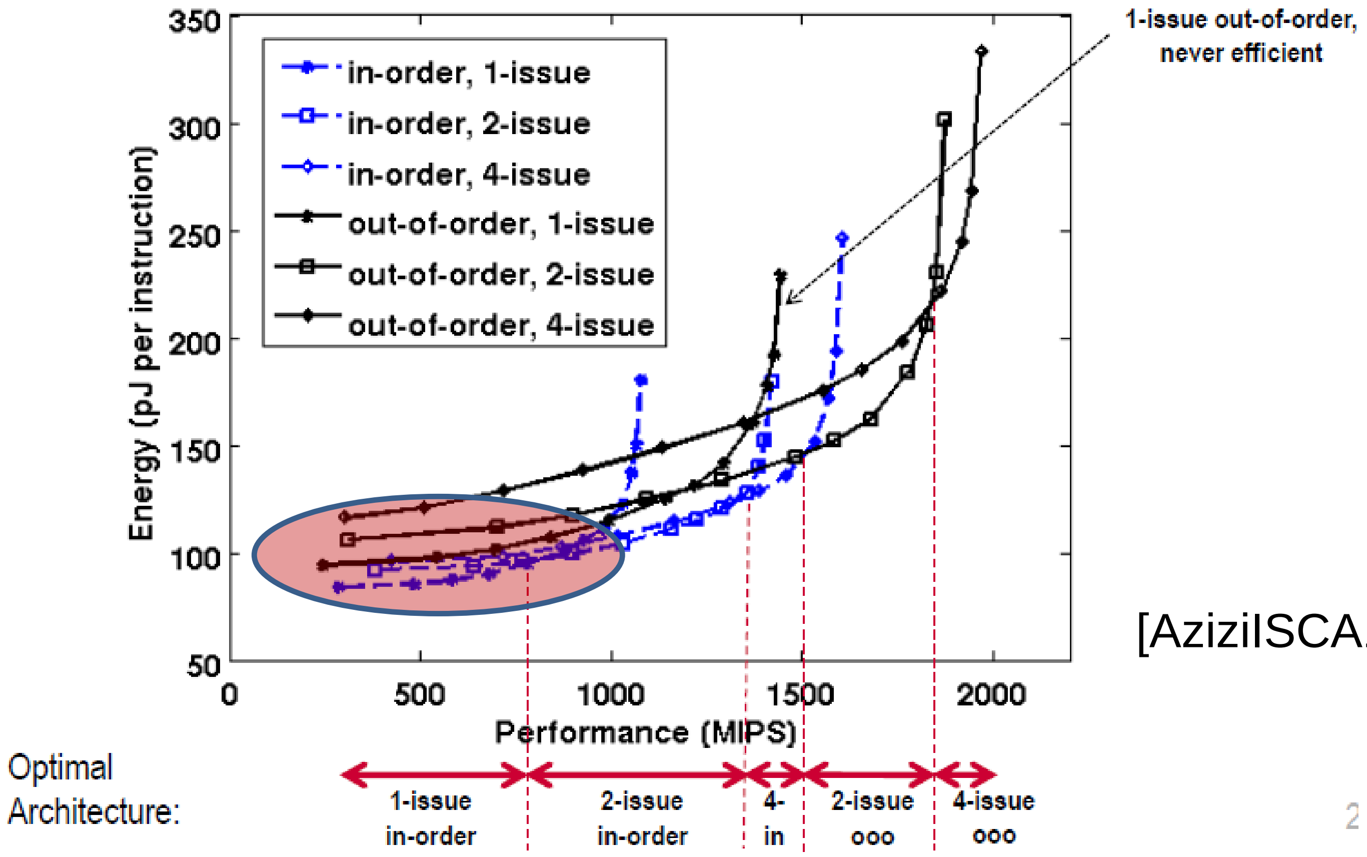
Prof. Luca Benini IIS-ETHZ



ETH zürich

Integrated Systems Laboratory

Introduction – the “best” core



[AziziISCA10]

- Single issue in-order is most energy efficient
- Put more than one + shared memory to fill cluster area

Introduction – Building PULP

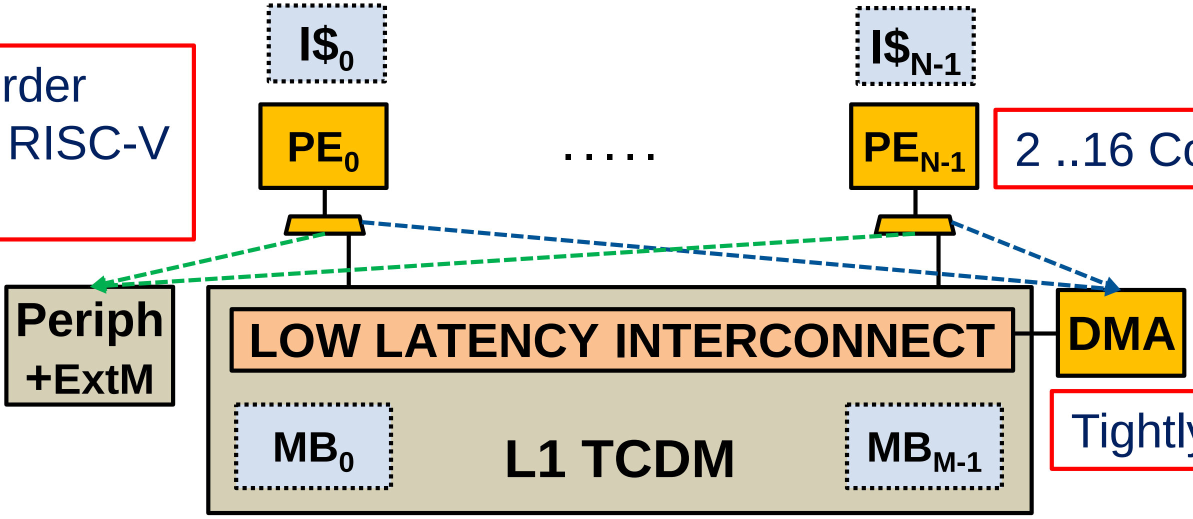
SIMD + MIMD + sequential

Private per-core instruction cache

4-stage, in-order OpenRISC / RISC-V core

DEMUX

2 ..16 Cores



Double buffering

Tightly Coupled DMA

1 Cycle Shared Multi-Banked L1 Data Memory + Low Latency Interconnect

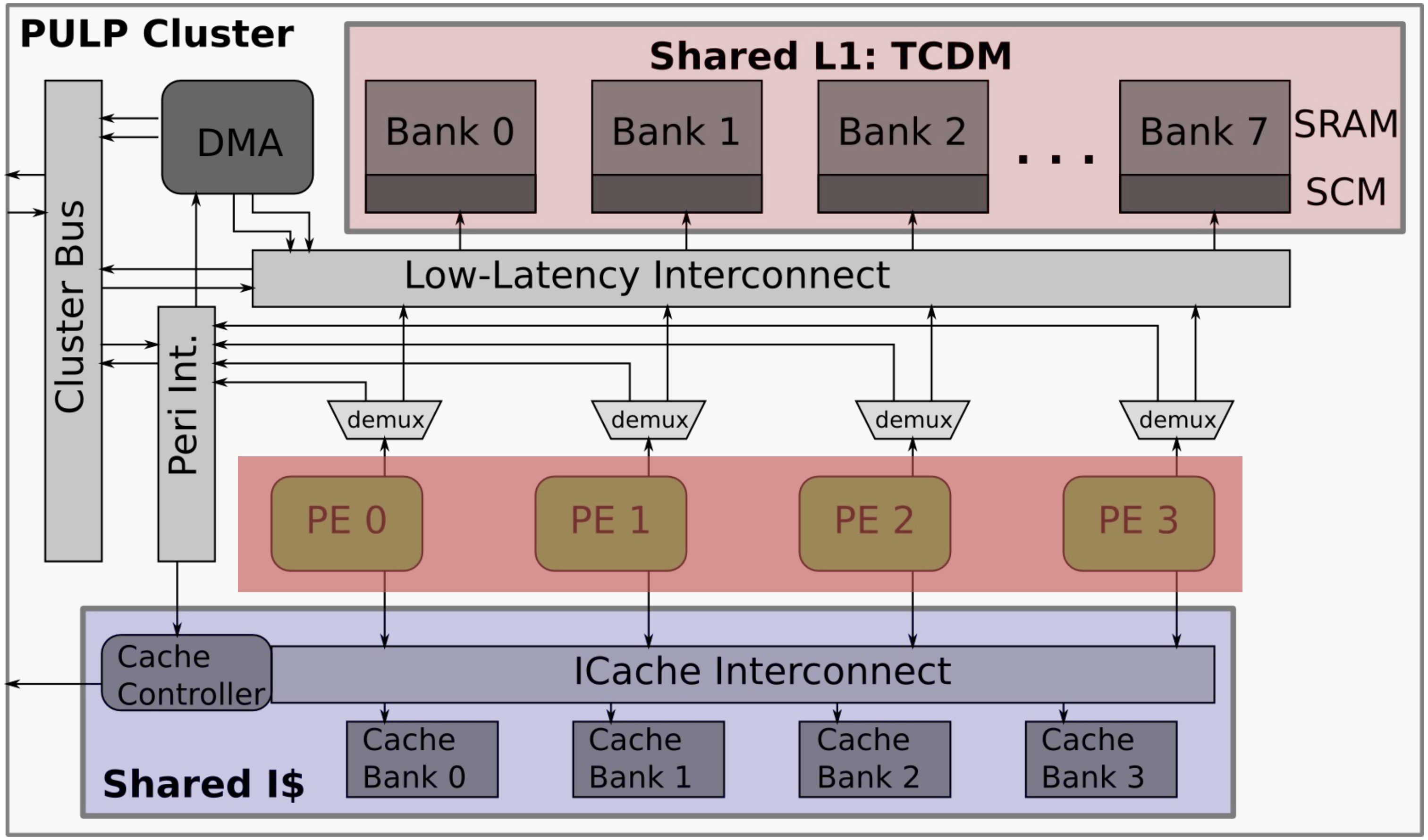
"GPU like" shared memory → low overhead data sharing

Near Threshold but parallel → Maximum Energy efficiency when Active

+ strong power management for (partial) idleness

Introduction – PULP Architecture

- Uses Open Source RISC processor
  - OpenRISC, RISC-V ISA



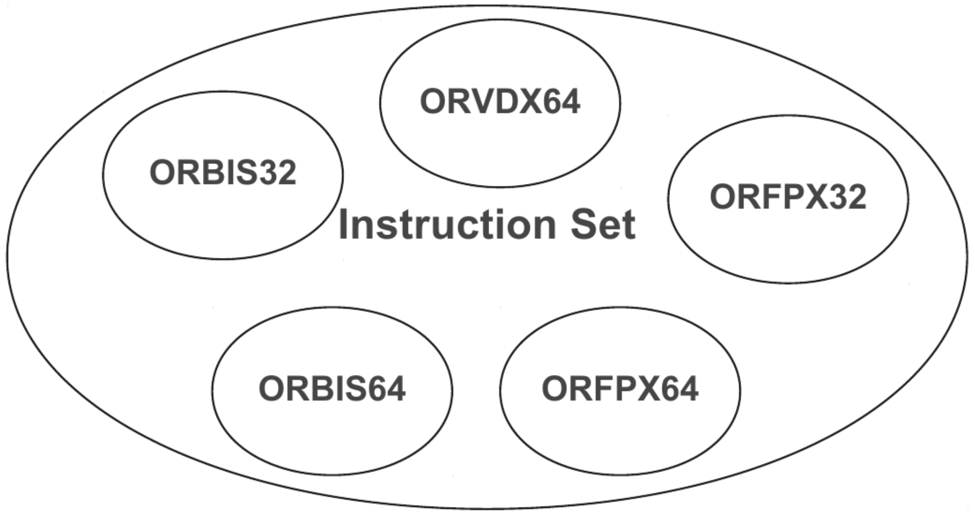
Introduction

- Outline:
  - OpenRISC Instruction-set
    - Basic instruction set
  - Micro-architecture
    - Organization of the pipeline
  - Instruction set extensions for improved performance
    - Hardware and software impact
  - RISC-V architecture
    - Difference to OpenRISC
  - Exercise session about OpenRISC/ RISC-V processor cores
    - Exercise session
- Goals:
  - Learn how to run applications on the Pulpino architecture
  - Understand the impact of the presented hardware extensions
    - Including some pro/ and cons

OpenRISC Instruction Set

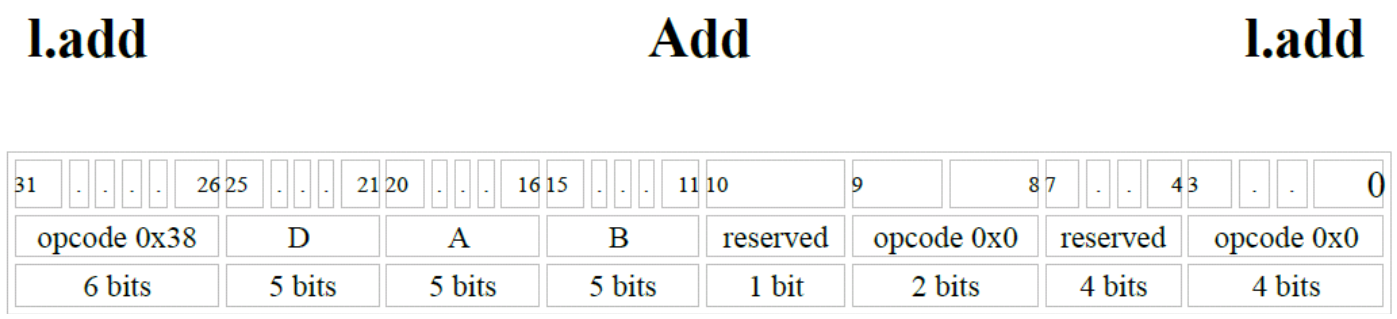
- Open source 32-/64bit RISC architecture
  - Similar to MIPS architecture described in Hennessey/Patterson
  - ORBIS32:
    - 32-bit integer instruction
    - 32-bit load/store instructions
    - Program flow instructions
  - ORBIS64:
    - 64-bit integer instructions
    - 64-bit load/store instructions
  - ORFPX32:
    - Single precision floating point instru
  - ORFPX64:
    - Double-precision floating point instructions
  - ORVDX64:
    - 64-bit vector instructions

⇒ In the following we focus on the 32-bit ORBIS32 instruction set!



