The Libre-SOC Hybrid 3D CPU

Augmenting the OpenPOWER ISA
to provide 3D and Video instructions
(properly and officially)

[proposed for] OpenPOWER Summit 2020

Sponsored by NLnet's PET Programme

September 10, 2020

Why another SoC?

- Intel Management Engine, QA issues, Spectre
- Endless proprietary drivers
  (affects product development cost)
- Opportunity to drastically simplify driver development
  and engage in "long-tail" markets
- Because for 30 years I Always Wanted To Design A CPU

Why OpenPOWER? (but first: Evaluation Criteria)

- Good ecosystem essential
  linux kernel, u-boot, compilers, OSes,
  Reference Implementation(s)
- Supportive Foundation and Members
  need to be able to submit ISA augmentations
  (for proper peer review)
- No NDAs, full transparency must be acceptable
  due to being funded under NLnet's PET Programme

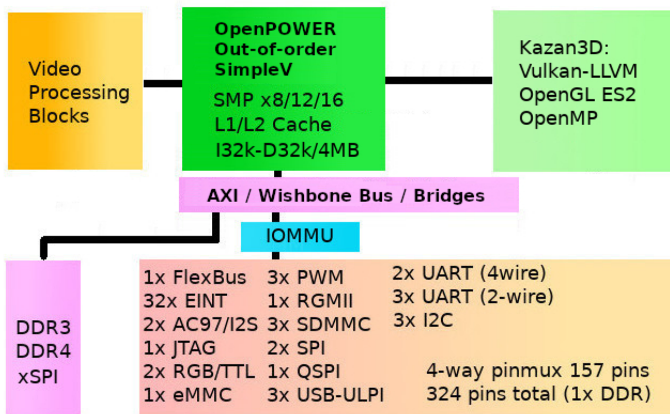
Why OpenPOWER?

- RISC-V: closed secretive mailing lists, closed secretive ISA Working Groups, no acceptance of transparency requirements, not well-established enough
- MIPS Open Initiative website was offline
- ARM and x86 are proprietary (x86 too complex)
- OpenRISC 1200 not enough adoption
- Nyuzi GPU too specialist (not a general-purpose ISA)
- MIAOW GPU is not a GPU (it's an AMD Vector Engine)
- "rolling your own" out of the question (20+ man-years)
- OpenPOWER: established for decades, excellent Foundation, Microwatt as Reference, approachable and friendly.

What goes into a typical SoC?

- 15 to 20mm BGA package: 2.5 to 5 watt power consumption heat sink normally not required (simplifies overall design)
- Fully-integrated peripherals (not Northbridge/Southbridge) USB, HDMI, RGB/TTL, SD/MMC, I2C, UART, SPI, GPIO etc. etc.
- Built-in GPU (shared memory bus, 3rd party licensed)
- Build-in VPU (likewise)
- Target price between $2.50 and $30 depending on market Radically different from IBM POWER9 Core (200 Watt)

Simple SBC-style SoC



Where to start? (roadmap)

- First thing: get a basic core working on an FPGA (use Microwatt as a reference)
- Next: create a low-cost test ASIC (180nm). (first OpenPOWER ASIC since IBM's POWER9, 10 years ago)
- (in parallel): Develop Vector ISA with 3D and Video extensions, under watchful eye of OpenPOWER Foundation
- Implement Vector ISA in simulator, then HDL, then FPGA and finally (only when ratified by OPF) into silicon
- Sell chips, make $$$.

What's different about Libre-SOC?

- Hybrid - integrated. The CPU *is* the GPU.
  The GPU *is* the CPU. The VPU *is* the CPU.
  There is No Separate VPU/GPU Pipeline
- written in nmigen (a python-based HDL). Not VHDL
  not Verilog (definitely not Chisel3/Scala)
  This is an extremely important strategic decision.
- Simple-V Vector Extension. See "SIMD Considered harmful".
  SV effectively a "hardware for-loop" on standard scalar ISA
  (conceptually similar to Zero-Overhead Loops in DSPs)

Hybrid Architecture: Augmented 6600

- CDC 6600 is a design from 1965. The *augmentations* are not. Help from Mitch Alsup includes *precise exceptions*, multi-issue and more. Academic literature on 6600 utterly misleading. 6600 Scoreboards completely underestimated (Seymour Cray and James Thornton solved problems they didn't realise existed elsewhere!)
- Front-end Vector ISA, back-end "Predicated (masked) SIMD" nmigen (python OO) strategically critical to achieving this.
- Out-of-order combined with Simple-V allows scalar operations at the developer end to be turned into SIMD at the back-end *without the developer needing to do SIMD*
- IEEE754 sin / cos / atan2, Texturisation opcodes, YUV2RGB all automatically vectorised.

