Appendix A

Xpulpimg Instruction Set

This appendix presents the full Xpulpimg instruction set extension. In the following, the mnemonics of the instructions are reported, along with their behavior and the specification of the employed operands fields from the instruction encoding.

This documentation is inspired from the CV32E40P core ISA specification [20] and from its reference RTL implementation [22]; for further details about the instructions and their encoding, refer to the mentioned sources.

The operands from the standard RISC-V ISA, used throughout the Xpulpimg extension, are specified in the following:

- `rs1` – first source register operand, whose index is encoded in the bits `19:15`;
- `rs2` – second source register operand, whose index is encoded in the bits `24:20`;
- `rd` – destination register operand, whose index is encoded in the bits `11:7`;
- `bimm12s` – 12-bit branch offset, encoded in the bits `31`, `7`, `30:25`, `11:8` and sign-extended;
- `iimm12s` – 12-bit I-type immediate, encoded in the bits `31:20` and sign-extended;
- `simm12s` – 12-bit S-type immediate, encoded in the bits `31:25`, `11:7` and sign-extended;

A.1. Generic arithmetic operations

This extension includes advanced arithmetic and logic operations to increase the efficiency of the ISA by reducing the number of instructions needed for generally useful DSP computation, such as minimum, maximum, absolute value, sign- or zero-extension. Such operations performed in one single instructions also reduce the need of employing control flow instructions, leading to higher IPC. A partial support for fixed-point operations is also included by supporting clip operations. Immediate branching instructions, allowing a comparison between a register and an immediate operands, are also supported.

General ALU instructions

`p.abs rd, rs1`	`rd` = *abs*(`rs1`)
`p.slet rd, rs1, rs2`	`rd` = `rs1` $\leq$ `rs2` ? 1 : 0 Comparison is signed
`p.sletu rd, rs1, rs2`	`rd` = `rs1` $\leq$ `rs2` ? 1 : 0 Comparison is unsigned
`p.min rd, rs1, rs2`	`rd` = `rs1` $\leq$ `rs2` ? `rs1` : `rs2` Comparison is signed
`p.minu rd, rs1, rs2`	`rd` = `rs1` $\leq$ `rs2` ? `rs1` : `rs2` Comparison is unsigned
`p.max rd, rs1, rs2`	`rd` = `rs1` $>$ `rs2` ? `rs1` : `rs2` Comparison is signed
`p.maxu rd, rs1, rs2`	`rd` = `rs1` $>$ `rs2` ? `rs1` : `rs2` Comparison is unsigned
`p.exths rd, rs1`	`rd` = *Sext*(`rs1[15:0]`)
`p.exthz rd, rs1`	`rd` = *Zext*(`rs1[15:0]`)
`p.extbs rd, rs1`	`rd` = *Sext*(`rs1[7:0]`)
`p.extbz rd, rs1`	`rd` = *Zext*(`rs1[7:0]`)

Clip instructions

New operand fields:

- `pimm5u` – Xpulp 5-bit immediate encoded in the bits `24:20` and zero-extended.

`p.clip rd, rs1, pimm5u`	If `rs1` $\leq -2^{\texttt{pimm5u}-1}$, `rd` = $-2^{\texttt{pimm5u}-1}$ else if `rs1` $\geq 2^{\texttt{pimm5u-1}} - 1$, `rd` = $2^{\texttt{pimm5u-1}} - 1$ else `rd` = `rs1` If `pimm5u` == 0, consider $-2^{\texttt{pimm5u}-1}$=-1, $2^{\texttt{pimm5u-1}} - 1$=0
`p.clipu rd, rs1, pimm5u`	If `rs1` ≤ 0, `rd` = 0 else if `rs1` $\geq 2^{\texttt{pimm5u-1}} - 1$, `rd` = $2^{\texttt{pimm5u-1}} - 1$ else `rd` = `rs1` If `pimm5u` == 0, consider $2^{\texttt{pimm5u-1}} - 1$=0
`p.clipr rd, rs1, rs2`	If `rs1` $\leq -$`rs2` $- 1$, `rd` = $-$`rs2` $- 1$ else if `rs1` $\geq$ `rs2`, `rd` = `rs2` else `rd` = `rs1`
`p.clipur rd, rs1, rs2`	If `rs1` ≤ 0, `rd` = 0 else if `rs1` $\geq$ `rs2`, `rd` = `rs2` else `rd` = `rs1`

Immediate branching

New operand fields:

- `pimm5s` – Xpulp 5-bit immediate encoded in the bits `24:20` and sign-extended.

The addition between the program counter and the (`bimm12s` << 1) offset is always signed.

`p.beqimm rs1, pimm5s, bimm12s`	If `rs1` == `pbimm5s`, PC = PC + (`bimm12s` << 1)
`p.bneimm rs1, pimm5s, bimm12s`	If `rs1` != `pbimm5s`, PC = PC + (`bimm12s` << 1)

A.2. Extended L/S addressing modes

This extension includes new addressing modes for the load and store operations of the standard RISC-V RV32I ISA. In particular, two new addressing modes are introduced:

- *post-increment* – the memory access is performed at the address specified by the base address only; the register containing the base address is then incremented of the specified offset and written back to the register file;
- *register-register* – the offset for the memory address is sourced from a register rather than an immediate; this addressing mode can be also coupled with the post-increment mode.