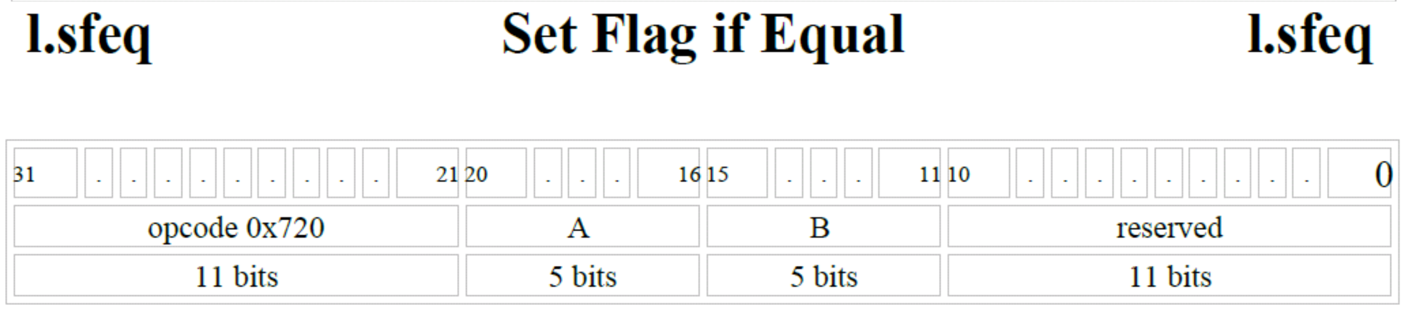
OpenRISC Instruction Set

- ORBISX32 consists of three types of instructions
  - R-type instructions:
    - Register - register operations
    - Examples:
      - ALU operations: *l.add, l.mul, l.sub, l.or, l.mac, etc.*
      - Comparisons: *l.sfeq, l.sfges, etc.*



l.add **Add** **l.add**

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10	9 8	7 . . 4	3 . . 0
opcode 0x38	D	A	B	reserved	opcode 0x0	reserved	opcode 0x0
6 bits	5 bits	5 bits	5 bits	1 bit	2 bits	4 bits	4 bits

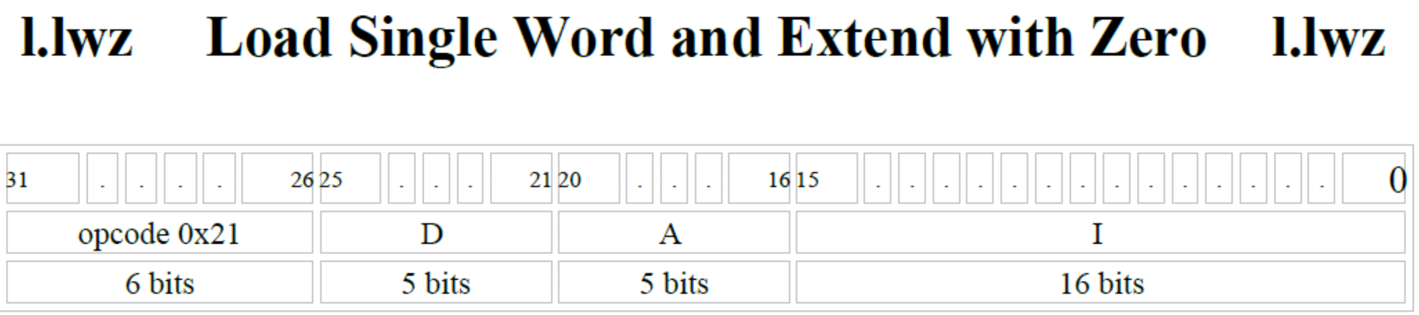


l.sfeq **Set Flag if Equal** **l.sfeq**

31 21	20 . . . 16	15 . . . 11	10 0
opcode 0x720	A	B	reserved
11 bits	5 bits	5 bits	11 bits

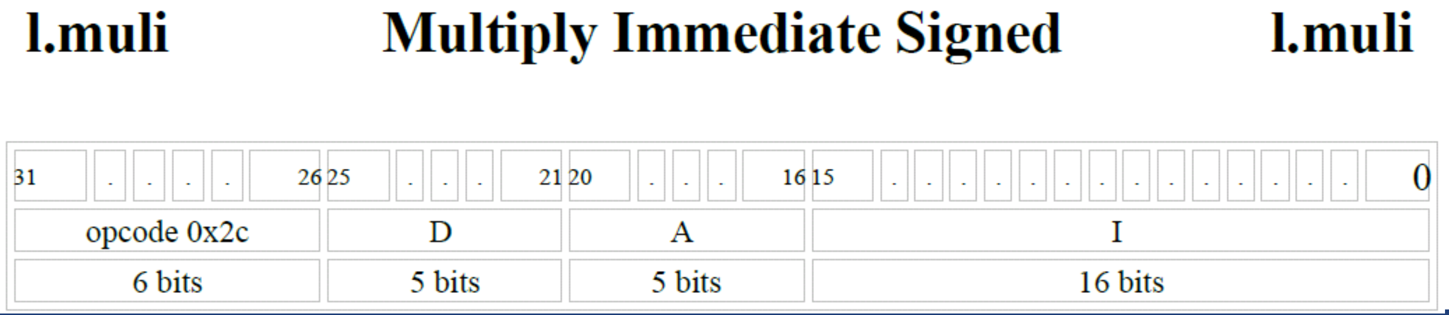
OpenRISC Instruction Set

- ORBISX32 consists of three types of instructions
  - I-type instructions:
    - Operations with an immediate
    - Examples:
      - Load/store operations: *l.lwz, l.sw. l.lhz, l.lbz etc.*
      - ALU operations: *l.addi, l.muli, l.ori, etc.*
      - Comparisons: *l.sfeqi, l.sfnei. etc.*



l.lwz Load Single Word and Extend with Zero l.lwz

31 26	25 . . . 21	20 . . . 16	15 0
opcode 0x21	D	A	I
6 bits	5 bits	5 bits	16 bits

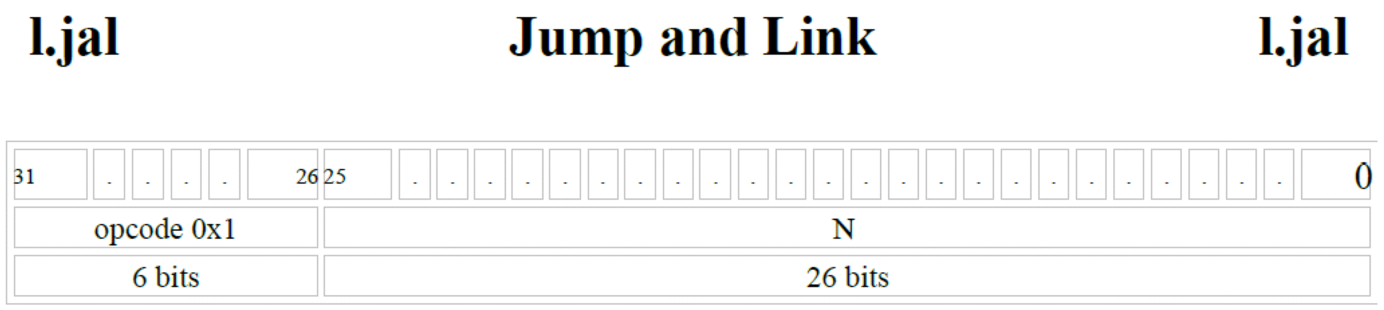


l.muli Multiply Immediate Signed l.muli

31 26	25 . . . 21	20 . . . 16	15 0
opcode 0x2c	D	A	I
6 bits	5 bits	5 bits	16 bits

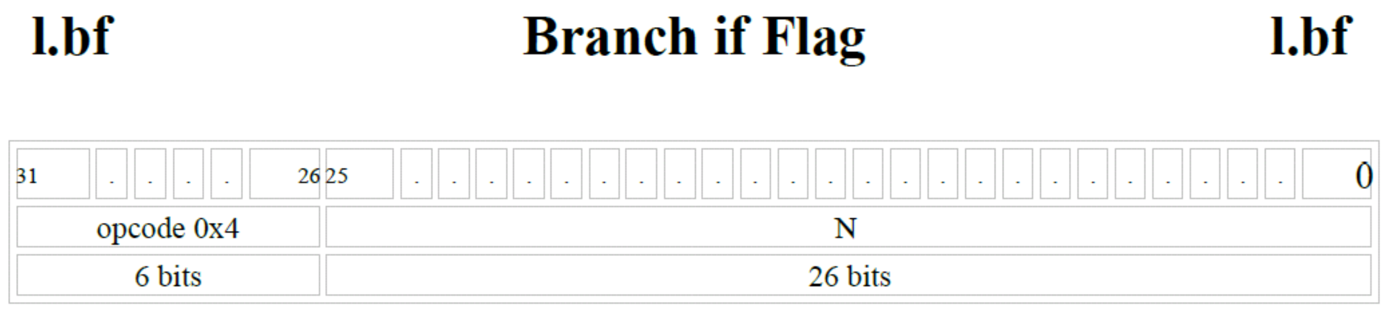
OpenRISC Instruction Set

- ORBISX32 consists of three types of instructions
  - J-type instructions:
    - Jumps and branches
    - Examples:
      - Jump instructions: *l.j, l.jal, l.rfe, etc.*
      - Conditional branches: *l.bf, l.bnf*



l.jal **Jump and Link** **l.jal**

31 26	25 0
opcode 0x1	N
6 bits	26 bits

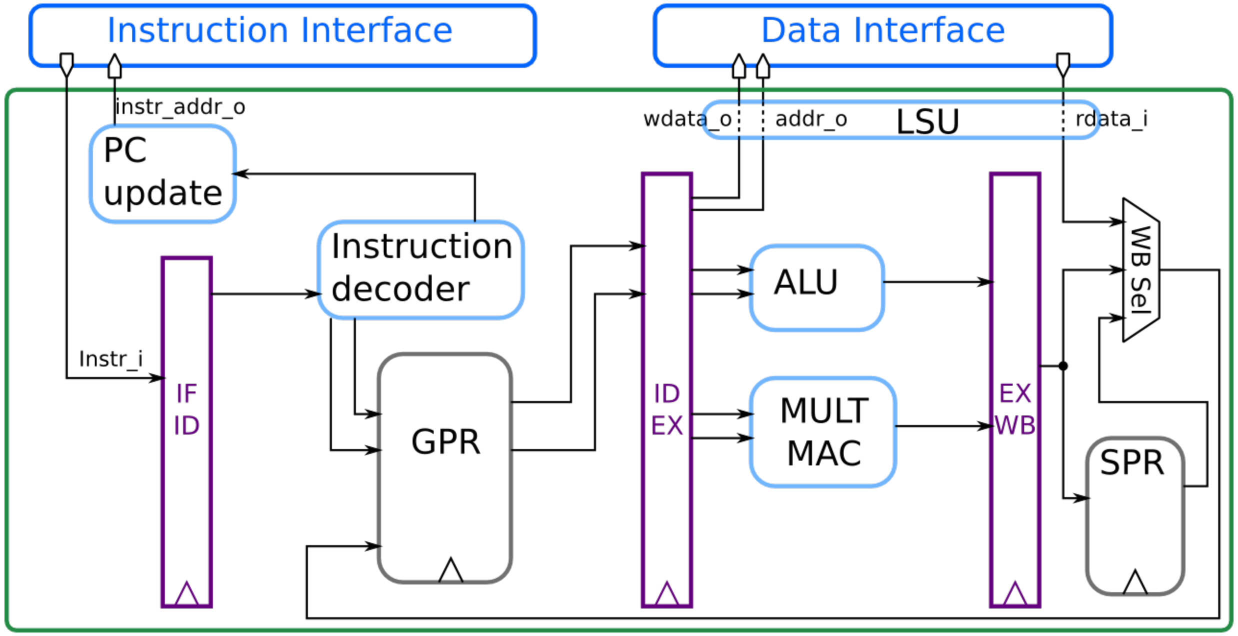


l.bf **Branch if Flag** **l.bf**

31 26	25 0
opcode 0x4	N
6 bits	26 bits

OpenRISC Micro-architecture:

- Core architecture which has been originally developed here at IIS in a semester thesis.
  - The architecture is called OR10N
  - It has been improved over the years and become a good core architecture
- Simple four-stage pipeline architecture: IF, ID, EX, WB
- Single cycle memory access



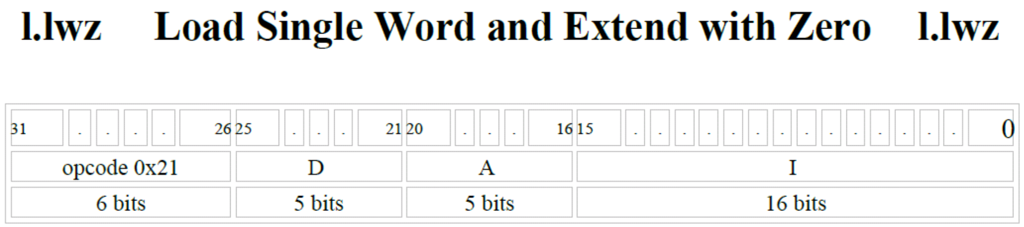
Register file & special purpose registers(SPR)

- Register file organization:
  - 32 registers 32-bit registers
  - Most important registers are:
    - r0 = always zero
    - r1 = stack pointer
    - r9 = link register, holds function return address
    - r11/r12 = return values

- Special purpose registers:
  - Status register contains flags {overflow, carry, branch}
  - Contains registers which are not regularly accessed:
    - Interrupt controller configuration
    - Timer
    - Data/instruction cache control
  - Debug unit
  - Performance counters

Load/Store Unit

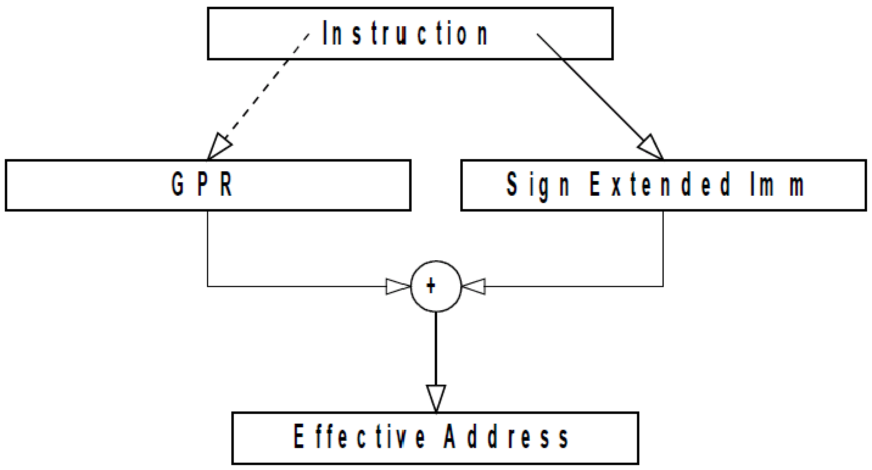
- 32 bit load-store interface
- Supported instructions:
  - Load word/halfword/ byte
    - With zero or sign extension



l.lwz Load Single Word and Extend with Zero l.lwz

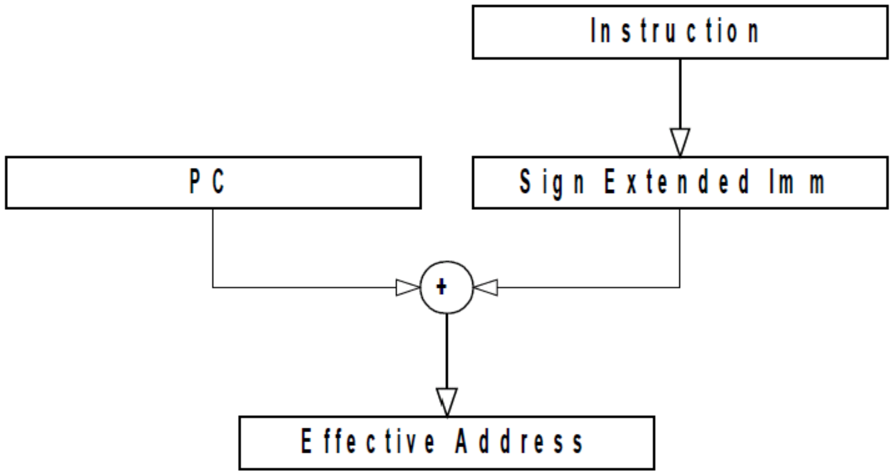
31 26	25 . . . 21	20 . . . 16	15 0
opcode 0x21	D	A	I
6 bits	5 bits	5 bits	16 bits