Why nmigen? (but first: evaluate other HDLs)

- Verilog: designed in the 1980s purely for doing unit tests (!)
- VHDL: again, a 1980s-era "Procedural" language (BASIC, Fortran). Does now have "records" which is nice.
- Chisel3 / Scala: OO, but very obscure (20th on index)
- pyrtl: not large enough community
- MyHDL: subset of python only

- Slowly forming a set of criteria: must be OO (python), must have wide adoption (python), must have good well-established programming practices already in place (python), must be easy to learn (python)
- HDL itself although a much smaller community must have the same criteria. Only nmigen meets that criteria.

Why nmigen?

- Uses python to build an AST (Abstract Syntax Tree). Actually hands that over to yosys (to create ILANG file) after which verilog can (if necessary) be created
- Deterministic synthesiseable behaviour (Signals are declared with their reset pattern: no more forgetting "if rst" block).
- python OO programming techniques can be deployed. classes and functions created which pass in parameters which change what HDL is created (IEEE754 FP16 / 32 / 64 for example)
- python-based for-loops can e.g. read CSV files then generate a hierarchical nested suite of HDL Switch / Case statements (this is how the Libre-soc PowerISA decoder is implemented)
- extreme OO abstraction can even be used to create "dynamic partitioned Signals" that have the same operator-overloaded "add", "subtract", "greater-than" operators

nmigen (dynamic) vs VHDL (static)

```python
power_op_types = {'function_unit': Function,
                  'internal_op': MicrOp,
                  'form': Form,
                  'asmcode': 8,
                  'in1_sel': In1Sel,
                  'in2_sel': In2Sel,
                  'in3_sel': In3Sel,
                  'out_sel': OutSel,
                  'cr_in': CRInSel,
                  'cr_out': CROutSel,
                  'ldst_len': LdstLen,
                  'upd': LDSTMode,
                  'rc_sel': RC,
                  'cry_in': CryIn
                  }

class PowerOp:
    def __init__(self, incl_asm=True, name=None, subset=None):
        self.subset = subset
        debug_report = set()
        fields = set()
        for field, ptype in power_op_types.items():
            fields.add(field)
            if subset and field not in subset:
                continue
            fname = get_pname(field, name)
            setattr(self, field, Signal(ptype, reset_less=True, name=fname))
            debug_report.add(field)
        for bit in single_bit_flags:
            field = get_signal_name(bit)
            fields.add(field)
            if subset and field not in subset:
                continue
            debug_report.add(field)
            fname = get_pname(field, name)
            setattr(self, field, Signal(reset_less=True, name=fname))
        print ("PowerOp debug", name, debug_report)
        print ("        fields", fields)

    def _eq(self, row=None):
        if row is None:
            row = default_values
        # TODO: this conversion process from a dict to an object
        # should really be done using e.g. namedtuple and then
        # call eq not _eq
```

```vhdl
type unit_t is (NONE, ALU, LDST);
type length_t is (NONE, is1B, is2B, is4B, is8B);

type decode_rom_t is record
    unit          : unit_t;
    insn_type     : insn_type_t;
    input_reg_a   : input_reg_a_t;
    input_reg_b   : input_reg_b_t;
    input_reg_c   : input_reg_c_t;
    output_reg_a  : output_reg_a_t;

    input_cr      : std_ulogic;
    output_cr     : std_ulogic;

    invert_a      : std_ulogic;
    invert_out    : std_ulogic;
    input_carry   : carry_in_t;
    output_carry  : std_ulogic;

    -- load/store signals
    length        : length_t;
    byte_reverse  : std_ulogic;
    sign_extend   : std_ulogic;
    update        : std_ulogic;
    reserve       : std_ulogic;

    -- multiplier and ALU signals
    is_32bit      : std_ulogic;
    is_signed     : std_ulogic;

    rc            : rc_t;
    lr            : std_ulogic;

    sgl_pipe      : std_ulogic;
end record;
```