- Addressing mode aligned data requests
  - l.lwz/s word aligned
  - l.lhz/s half word aligned
  - l.lbz/s byte aligned
- Stall pipeline if exception has been detected
  - Access to protected address
  - Unaligned access
- No out of order requests



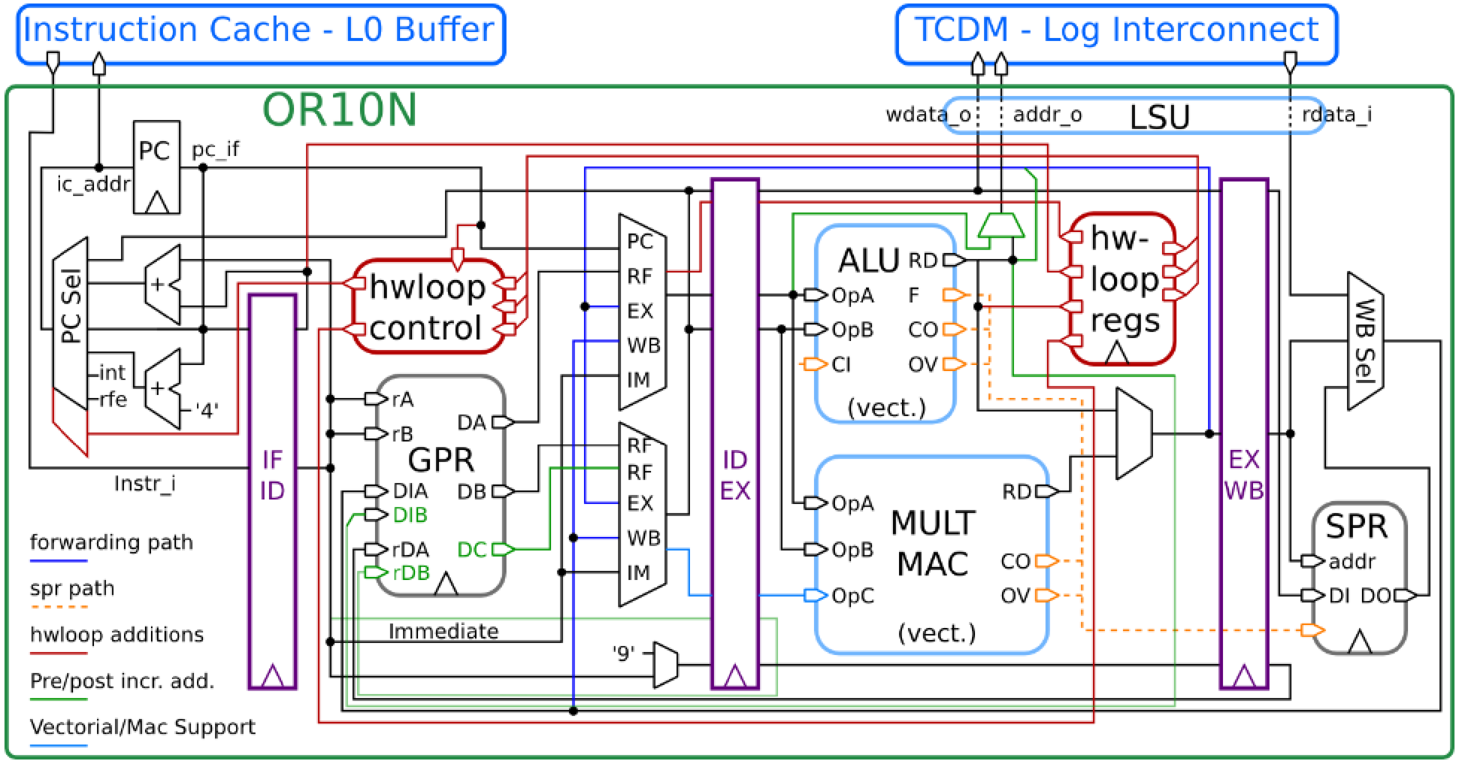
OpenRISC: Control Flow

- Branches
  - l.bnf : jump to PC + sign extended immediate if flag is not set
  - l.bf : jump to PC + sign extended immediate if flag is set
  - Delay slot is always executed
- Jumps:
  - l.jr : jump to address stored in a register
  - l.jalr : jump to address stored in a register and link r9 to instruction after delay slot
  - l.j : jump to PC + sign extended immediate
  - l.jal : jump to PC + sign extended immediate and link r9
  - l.rfe : return from exception, jump to EPCR
- No support for VLIW
  - Instructions are always 32 bit



OR10N Instruction Extensions for OR10N:

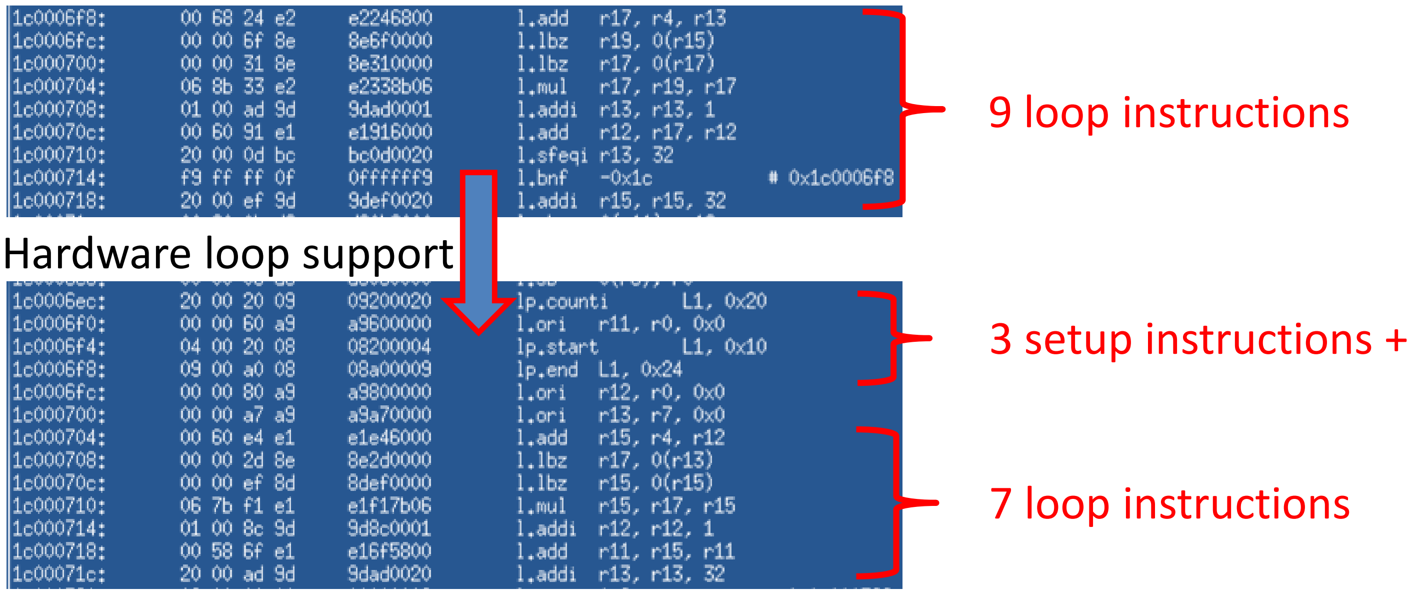
- In order to improve performance and efficiency of the core we have evaluated several instructions and added the following instructions:
  - Hardware loops
  - Pre/post memory address update
  - New MAC
  - Vector unit
  - Unaligned memory access



Instruction Extensions: Hardware Loops

- Hardware loops or Zero Overhead Loops can be implemented to remove the branch overhead in for loops.
- After configuration with start, end, count variables no more comparison and branches are required.
- Smaller loop benefit more!

- Loop needs to be set up beforehand and is fully defined by:
  - Start address
  - End address
  - Counter



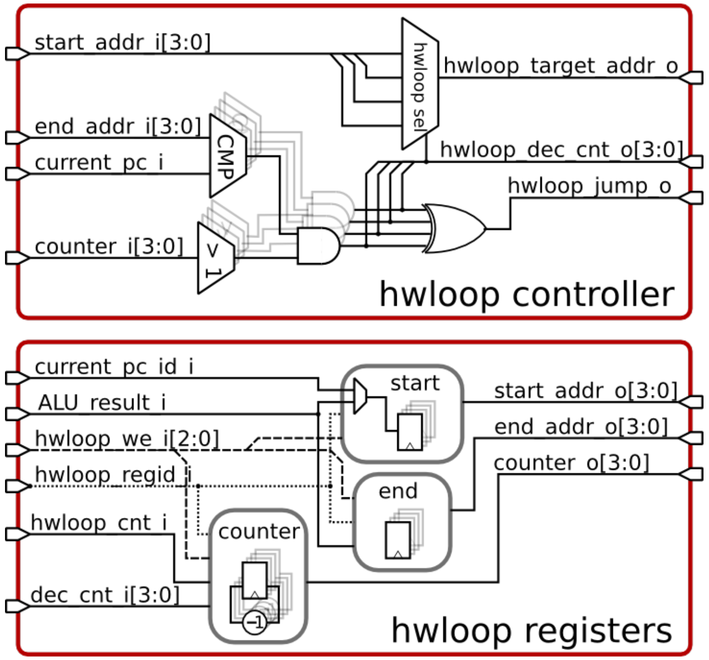
```
1c0006f8:    00 68 24 e2    e2246800    l.add   r17, r4, r13
1c0006fc:    00 00 6f 8e    8e6f0000    l.lbz   r19, 0(r15)
1c000700:    00 00 31 8e    8e310000    l.lbz   r17, 0(r17)
1c000704:    06 8b 33 e2    e2338b06    l.mul   r17, r19, r17
1c000708:    01 00 ad 9d    9dad0001    l.addi  r13, r13, 1
1c00070c:    00 60 91 e1    e1916000    l.add   r12, r17, r12
1c000710:    20 00 0d bc    bc0d0020    l.sfeqi r13, 32
1c000714:    f9 ff ff 0f    0ffffff9    l.bnf   -0x1c          # 0x1c0006f8
1c000718:    20 00 ef 9d    9def0020    l.addi  r15, r15, 32
```

Hardware loop support

```
1c0006ec:    20 00 20 09    09200020    lp.counti      L1, 0x20
1c0006f0:    00 00 60 a9    a9600000    l.ori   r11, r0, 0x0
1c0006f4:    04 00 20 08    08200004    lp.start       L1, 0x10
1c0006f8:    09 00 a0 08    08a00009    lp.end  L1, 0x24
1c0006fc:    00 00 80 a9    a9800000    l.ori   r12, r0, 0x0
1c000700:    00 00 a7 a9    a9a70000    l.ori   r13, r7, 0x0
1c000704:    00 60 e4 e1    e1e46000    l.add   r15, r4, r12
1c000708:    00 00 2d 8e    8e2d0000    l.lbz   r17, 0(r13)
1c00070c:    00 00 ef 8d    8def0000    l.lbz   r15, 0(r15)
1c000710:    06 7b f1 e1    e1f17b06    l.mul   r15, r17, r15
1c000714:    01 00 8c 9d    9d8c0001    l.addi  r12, r12, 1
1c000718:    00 58 6f e1    e16f5800    l.add   r11, r15, r11
1c00071c:    20 00 ad 9d    9dad0020    l.addi  r13, r13, 32
```

Instruction Extensions: Hardware Loops

- Two sets registers implemented to support nested loops.
- Area costs:
  - Processor core area increases by 5%
- Performance:
  - Speedup can be up to factor 2!



- Hardware loop setup with:
  - 3 separate instructions
    lp.start, lp.end, lp.count, lp.counti
    ⇒ No restriction on start/end address
  - Fast setup instructions
    lp.setup, lp.setupi
    ⇒ Start address= PC + 4
    ⇒ End address= start address + offset
    ⇒ Counter from immediate/register

Instruction format and Opcode	**Semantics**

`lp.start J, S          (eg. lp.start L0, 10)`
`000010 000 JJ SSSSSSSSSSSSSSSSSSSSS`
	`HWLP_START[J]=sext(S*4)+PC`

`lp.end J, S              (eg. lp.end L0, -8)`
`000010 001 JJ EEEEEEEEEEEEEEEEEEEEE`
	`HWLP_END[J]   =sext(E*4)+PC`

`lp.counti J, C        (eg. lp.counti L0, 8)`
`000010 010 JJ CCCCCCCCCCCCCCCCCCCCC`
	`HWLP_COUNT[J]=zext(C)`

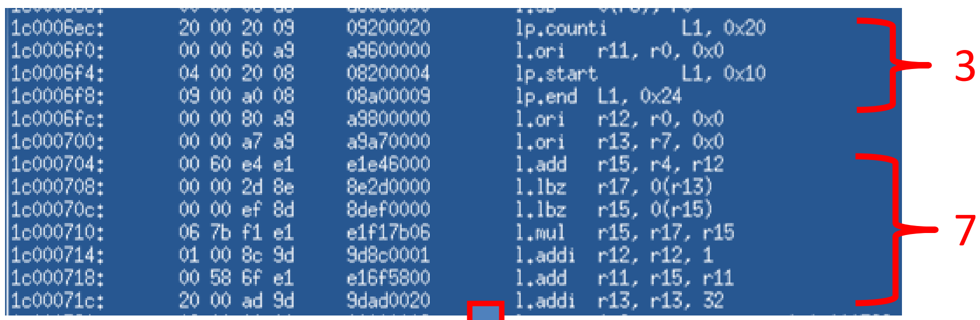
`lp.count J, rA         (eg. lp.count L0, r5)`
`000010 011 JJ AAAAA----------------`
	`HWLP_COUNT[J]=[rA]`

`lp.setupi J, E, C (eg. lp.setupi L0, 4, 8)`
`000010 100 JJ CCCCCCCCCCCCEEEEEEEEE`
	`HWLP_START[J]=PC+4`
	`HWLP_END[J]   =zext(E*4)+PC`
	`HWLP_COUNT[J]=zext(C)`

`lp.setup J, E, rA (eg. lp.setup L0, 8, r5)`
`000010 101 JJ AAAAAEEEEEEEEEEEEEEEE`
	`HWLP_START[J]=PC+4`
	`HWLP_END[J]   =zext(E*4)+PC`
	`HWLP_COUNT[J]=[rA]`

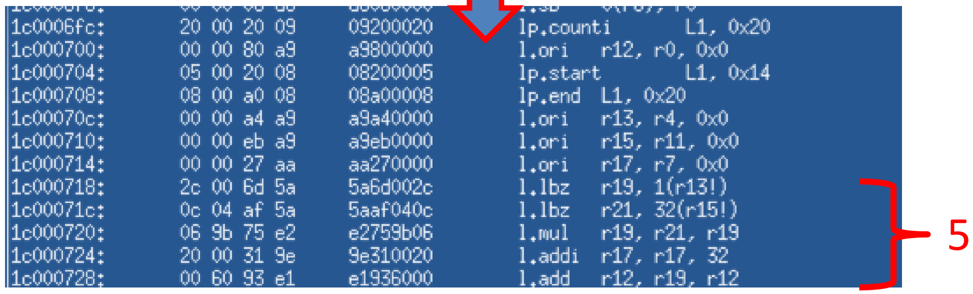
Instruction Extensions: Post increment

- Automatic address update
  - Update base register with computed address after the memory access
  - ⇒ Save instructions to update address register
  - Post-increment:
    - Base address serves as memory address
- Offset can be stored in:
  - Register
  - Immediate



```
1c0006ec:    20 00 20 09    09200020    lp.counti     L1, 0x20
1c0006f0:    00 00 60 a9    a9600000    l.ori   r11, r0, 0x0
1c0006f4:    04 00 20 08    08200004    lp.start      L1, 0x10
1c0006f8:    09 00 a0 08    08a00009    lp.end  L1, 0x24
1c0006fc:    00 00 80 a9    a9800000    l.ori   r12, r0, 0x0
1c000700:    00 00 a7 a9    a9a70000    l.ori   r13, r7, 0x0
1c000704:    00 60 e4 e1    e1e46000    l.add   r15, r4, r12
1c000708:    00 00 2d 8e    8e2d0000    l.lbz   r17, 0(r13)
1c00070c:    00 00 ef 8d    8def0000    l.lbz   r15, 0(r15)
1c000710:    06 7b f1 e1    e1f17b06    l.mul   r15, r17, r15
1c000714:    01 00 8c 9d    9d8c0001    l.addi  r12, r12, 1
1c000718:    00 58 6f e1    e16f5800    l.add   r11, r15, r11
1c00071c:    20 00 ad 9d    9dad0020    l.addi  r13, r13, 32
```

post increment support



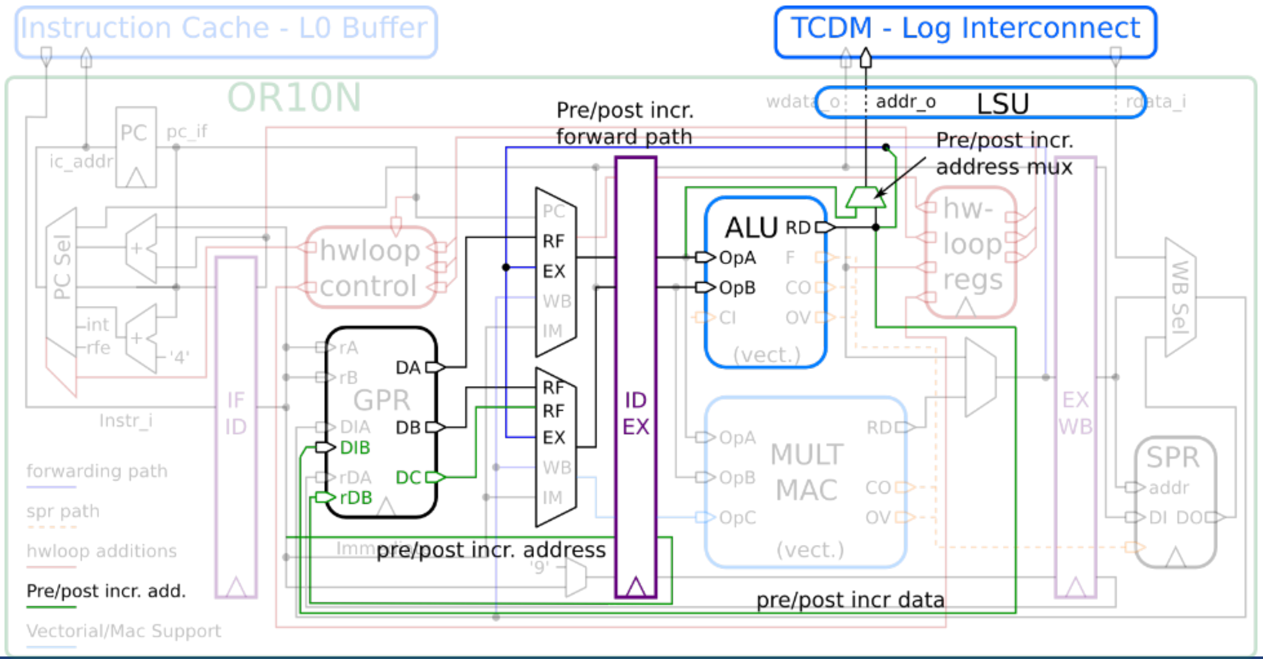
```
1c0006fc:    20 00 20 09    09200020    lp.counti     L1, 0x20
1c000700:    00 00 80 a9    a9800000    l.ori   r12, r0, 0x0
1c000704:    05 00 20 08    08200005    lp.start      L1, 0x14
1c000708:    08 00 a0 08    08a00008    lp.end  L1, 0x20
1c00070c:    00 00 a4 a9    a9a40000    l.ori   r13, r4, 0x0
1c000710:    00 00 eb a9    a9eb0000    l.ori   r15, r11, 0x0
1c000714:    00 00 27 aa    aa270000    l.ori   r17, r7, 0x0
1c000718:    2c 00 6d 5a    5a6d002c    l.lbz   r19, 1(r13!)
1c00071c:    0c 04 af 5a    5aaf040c    l.lbz   r21, 32(r15!)
1c000720:    06 9b 75 e2    e2759b06    l.mul   r19, r21, r19
1c000724:    20 00 31 9e    9e310020    l.addi  r17, r17, 32
1c000728:    00 60 93 e1    e1936000    l.add   r12, r19, r12
```

⇒ save 2 additional instructions to track the read address of the operands!

Instruction Extensions: Post increment

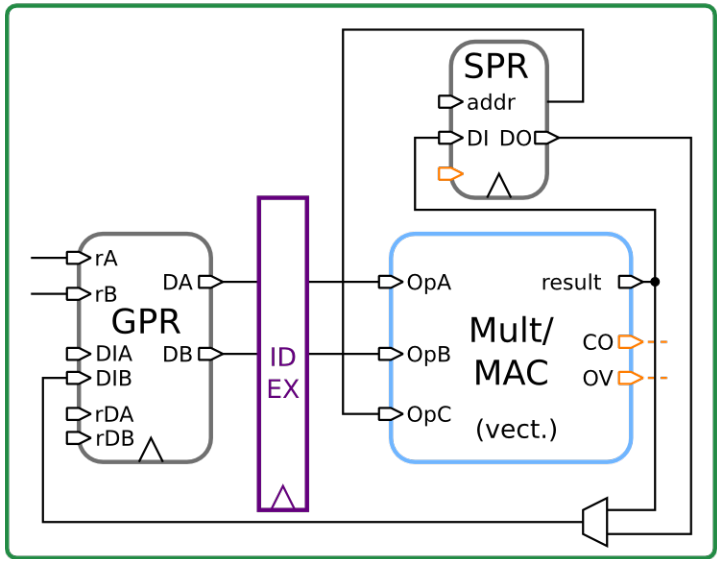
- Register file requires additional write port
- Register file requires additional read port if offset is stored in register
- Processor core area increases by 8-12 %
  - Ports can be used for other instructions

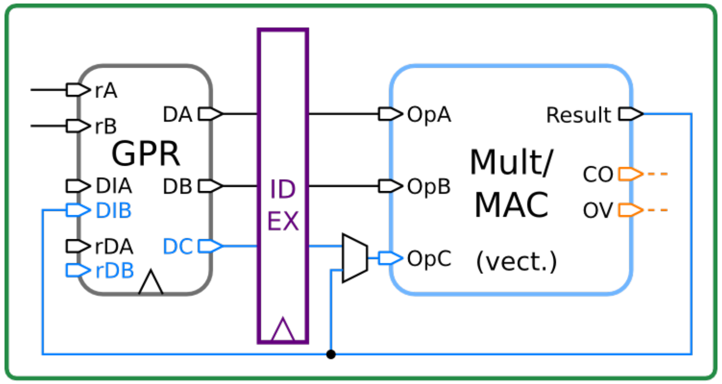
Old Instruction format	Opcode
`l.s[bhw] I(rA), rB`	`MMMMMM IIIII AAAAA BBBBB IIIIIIIIIII`
`l.l[bhw][zs] rD, I(rA)`	`MMMMMM DDDDD AAAAA IIIIIIIIIIIIIIII`
New Instruction format	**Opcode**
`l.s[bhw] I(rA!), rB`	`010100 MMMMM AAAAA BBBBB IIIIIIIIIII`
`l.s[bhw] I(!rA), rB`	`010100 MMMMM AAAAA BBBBB IIIIIIIIIII`
`l.l[bhw][zs] rD, rC(rA)`	`010110 MMMMM AAAAA DDDDD CCCCC ------`
`l.l[bhw][zs] rD, I(rA!)`	`010100 MMMMM AAAAA DDDDD IIIIIIIIIII`
`l.l[bhw][zs] rD, I(!rA)`	`010100 MMMMM AAAAA DDDDD IIIIIIIIIII`
`l.l[bhw][zs] rD, rC(rA!)`	`010111 MMMMM AAAAA DDDDD CCCCC ------`
`l.l[bhw][zs] rD, rC(!rA)`	`010111 MMMMM AAAAA DDDDD CCCCC ------`



New MAC: Accumulation on register file

- Old MAC:
  - Accumulation on special 64 bit register
- New MAC:
  - Accumulation only on 32 bit data
  - Directly on the register file
- Pro:
  - Faster access to mac accumulation
  - Many accumulations in parallel
  - Single cycle mult/mac
- Contra:
  - Additional read port on the register file
    - can be used for pre/post increment with register

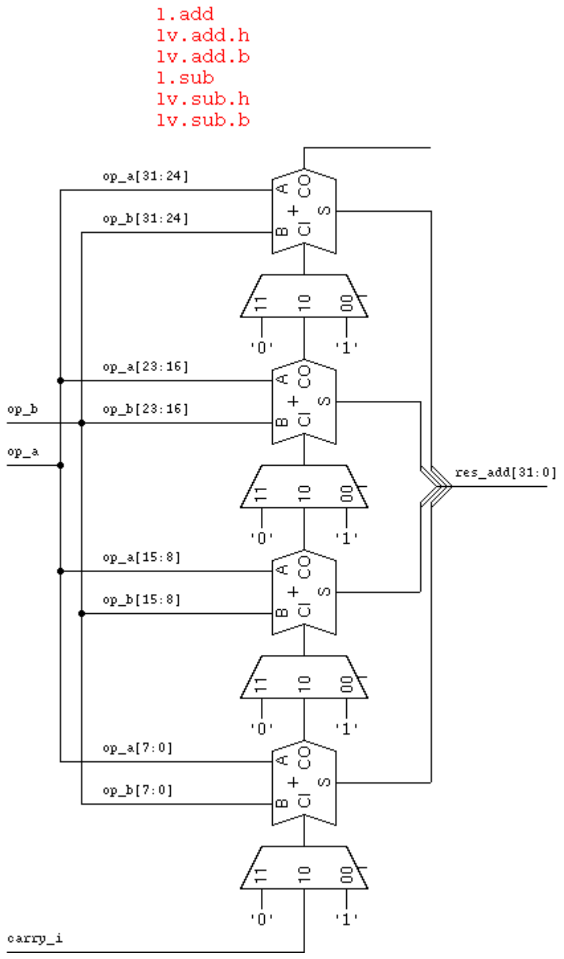




Instruction Extensions: 32 bit Vector Support

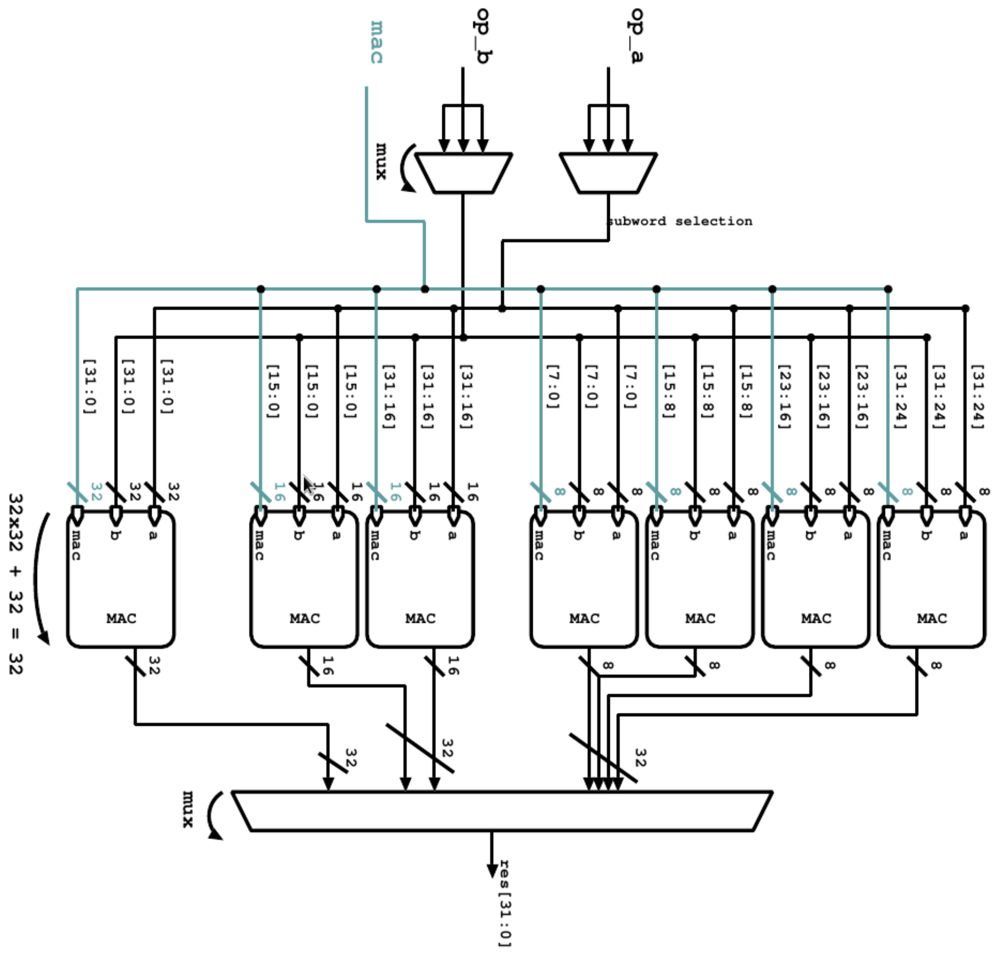
- Vector modes: (bytes, halfwords, word)
  - 4 byte operations
    - With byte select
  - 2 halfword operations
    - With halfword select
  - 1 word operation
- Vector ALU supports:
  - Vector additions
  - Vector subtractions
  - Vector comparisons:
    - Rise flag if *any*, or *all* conditions are true
- Fused vector Mult/Mac:
  - Dynamic range problem in vector multiplications