nmigen PowerISA Decoder

lkcl@fizzy: ~/src/libresoc/microwatt/tests/xics

```
-- Note: reformat with column -t -o ' '
constant decode_op_31_array : op_31_subop_array_t := (
    --                      unit   internal   in1     in2        in3   out   CR  CR  inv inv cry  cry ldst BR  sgn upd rsrv 32b sgn rc   lk  sgl
    --                             op                                        in  out A   out in   out len      ext                            pipe
    2#0100001010#  =>       (ALU,  OP_ADD,    RA,     RB,        NONE, RT,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- add
    2#1100001010#  =>       (ALU,  OP_ADD,    RA,     RB,        NONE, RT,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- addo
    2#0000001010#  =>       (ALU,  OP_ADD,    RA,     RB,        NONE, RT,   '0', '0', '0', '0', ZERO, '1', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- addc
    2#1000001010#  =>       (ALU,  OP_ADD,    RA,     RB,        NONE, RT,   '0', '0', '0', '0', ZERO, '1', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- addco
    2#0010001010#  =>       (ALU,  OP_ADD,    RA,     RB,        NONE, RT,   '0', '0', '0', '0', CA,   '1', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- adde
    2#1010001010#  =>       (ALU,  OP_ADD,    RA,     RB,        NONE, RT,   '0', '0', '0', '0', CA,   '1', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- addeo
    2#0010101010#  =>       (ALU,  OP_ADD,    RA,     RB,        NONE, RT,   '0', '0', '0', '0', OV,   '1', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- addex
    2#0001001010#  =>       (ALU,  OP_ADDG6S, RA,     RB,        NONE, RT,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0'), -- addg6s
    2#0011101010#  =>       (ALU,  OP_ADD,    RA,     CONST_M1,  NONE, RT,   '0', '0', '0', '0', CA,   '1', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- addme
    2#1011101010#  =>       (ALU,  OP_ADD,    RA,     CONST_M1,  NONE, RT,   '0', '0', '0', '0', CA,   '1', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- addmeo
    2#0011001010#  =>       (ALU,  OP_ADD,    RA,     NONE,      NONE, RT,   '0', '0', '0', '0', CA,   '1', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- addze
    2#1011001010#  =>       (ALU,  OP_ADD,    RA,     NONE,      NONE, RT,   '0', '0', '0', '0', CA,   '1', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- addzeo
    2#0000011100#  =>       (ALU,  OP_AND,    NONE,   RB,        RS,   RA,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- and
    2#0000111100#  =>       (ALU,  OP_AND,    NONE,   RB,        RS,   RA,   '0', '0', '1', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- andc
    2#0011111100#  =>       (ALU,  OP_BPERM,  NONE,   NONE,      RS,   RA,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0'), -- bperm
    2#0100111010#  =>       (ALU,  OP_BCD,    NONE,   NONE,      RS,   RA,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0'), -- cbcdtd
    2#0100011010#  =>       (ALU,  OP_BCD,    NONE,   NONE,      RS,   RA,   '0', '0', '1', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0'), -- cdtbcd
    2#0000000000#  =>       (ALU,  OP_CMP,    RA,     RB,        NONE, NONE, '0', '1', '1', '0', ONE,  '0', NONE, '0', '0', '0', '0', '0', '1', NONE, '0', '0'), -- cmp
    2#0111111100#  =>       (ALU,  OP_CMPB,   NONE,   RB,        RS,   RA,   '0', '1', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0'), -- cmpb
    2#0011100000#  =>       (ALU,  OP_CMPEQB, RA,     RB,        NONE, NONE, '0', '1', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0'), -- cmpeqb
    2#0000100000#  =>       (ALU,  OP_CMP,    RA,     RB,        NONE, NONE, '0', '1', '1', '0', ONE,  '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0'), -- cmpl
    2#0011000000#  =>       (ALU,  OP_CMPRB,  RA,     RB,        NONE, NONE, '0', '1', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0'), -- cmprb
    2#0000111010#  =>       (ALU,  OP_CNTZ,   NONE,   NONE,      RS,   RA,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- cntlzd
    2#0000011010#  =>       (ALU,  OP_CNTZ,   NONE,   NONE,      RS,   RA,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '1', '0', RC,   '0', '0'), -- cntlzw
    2#1000111010#  =>       (ALU,  OP_CNTZ,   NONE,   NONE,      RS,   RA,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', RC,   '0', '0'), -- cnttzd
    2#1000011010#  =>       (ALU,  OP_CNTZ,   NONE,   NONE,      RS,   RA,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '1', '0', RC,   '0', '0'), -- cnttzw
    2#1011110011#  =>       (ALU,  OP_DARN,   NONE,   NONE,      NONE, RT,   '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '0'), -- darn
    2#0001010110#  =>       (ALU,  OP_NOP,    NONE,   NONE,      NONE, NONE, '0', '0', '0', '0', ZERO, '0', NONE, '0', '0', '0', '0', '0', '0', NONE, '0', '1'), -- dcbf
```

lkcl@fizzy: ~/src/libresoc/soc/libreriscv/openpower/isatables

```
0b0011000000,,,,,,,,,,,,,,,,,,,,,,,,cmprb,X
0b0000111010,LOGICAL,OP_CNTZ,RS,NONE,NONE,RA,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,0,0,RC,0,0,cntlzd,X
0b0000011010,LOGICAL,OP_CNTZ,RS,NONE,NONE,RA,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,1,0,RC,0,0,cntlzw,X
0b1000111010,LOGICAL,OP_CNTZ,RS,NONE,NONE,RA,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,0,0,RC,0,0,cnttzd,X
0b1000011010,LOGICAL,OP_CNTZ,RS,NONE,NONE,RA,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,1,0,RC,0,0,cnttzw,X
0b1011110011,,,,,,,,,,,,,,,,,,,,,,,,darn,X
0b0001010110,ALU,OP_NOP,NONE,NONE,NONE,NONE,NONE,NONE,0,0,ZERO,0,NONE,0,0,0,0,0,0,NONE,0,1,dcbf,X
0b0000110110,ALU,OP_NOP,NONE,NONE,NONE,NONE,NONE,NONE,0,0,ZERO,0,NONE,0,0,0,0,0,0,NONE,0,1,dcbst,X
0b0100010110,ALU,OP_NOP,NONE,NONE,NONE,NONE,NONE,NONE,0,0,ZERO,0,NONE,0,0,0,0,0,0,NONE,0,1,dcbt,X
0b0011110110,ALU,OP_NOP,NONE,NONE,NONE,NONE,NONE,NONE,0,0,ZERO,0,NONE,0,0,0,0,0,0,NONE,0,1,dcbtst,X
0b1111110110,,,,,,,,,,,,,,,,,,,,,,,,dcbz,X
0b0110001001,DIV,OP_DIVE,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,0,0,RC,0,0,divdeu,XO
0b1110001001,DIV,OP_DIVE,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,0,0,RC,0,0,divdeuo,XO
0b0110001011,DIV,OP_DIVE,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,1,0,RC,0,0,divweu,XO
0b1110001011,DIV,OP_DIVE,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,1,0,RC,0,0,divweuo,XO
0b0110101001,DIV,OP_DIVE,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,0,1,RC,0,0,divde,XO
0b1110101001,DIV,OP_DIVE,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,0,1,RC,0,0,divdeo,XO
0b0110101011,DIV,OP_DIVE,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,1,1,RC,0,0,divwe,XO
0b1110101011,DIV,OP_DIVE,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,1,1,RC,0,0,divweo,XO
0b0111001001,DIV,OP_DIV,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,0,0,RC,0,0,divdu,XO
0b1111001001,DIV,OP_DIV,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,0,0,RC,0,0,divduo,XO
0b0111001011,DIV,OP_DIV,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,1,0,RC,0,0,divwu,XO
0b1111001011,DIV,OP_DIV,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,1,0,RC,0,0,divwuo,XO
0b0111101001,DIV,OP_DIV,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,0,1,RC,0,0,divd,XO
0b1111101001,DIV,OP_DIV,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,0,1,RC,0,0,divdo,XO
0b0111101011,DIV,OP_DIV,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,1,1,RC,0,0,divw,XO
0b1111101011,DIV,OP_DIV,RA,RB,NONE,RT,NONE,CR0,0,0,ZERO,0,NONE,0,0,0,0,1,1,RC,0,0,divwo,XO
0b0100011100,LOGICAL,OP_XOR,RS,RB,NONE,RA,NONE,CR0,0,1,ZERO,0,NONE,0,0,0,0,0,0,RC,0,0,eqv,X
0b1110111010,ALU,OP_EXTS,RS,NONE,NONE,RA,NONE,CR0,0,0,ZERO,0,is1B,0,0,0,0,0,0,RC,0,0,extsb,X
0b1110011010,ALU,OP_EXTS,RS,NONE,NONE,RA,NONE,CR0,0,0,ZERO,0,is2B,0,0,0,0,0,0,RC,0,0,extsh,X
0b1111011010,ALU,OP_EXTS,RS,NONE,NONE,RA,NONE,CR0,0,0,ZERO,0,is4B,0,0,0,0,0,0,RC,0,0,extsw,X
```