Vectorial Adder:



32 bit Vector Support: Vectorial Multiplication

- Multiplication on
  - Word,
  - Halfword
  - Byte
- 3 different multipliers:
  - 4* (8x8=8) mults.
  - 2* (16x16=16) mults.
  - 1*(32x32=32) mult.
- Output has same dynamic range as input!
  - Overflows fast in 8bit case



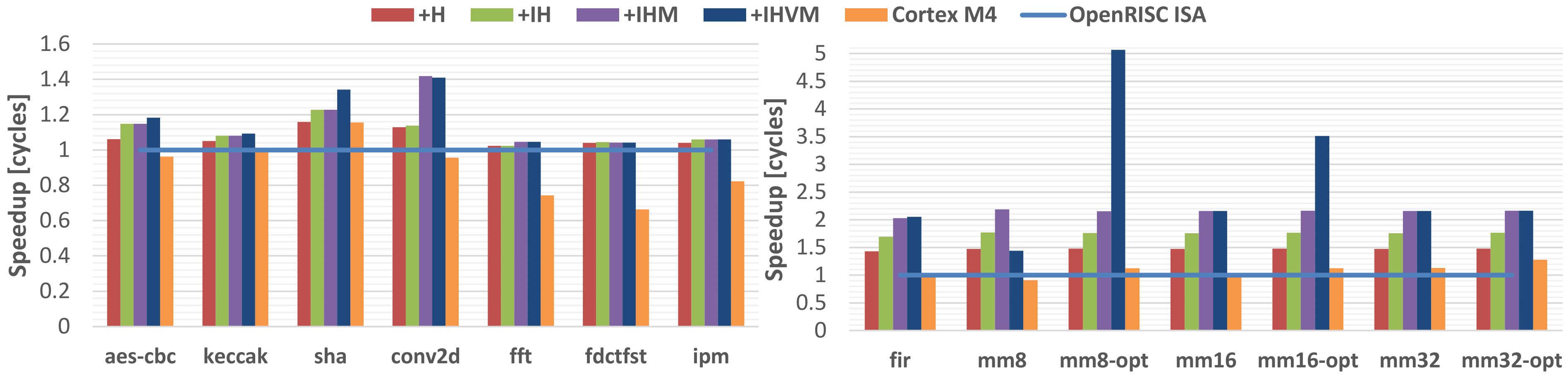
Performance Improvements:

Hardware loops (H) & pre-/post increment (IH):
- Up to 1.75x speedup

Optimized MAC (IHM)
- Up to 2.25x speedup

Vector extensions (IHVM)
- Up to 5x speedup but only rarely used => Dot product

Cortex M4: STM32F429ZI
- Slower, no intrinsic used

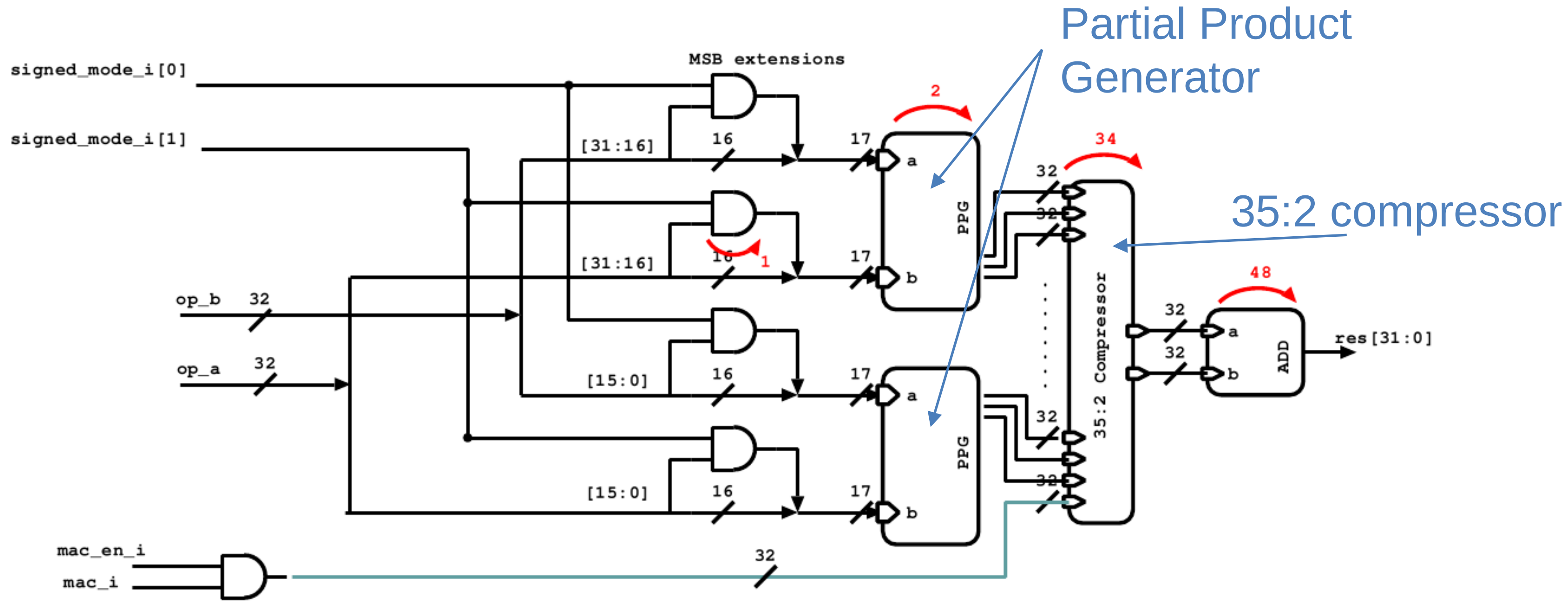


32 bit Vector Support: Dot Product Multiplier

- Dot Product: (half word example)

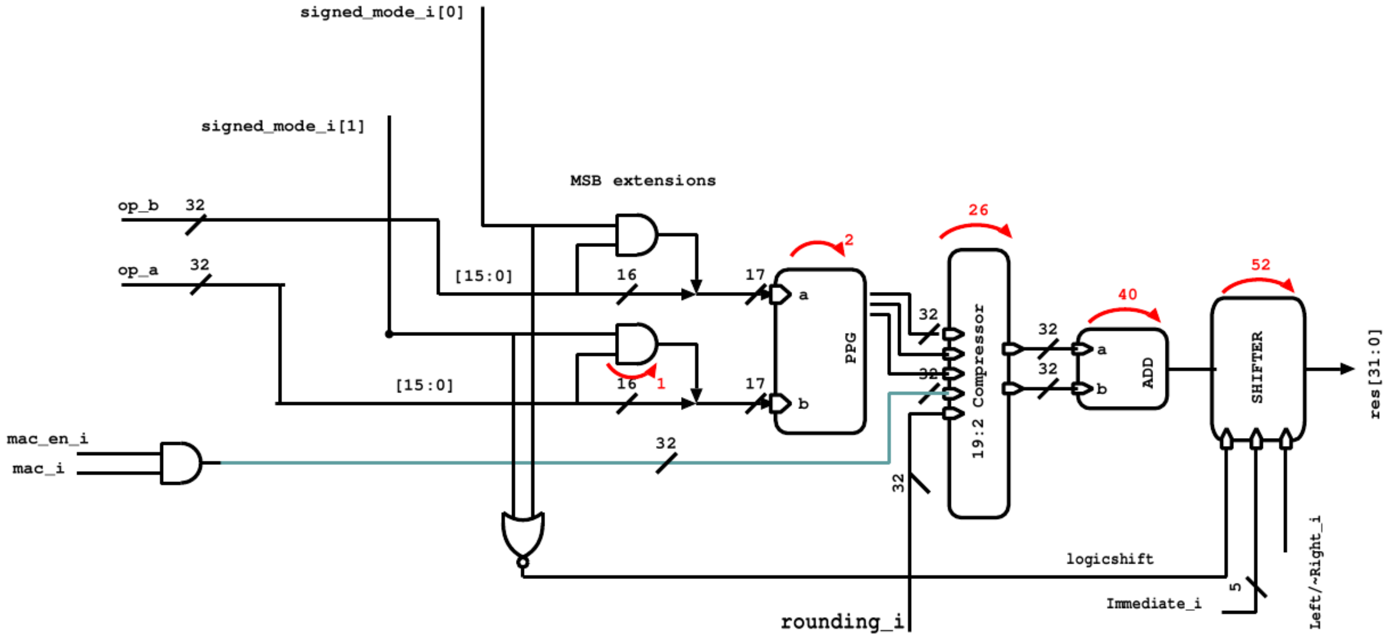
$$C[31:0] = \underbrace{A[31:16]*B[31:16]}_{\text{32 bit}} + \underbrace{A[15:0]*B[15:0]}_{\text{32 bit}} + \underbrace{C[31:0]}_{\text{32 bit}}$$

=> 2 multiplications, 1 addition, 1 accumulation in 1 cycle!

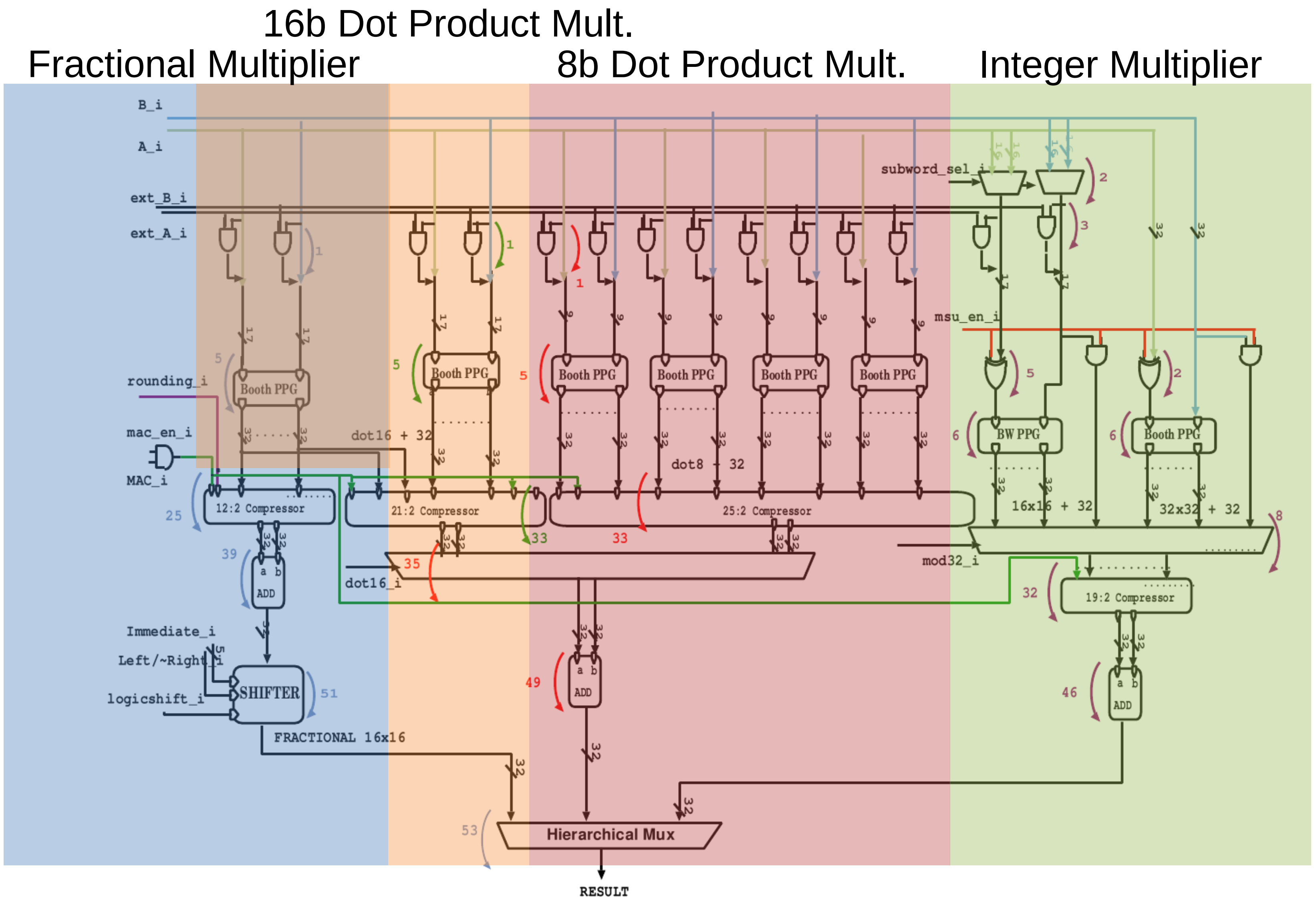


Instruction Extension: Fractional Support

- Fractional format in Q-format [1]
  - Q3.12 means 1 sign bit, 3 integer bits and + 12 fractional bits (16 bit in total)
  - Q3.12 format in a 32 bit reg: - - - - - - - - - - - - - - ---SIIIFFFFFFFFFFFF
- Multiplication in Q format requires a shifter:
  - Q3.12 * Q3.12 = SSIIIIIIFFFFFFFFFFFFFFFFFFFFFFFF
  - Q3.12 * Q3.12 >> 12 = - - - - - - - - - - - - - SSIIIIIIFFFFFFFFFFFF

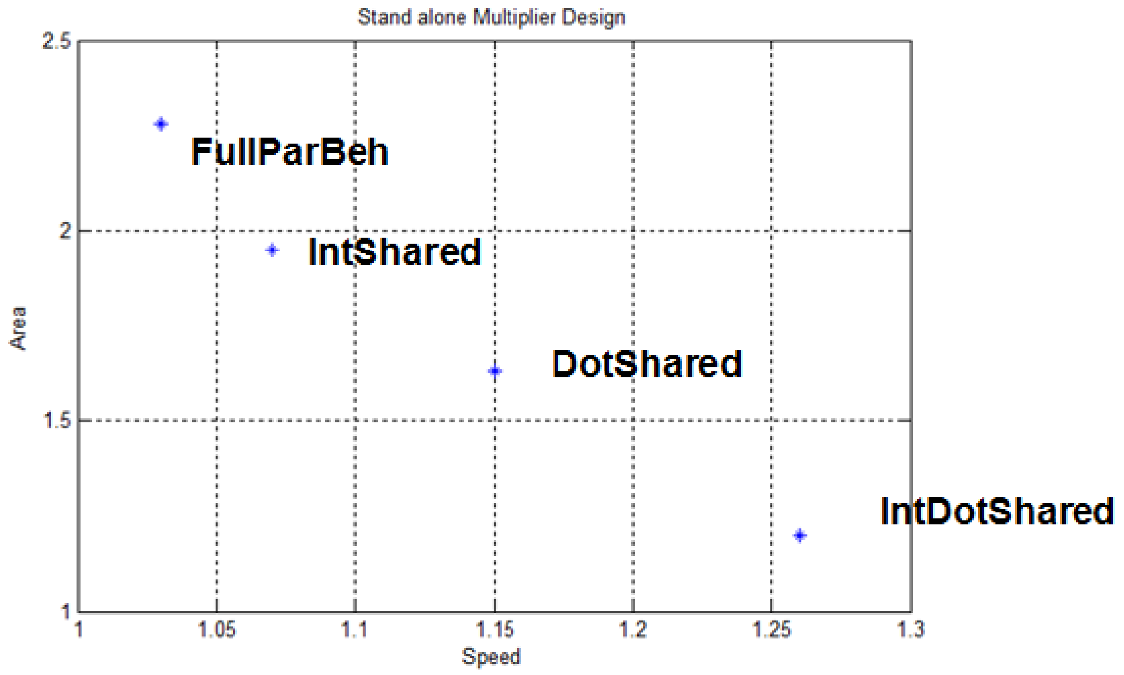


32 bit Vector Support: Full Multiplier Architecture



32 bit Vector Support: Full Multiplier Architecture

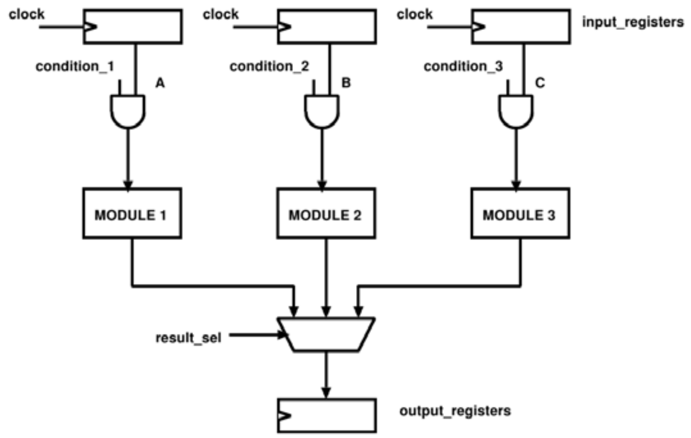
- Supports:
  - Fractional multiplications
  - Dot Products for 16b vectors
  - Dot Products for 16b vectors
  - Integer multiplication 32b*32b into 32b
- Multiplier costs:
  - Up to 2.3x bigger without resource sharing
  - Sharing makes arch. slower
  - Power ?

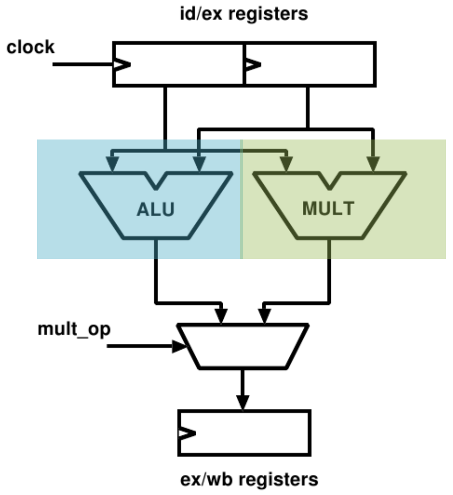


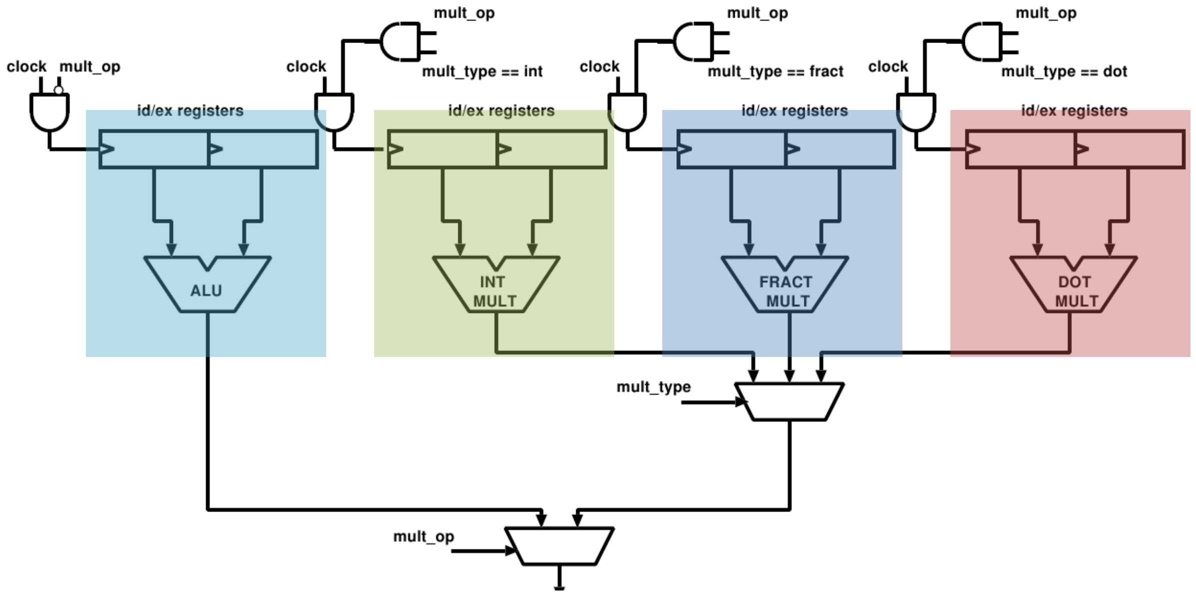
32 bit Vector Support: Power Reduction

- Operand Isolation
  - Gating inputs to zero

- Separate input registers:
  - Eliminating all active power of unused units
  - 30-60% power reduction with respect to a design without separate input registers



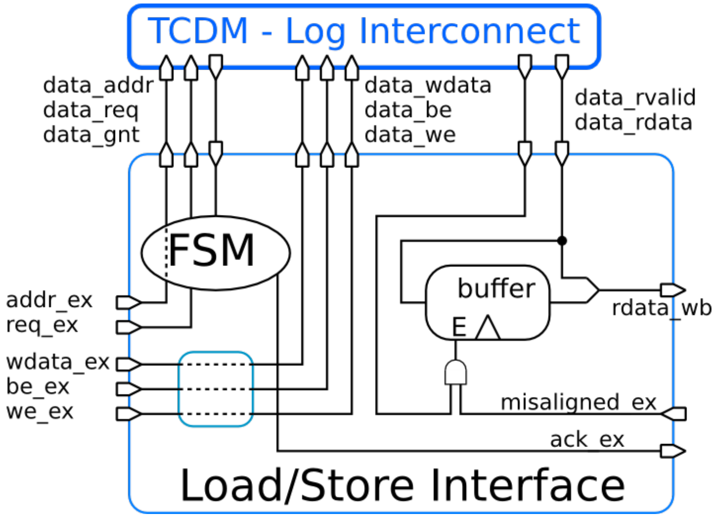




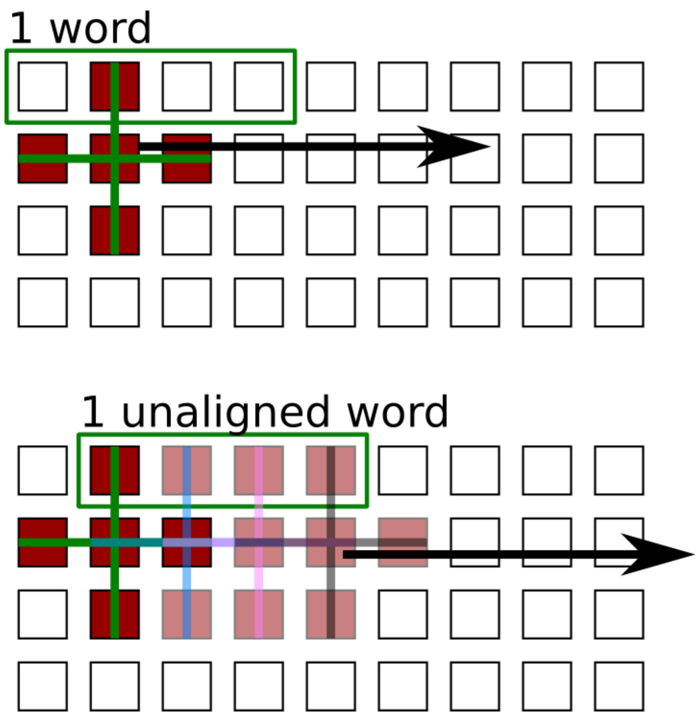
32 bit Vector Support: Unaligned memory access

- Unaligned memory access with 32 bit data interface:
  - Difficult to read/write unaligned words, because memories are 32 bit wide
  - Possible with multibanked memories
    - But significant hardware costs
    - Area and timing

- Implemented with two subsequent memory requests



Example: stencil with vector



32 bit Vector Support: Shuffle Instruction

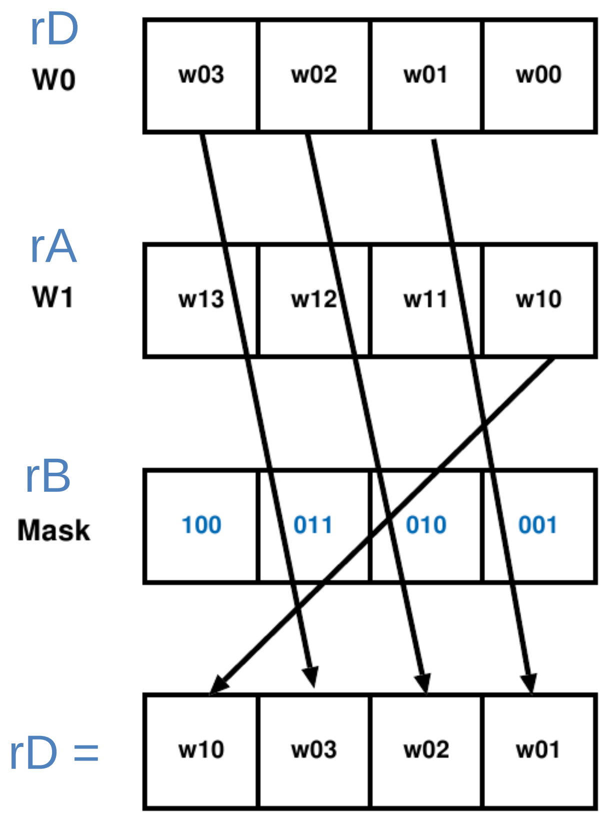
- In order to use the vector unit the elements have to be aligned in the register file
- Shuffle allows to recombine bytes into 1 register:

- lv.shuffle2.b rD, rA, rB

Mask bits

  – rD{3} = (rB[26]==0) ? rA:rD) {rB[25:24]}
  – rD{2} = (rB[18]==0) ? rA:rD) {rB[17:16]}
  – rD{1} = (rB[10]==0) ? rA:rD) {rB[9: 8]}
  – rD{0} = (rB[2]==0) ? rA:rD) {rB[1: 0]}

  – With rX{i} = rX[(i+1)*8-1:i*8]



32 bit Vector Support: Summary

- ALU extensions for packed SIMD are easy
  - Vector addition, comparisons, etc.
- Multiplier is more complicated. Dot product is very useful for simple kernels
  - convolutions, multiplications etc.
- Load store unit needs support for unaligned access
  - Good trade-off with support in two cycles
- Vector recombination instructions are very important for the compiler!
  - To recombine bytes/ halfwords directly in registers instead of reloading everything from memory

The RISC-V ISA

- Modern ISA created by UC Berkeley for their research
- Available for 32-bit, 64-bit and 128-bit
- Little-endian
- Published as Free and Open RISC ISA

- The ISA specifications were previously controlled by UCB, now shifting to the RISC-V foundation

- RISC-V foundation is controlled by the members
Everyone can become member, just costs a bit of money

- See official website http://riscv.org

The RISC-V ISA: Introduction

- Generally kept very simple and extendable
- Separated into multiple specifications
  - User-Level ISA spec (compute instructions)
  - Compressed ISA spec (16-bit instructions)
  - Privileged ISA spec (supervisor-mode instructions)
  - More to come
- Implementations:
  - We have a similar architecture than in OpenRISC
  - Can be used in Semester/Master - Projects

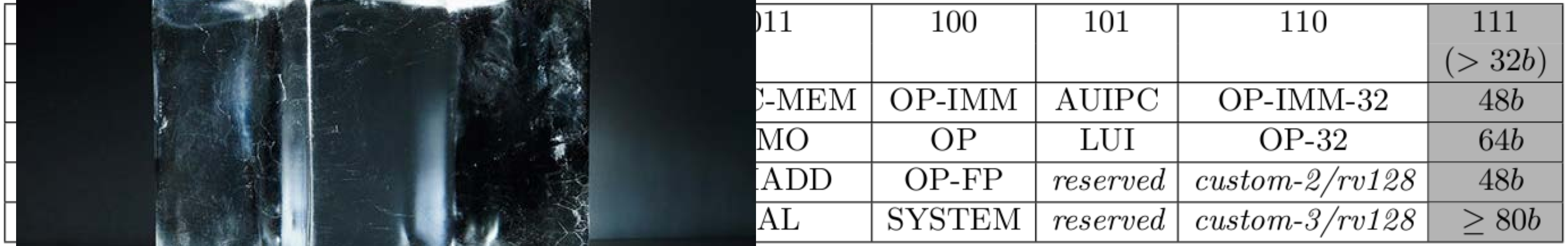
User-Level ISA Spec

- Defines the normal instructions needed for computation
- Spec separated into “extensions”
- Defines a mandatory base integer instruction set: “I extension”
- ISA support is given by RV + word-width + extensions supported
  - E.g. **RV32I** means 32-bit RISC-V with support for the I instruction set

- I: Integer instructions; alu, branches, jumps, loads and stores
  - Support for misaligned memory access is mandatory
- M: Multiplication and (!!!) Division
- A: Atomic instructions
- F: Single-Precision Floating-Point
- D: Double-Precision Floating-Point
- C: Compressed Instructions (more later)