nmigen PowerISA Decoder

```python
########################################################
# PRIMARY FUNCTION SPECIFYING THE FULL POWER DECODER

def create_pdecode(name=None, col_subset=None, row_subset=None):
    """create_pdecode - creates a cascading hierarchical POWER ISA decoder

    subsetting of the PowerOp decoding is possible by setting col_subset
    """

    # minor 19 has extra patterns
    m19 = []
    m19.append(Subdecoder(pattern=19, opcodes=get_csv("minor_19.csv"),
                          opint=True, bitsel=(1, 11), suffix=None,
                          subdecoders=[]))
    m19.append(Subdecoder(pattern=19, opcodes=get_csv("minor_19_00000.csv"),
                          opint=True, bitsel=(1, 6), suffix=None,
                          subdecoders=[]))

    # minor opcodes.
    pminor = [
        m19,
        Subdecoder(pattern=30, opcodes=get_csv("minor_30.csv"),
                   opint=True, bitsel=(1, 5), suffix=None, subdecoders=[]),
        Subdecoder(pattern=31, opcodes=get_csv("minor_31.csv"),
                   opint=True, bitsel=(1, 11), suffix=0b00101, subdecoders=[]),
        Subdecoder(pattern=58, opcodes=get_csv("minor_58.csv"),
                   opint=True, bitsel=(0, 2), suffix=None, subdecoders=[]),
        Subdecoder(pattern=62, opcodes=get_csv("minor_62.csv"),
                   opint=True, bitsel=(0, 2), suffix=None, subdecoders=[]),
    ]

    # top level: extra merged with major
    dec = []
    opcodes = get_csv("major.csv")
    dec.append(Subdecoder(pattern=None, opint=True, opcodes=opcodes,
                          bitsel=(26, 32), suffix=None, subdecoders=pminor))
    opcodes = get_csv("extra.csv")
    dec.append(Subdecoder(pattern=None, opint=False, opcodes=opcodes,
                          bitsel=(0, 32), suffix=None, subdecoders=[]))

    return TopPowerDecoder(32, dec, name=name, col_subset=col_subset,
                                    row_subset=row_subset)
```

Why another Vector ISA? (or: not-exactly another)

- Simple-V is a 'register tag' system. *There are no opcodes* SV 'tags' scalar operations (scalar regfiles) as 'vectorised'
- (PowerISA SIMD is around 700 opcodes, making it unlikely to be able to fit a PowerISA decoder in only one clock cycle)
- Effectively a 'hardware sub-counter for-loop': pauses the PC then rolls incrementally through the operand register numbers issuing *multiple* scalar instructions into the pipelines (hence the reason for a multi-issue OoO microarchitecture)
- Current *and future* PowerISA scalar opcodes inherently *and automatically* become 'vectorised' by SV without needing an explicit new Vector opcode.
- Predication and element width polymorphism are also 'tags'. elwidth polymorphism allows for FP16 / 80 / 128 to be added to the ISA *without modifying the ISA*

Simple-V ADD in a nutshell

```
function op_add(rd, rs1, rs2, predr) # add not VADD!
 int i, id=0, irs1=0, irs2=0;
 for (i = 0; i < VL; i++)
   if (ireg[predr] & 1<<i) # predication uses intregs
     ireg[rd+id] <= ireg[rs1+irs1] + ireg[rs2+irs2];
    if (reg_is_vectorised[rd] ) { id += 1; }
    if (reg_is_vectorised[rs1]) { irs1 += 1; }
    if (reg_is_vectorised[rs2]) { irs2 += 1; }
```

- Above is oversimplified: Reg. indirection left out (for clarity).
- SIMD slightly more complex (case above is elwidth = default)
- Scalar-scalar and scalar-vector and vector-vector now all in one
- OoO may choose to push ADDs into instr. queue (v. busy!)

Summary

- Goal is to create a mass-volume low-power embedded SoC suitable for use in netbooks, chromebooks, tablets, smartphones, IoT SBCs.
- No DRM. 'Trustable' (by the users, not by Media Moguls) design ethos as a *business* objective: requires full transparency as well as Formal Correctness Proofs
- Collaboration with OpenPOWER Foundation and Members absolutely essential. No short-cuts. Standards to be developed and ratified so that everyone benefits.
- Working on the back of huge stability of POWER ecosystem
- Combination of which is that Board Support Package is 100% upstream, app and product development by customer is hugely simplified and much more attractive

The end

Thank you

Questions?

- Discussion: Libre-SOC-dev mailing list
- Freenode IRC #libre-soc
- http://libre-soc.org/
- http://nlnet.nl/PET