User-Level ISA Spec: Standard and Non-standard extensions

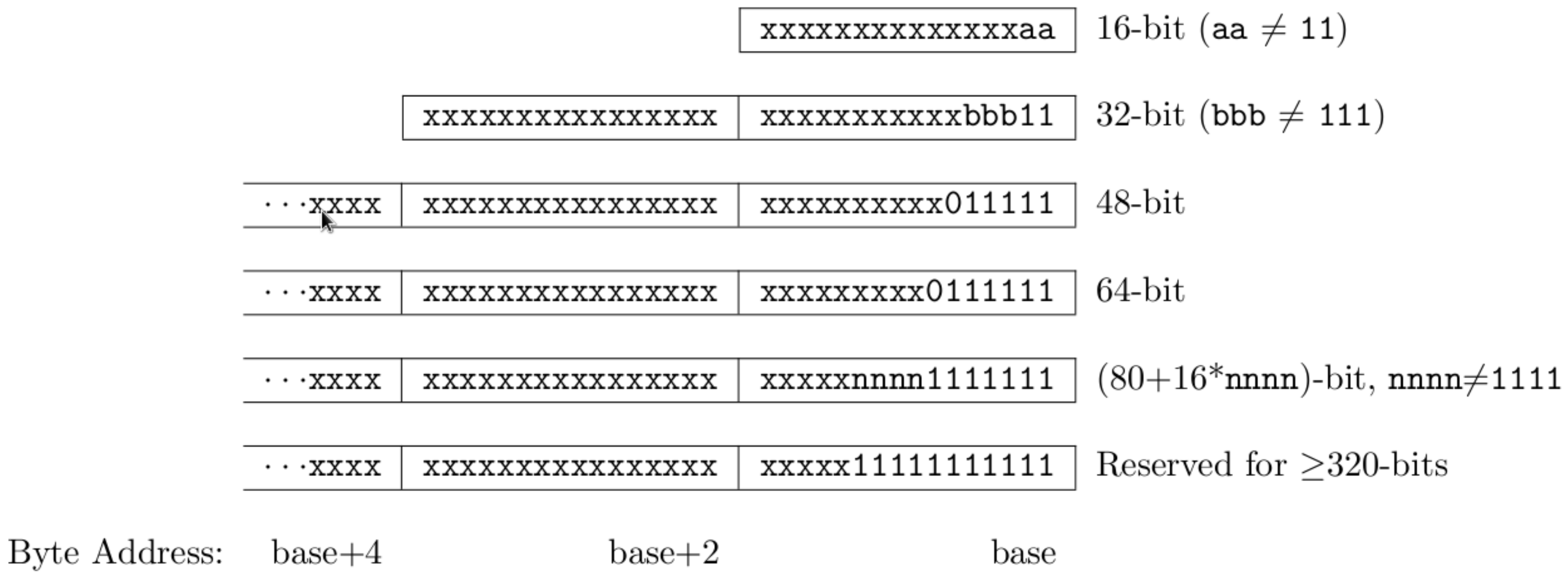
- Extensions mentioned so far are so called **Standard Extensions**
- Reserved opcodes for standard extensions
- Rest of opcodes free for non-standard implementations
- Standard extensions will be frozen and will not change in the future



11	100	101	110	111 ($> 32b$)
-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
MO	OP	LUI	OP-32	$64b$
ADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
AL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

Instruction Length Encoding

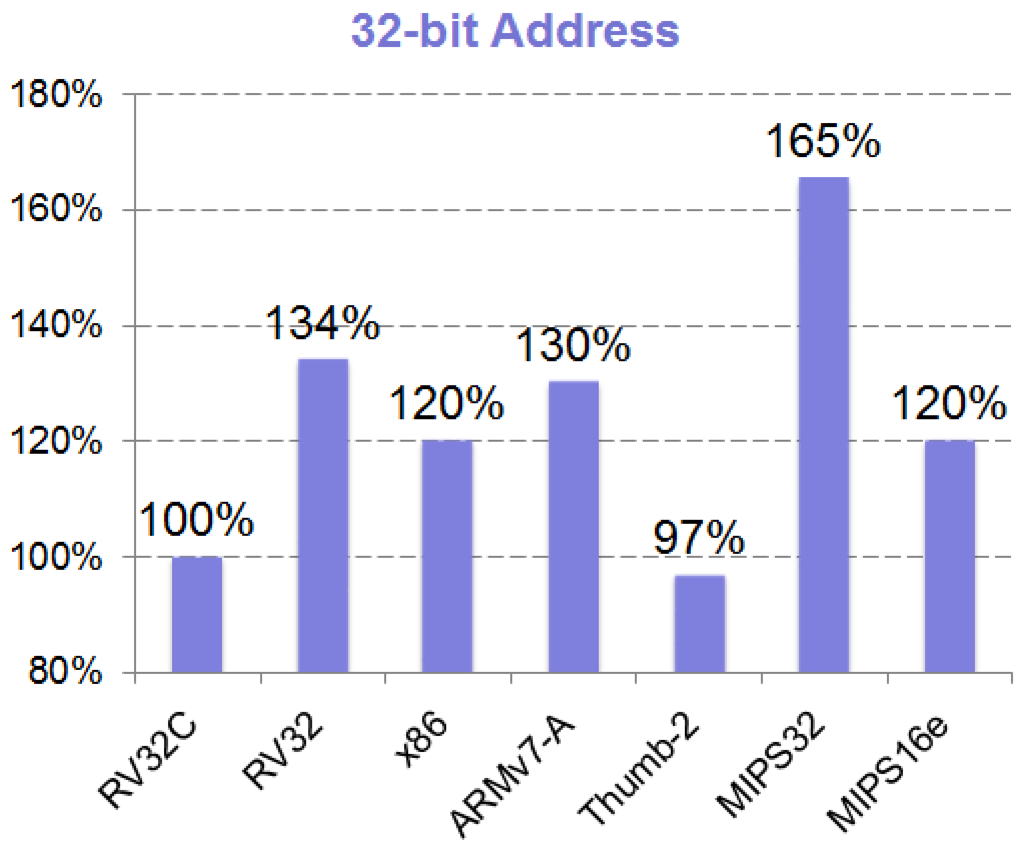
- Supports by design 16, 32, 48, 64, … bit long instruction words



Compressed Instruction Spec (Draft)

- Still a draft, but will be frozen very soon if there are no complaints
- Compressed instructions are 16-bit wide
  - Allows to reduce code size
  - needs support for misaligned instruction memory access
- Compressed instruction ISA spec is no ISA per se
  - All compressed instructions map to I instructions, can be expanded
  - Preprocessing step for instructions needed or separate decoding for compressed instructions

Compressed Instruction Spec (Draft) Claim code size reduction by ~34%



Differences to OpenRISC: No flags, no delay slot

- Branches:

OpenRISC

```
l.sfeq rA, rB
l.bf 0x20
l.nop
```

RISC-V

```
beq rA, rB, 0x20
```

- Jumps are more general

OpenRISC

```
l.jalr rB
```

RISC-V

```
jalr rD, rs1, 0x10
```

Differences to OpenRISC: Opcode Sizes

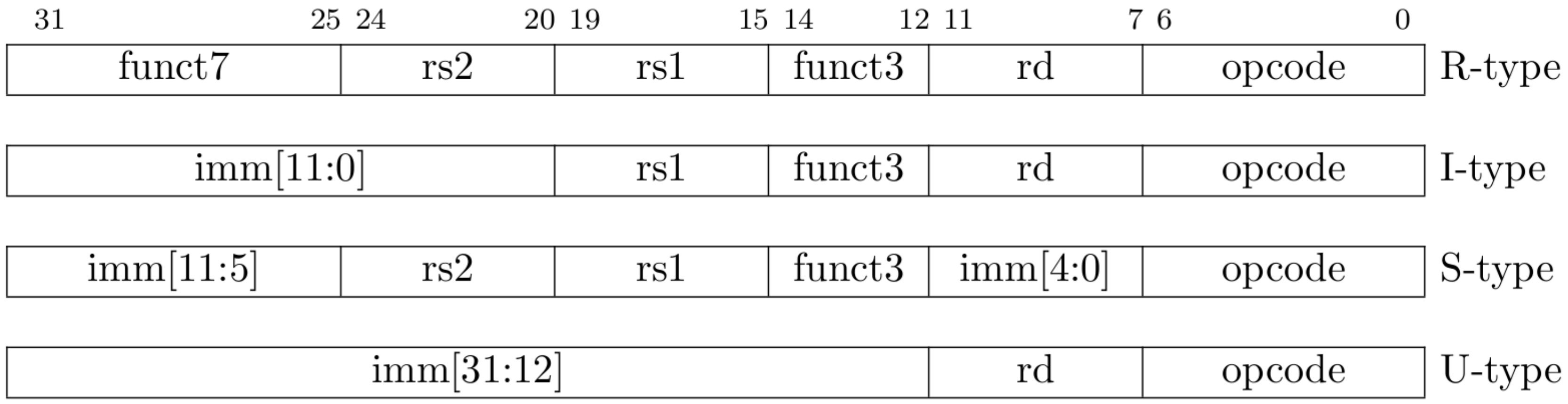
- RISC-V: 7-bit main opcodes
  - actually only 5-bits due to compressed instructions
  - sub-opcodes (funct*) for most instruction types

31 25	24 20	19 15	14 12	11 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[31:12]				rd	opcode	U-type

- OpenRISC: 6-bit main opcodes

Differences to OpenRISC: Immediate Size

- RISC-V: 12-bit immediates, all sign-extended
  - sign-bit always in same position


31
25 24
20 19
15 14
12 11
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[31:12] | rd | opcode
U-type

- OpenRISC: 16-bit immediates, mixed sign- and zero-extended

Differences to OpenRISC: Constructing a 32-bit number

- OpenRISC has 16 bit immediates with zero-extension
  - Just use 2 instructions
- RISC-V has 12 bit immediates with sign-extension
  - load upper immediate first and then add sign-extended value
  - upper immediate is set to imm+1 to correct for sign-extension if necessary

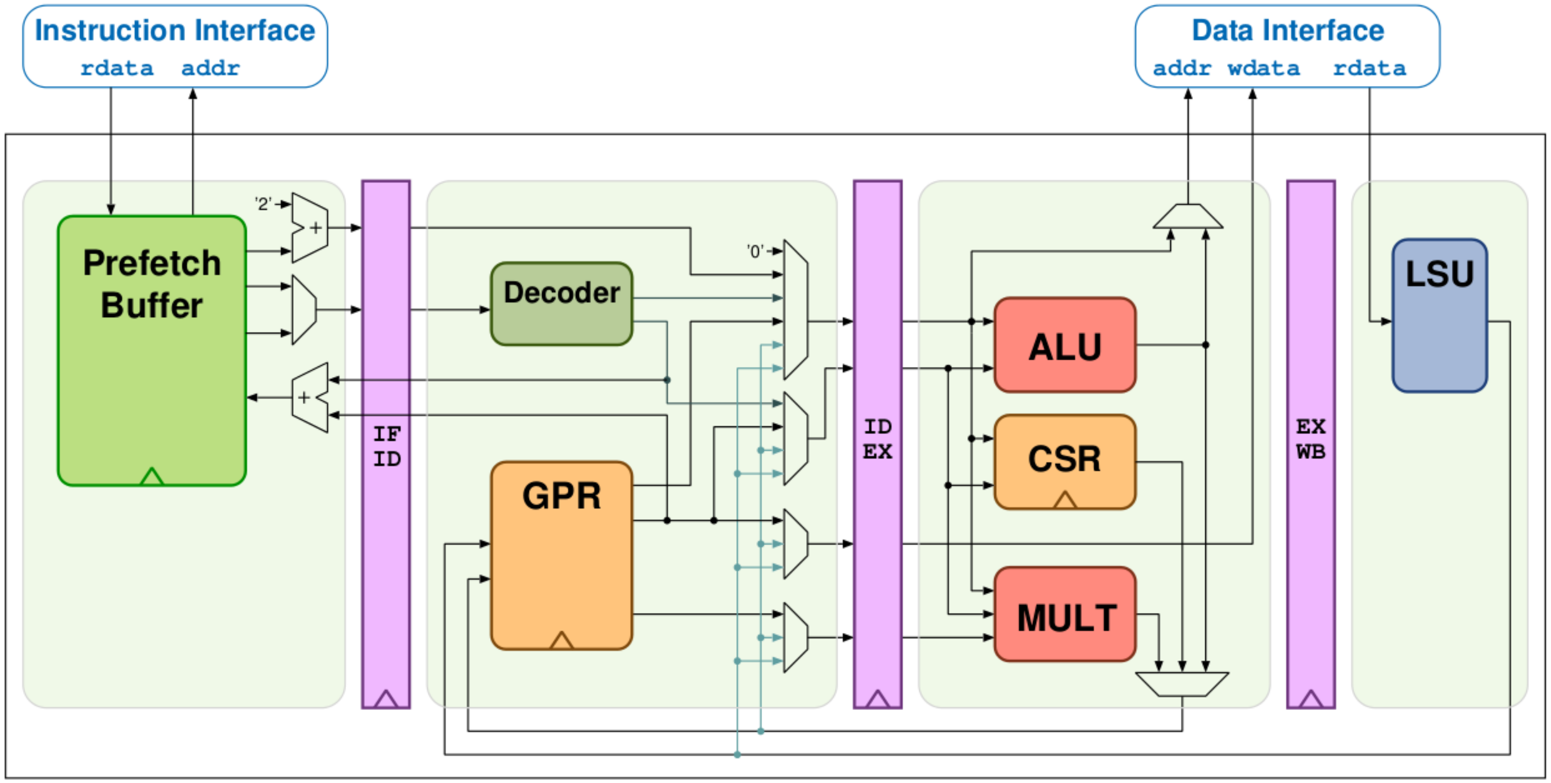
OpenRISC

```
l.movhi r1, 0x1000
l.ori     r1, r1, 0x1800
```

RISC-V

```
lui   x1, 0x10002
add  x1, x1, -2048
```

RI5CY: Core architecture (4 stage pipeline)



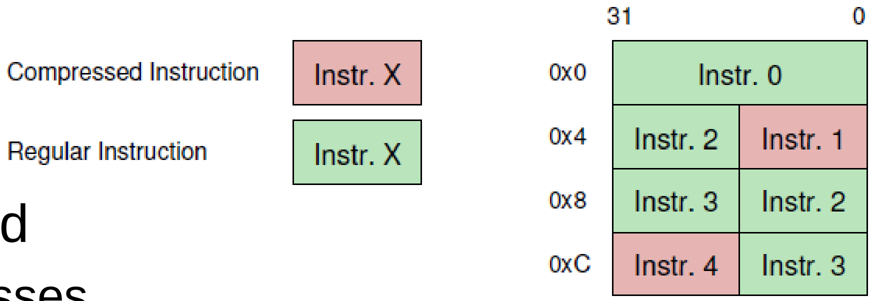
- CSR in EX instead of WB

RI5CY: Prefetcher: Why is branching difficult

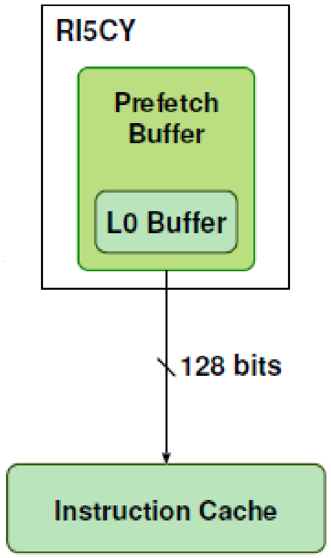
- No delay slot in RISC-V:
  - Jumps loose one cycle
    - Next instruction already fetching and probably ready in IF stage
- Combined branches: No setflag instruction
  - Branch decision computed with branching instruction
  - Branch decision computed in EX stage

  - Taken branches loose two cycles
    - Branch only available late in pipeline
  - Not taken branches don’t loose cycles

RI5CY: Prefetcher: The need for prefetching

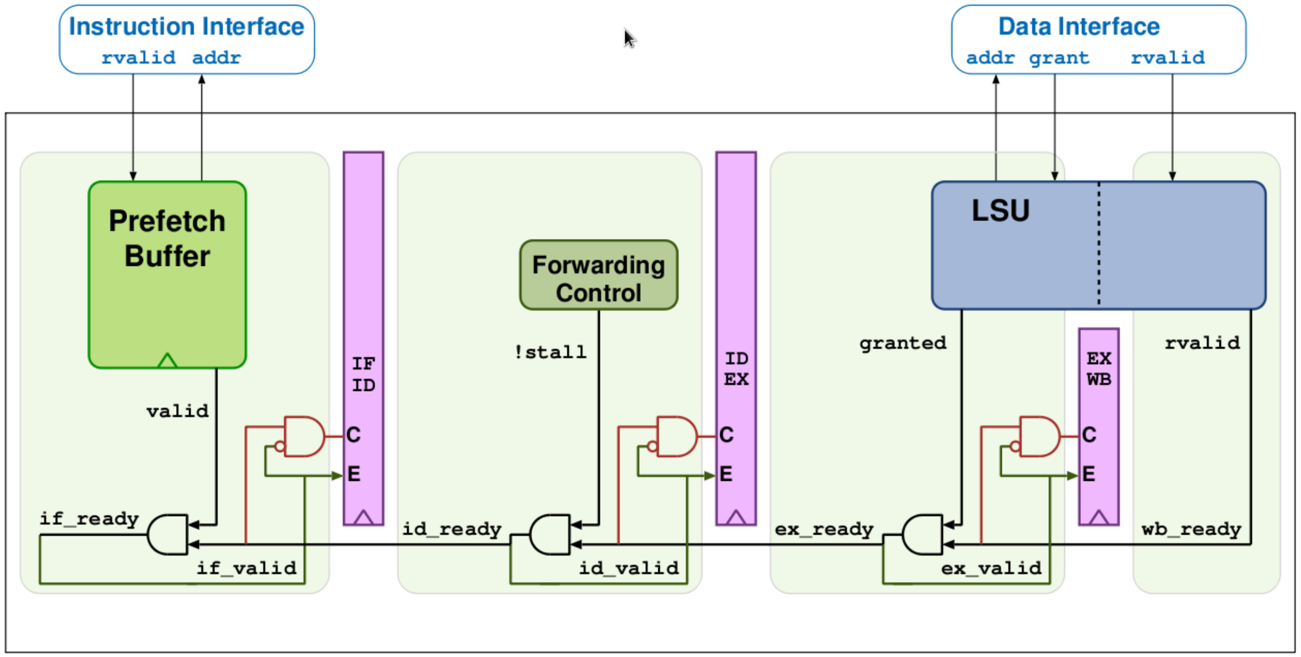


- Instruction memory is word-aligned
  - Does not accept misaligned accesses
- Cross-word instruction needs to be assembled from two words
- If lower half word is compressed, no need to fetch next word already
- Solution: Prefetcher with storage for >2 words
  - We choose 4 words (1 cacheline) for optimal performance and to deal with cache misses



RI5CY: Fully independent pipeline

- Ready & valid signals running left and right respectively
- Pipeline stages can be empty
  - Not possible in OR10N
- Easy to integrate multi-cycle instructions in each stage
  - Similar for "limited" out-of-order execution



RI5CY: Simulation Checker

- Instruction Set Simulator running in parallel using instruction and data access inputs from RTL
- After every cycle write-back data to RF is compared
- ISS serves as golden model
- Could also execute a random instruction stream, since check is automatically and implicitly done

- The simchecker is not activated by default
- To activate:
  - Uncomment **`` `define simchecker ``** in riscv_defines.sv
  - Uncomment sv_lib loading in vsim setup files
  - Build and copy rtl_checker from SDK to vsim work directory

RI5CY: Random Stall Injection

- Only on PULPino for the moment
- Random latency (non-synthesizable) introduced to instruction and data accesses
- Checks core interfaces in a way that we would seldom see on the platform

- To activate
  - Uncomment `define DATA_STALL_RANDOM and `define INSTR_STALL_RANDOM in config.sv of PULPino

RI5CY: Tracer

- Similar to objdump output
- Displays also read and written registers
  - x??:read value
  - x??=written value
  - PA: physical address for memory accesses
- Virtual platform follows same output format
- Files automatically generated for RTL simulation
  - trace_core_xx_xx.log

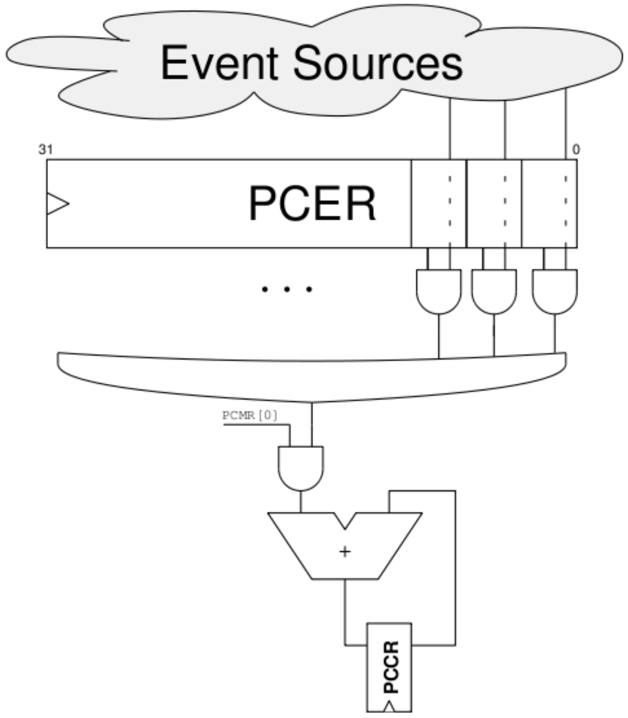
```
Timestamp      Cyc. PC       Instr.   Assembler String       Registers

24530000 ps    1197 0000012c ffaddce3 bge     x27, x26, -8   x27:001002d4 x26:001002d4
24510000 ps    1196 00000128 004d0d13 addi    x26, x26, 4    x26=001002d4 x26:001002d0
24590000 ps    1200 00000124 000d2023 sw      x0, 0(x26)     x26:001002d4  PA:001002d4
24630000 ps    1202 0000012c ffaddce3 bge     x27, x26, -8   x27:001002d4 x26:001002d8
24610000 ps    1201 00000128 004d0d13 addi    x26, x26, 4    x26=001002d8 x26:001002d4
24650000 ps    1203 00000130 00000513 addi    x10, x0, 0     x10=00000000
24670000 ps    1204 00000134 00100593 addi    x11, x0, 1     x11=00000001
```

RI5CY: Performance Counters

- Events to be counted:
  - #cycles
  - #instructions
  - #ld_stall: load data hazards
  - #jr_stall: number of jump register data hazards
  - #imiss: cycles waiting for instructions
  - #ld
  - #st
  - #jump
  - #branch: total number of branches (w/o jumps)
  - #btaken: branches that were taken
  - #rvc: number of rvc insns
  - #ld_ext: LD to non-tcdm
    - misaligned access counted twice
  - #st_ext: ST to non-tcdm
  - #ld_ext_cyc
  - #st_ext_cyc
  - #tcdm_cont: cycles wasted due to waiting for grants in L1

RI5CY: Performance Counters

- On ASIC only one counter + one register
- On FPGA/RTL sim
  - one counter + one register per metric
- Binary tracer for performance counters is not yet available (for KCG)



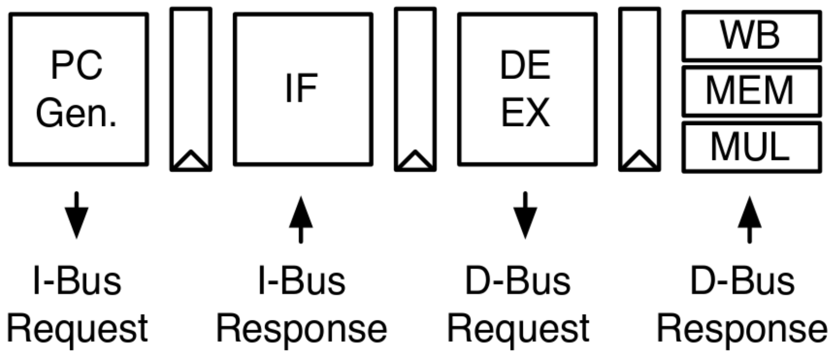
Other RISC-V Cores Z-Scale / V-Scale

- From UC Berkeley
- Their take on a small core
- Z-Scale: Written in Chisel
- V-Scale: Written in Verilog
- Z-Scale & V-Scale virtually identical

Mini project:
Compare V-Scale/Z-Scale to our core [1]

Semester project:
Design a mini core with lower power consumption than V-Scale, Z-Scale

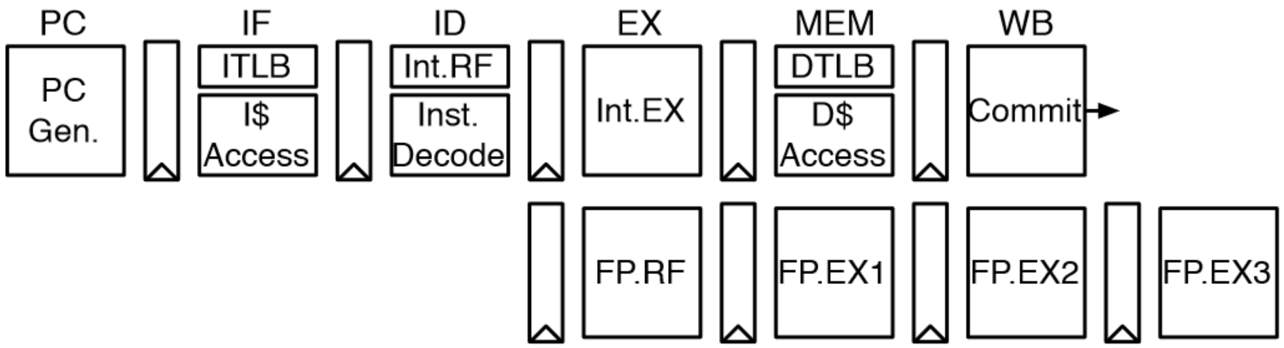
Z-scale Pipeline



Category	ARM Cortex-M0	RISC-V Zscale
ISA	32-bit ARM v6	32-bit RISC-V (RV32IM)
Architecture	Single-Issue In-Order 3-stage	Single-Issue In-Order 3-stage
Performance	0.87 DMIPS/MHz	1.35 DMIPS/MHz
Process	TSMC 40LP	TSMC 40GPLUS
Area w/o Caches	0.0070 mm^2	0.0098 mm^2
Area Efficiency	124 DMIPS/MHz/mm^2	138 DMIPS/MHz/mm^2
Frequency	≤50 MHz	~500 MHz
Voltage (RTV)	1.1 V	0.99 V
Dynamic Power	5.1 μW/MHz	1.8 μW/MHz

Other RISC-V Cores: Rocket

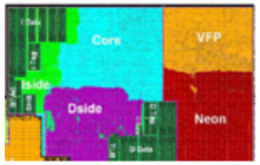
- From UC Berkeley, written in Chisel
- 64-Bit Implementation



Category	ARM Cortex-A5	RISC-V Rocket
ISA	32-bit ARM v7	64-bit RISC-V v2
Architecture	Single-Issue In-Order	Single-Issue In-Order 5-stage
Performance	1.57 DMIPS/MHz	1.72 DMIPS/MHz
Process	TSMC 40GPLUS	TSMC 40GPLUS
Area w/o Caches	0.27 mm^2	0.14 mm^2
Area with 16K Caches	0.53 mm^2	0.39 mm^2
Area Efficiency	2.96 DMIPS/MHz/mm^2	4.41 DMIPS/MHz/mm^2
Frequency	>1GHz	>1GHz
Dynamic Power	<0.08 mW/MHz	0.034 mW/MHz

Other RISC-V Cores
BOOM: Berkeley Out-of-Order Processor

- From UC Berkeley, written in Chisel
- Parametrizable for Dual-Issue/Quad-Issue

Category	ARM Cortex-A9	RISC-V BOOM-2w
ISA	32-bit ARM v7	64-bit RISC-V v2 (RV64G)
Architecture	2 wide, 3+1 issue Out-of-Order 8-stage	2 wide, 3 issue Out-of-Order 6-stage
Performance	**3.59** CoreMarks/MHz	**3.91** CoreMarks/MHz
Process	TSMC 40GPLUS	TSMC 40GPLUS
Area with 32K caches	~2.5 mm^2	~1.00 mm^2
Area efficiency	**1.4** CoreMarks/MHz/mm^2	**3.9** CoreMarks/MHz/mm^2
Frequency	1.4 GHz	1.5 GHz
Power	**0.5-1.9 W** (2 cores + L2) @ TSMC 40nm, 0.8-2.0 GHz	**0.25 W** (1 core + L1) @ TSMC 45nm, 1 GHz
note: not to scale		

+9%!

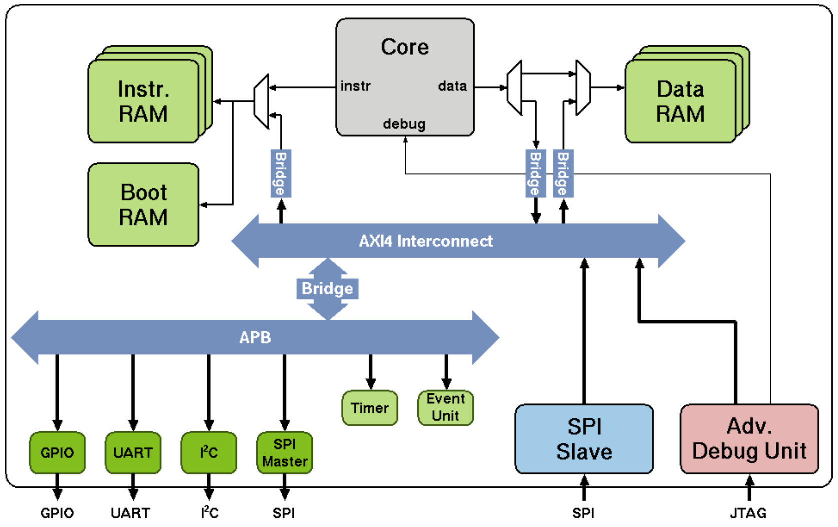
Master project: [1]
Design and implement a VLIW-architecture supporting the RISC-V ISA

Mini Project:
Initial design consideration for the implementation (pro/cons evaluation)

Exercise session:

- In the exercise we are going to cover:
  - How to compile and run an application using:
    - The open source Pulpino platform
  - Impact of the new instructions:
    - Hardware loops
    - Pre/post increment
    - Vector support
    - Dot product
    - Shuffle instruction
  - How to use highly optimized kernels in programs
    - Convolutions
  - Comparison of RISC-V to ARM Cortex M4

Pulpino architecture



Q&A

?