Load and store instructions with these new addressing modes comes in all the widths (byte, half-word, word) and signedness (signed and unsigned sub-word loads).

Load instructions

The addition between the `rs1` base address and the immediate or register offset is always signed.

`p.lb rd, iimm12s(rs1!)`	`rd` = *Sext*(Mem8[`rs1`]) `rs1` = `rs1` + `iimm12s`
`p.lbu rd, iimm12s(rs1!)`	`rd` = *Zext*(Mem8[`rs1`]) `rs1` = `rs1` + `iimm12s`
`p.lh rd, iimm12s(rs1!)`	`rd` = *Sext*(Mem16[`rs1`]) `rs1` = `rs1` + `iimm12s`
`p.lhu rd, iimm12s(rs1!)`	`rd` = *Zext*(Mem16[`rs1`]) `rs1` = `rs1` + `iimm12s`
`p.lw rd, iimm12s(rs1!)`	`rd` = Mem32[`rs1`] `rs1` = `rs1` + `iimm12s`
`p.lb rd, rs2(rs1!)`	`rd` = *Sext*(Mem8[`rs1`]) `rs1` = `rs1` + `rs2`
`p.lbu rd, rs2(rs1!)`	`rd` = *Zext*(Mem8[`rs1`]) `rs1` = `rs1` + `rs2`
`p.lh rd, rs2(rs1!)`	`rd` = *Sext*(Mem16[`rs1`]) `rs1` = `rs1` + `rs2`
`p.lhu rd, rs2(rs1!)`	`rd` = *Zext*(Mem16[`rs1`]) `rs1` = `rs1` + `rs2`

`p.lw rd, rs2(rs1!)`	`rd` = Mem32[`rs1`] `rs1` = `rs1` + `rs2`
`p.lb rd, rs2(rs1)`	`rd` = *Sext*(Mem8[`rs1` + `rs2`])
`p.lbu rd, rs2(rs1)`	`rd` = *Zext*(Mem8[`rs1` + `rs2`])
`p.lh rd, rs2(rs1)`	`rd` = *Sext*(Mem16[`rs1` + `rs2`])
`p.lhu rd, rs2(rs1)`	`rd` = *Zext*(Mem16[`rs1` + `rs2`])
`p.lw rd, rs2(rs1)`	`rd` = Mem32[`rs1` + `rs2`]

Store instructions

New operand fields:

- `rs3` – third source register operand, whose index is encoded in the bits `11:7`.

The addition between the `rs1` base address and the immediate or register offset is always signed.

`p.sb rs2, simm12s(rs1!)`	Mem8[`rs1`] = `rs2` `rs1` = `rs1` + `simm12s`
`p.sh rs2, simm12s(rs1!)`	Mem16[`rs1`] = `rs2` `rs1` = `rs1` + `simm12s`
`p.sw rs2, simm12s(rs1!)`	Mem32[`rs1`] = `rs2` `rs1` = `rs1` + `simm12s`
`p.sb rs2, rs3(rs1!)`	Mem8[`rs1`] = `rs2` `rs1` = `rs1` + `rs3`
`p.sh rs2, rs3(rs1!)`	Mem16[`rs1`] = `rs2` `rs1` = `rs1` + `rs3`
`p.sw rs2, rs3(rs1!)`	Mem32[`rs1`] = `rs2` `rs1` = `rs1` + `rs3`
`p.sb rs2, rs3(rs1)`	Mem8[`rs1` + `rs3`] = `rs2`
`p.sh rs2, rs3(rs1)`	Mem16[`rs1` + `rs3`] = `rs2`
`p.sw rs2, rs3(rs1)`	Mem32[`rs1` + `rs3`] = `rs2`

A.3. MAC operations

This extension includes 32-bit multiply-accumulate operations. Not only do these instruction use the destination register to write-back the result, but they also use it as a source operand to perform the accumulation operation. This operation can be an addition of the multiplication result to the content of the destination register, or a subtraction from it.

Multiply-accumulate instruction

`p.mac rd, rs1, rs2`	$rd = rd + rs1 \cdot rs2$
`p.msu rd, rs1, rs2`	$rd = rd - rs1 \cdot rs2$

A.4. Packed-SIMD extension

This extension introduces packed-SIMD operations for sub-words of:

- 8 bits (`.b` mode) – to perform 4 operations on the 4 bytes of a 32-bit word at the same time;
- 16 bits (`.h` mode) – to perform 2 operations on the 2 half-words of a 32-bit word at the same time.

SIMD instructions comes with three execution modes affecting the second operand:

- *vectorial* (default mode) – the two input registers `rs1` and `rs2` are considered vectors of two 16-bit or four 8-bit elements, and an element-wise operation is performed;
- *scalar replication* (`.sc` mode) – the lowest half-word or byte of `rs2` is considered a scalar and used for the operation with the array of half-words or bytes in `rs1`;
- *immediate scalar replication* (`.sci` mode) – this variation has the same behavior of the scalar replication mode but uses the 6-bit signed or unsigned immediate as scalar operand.

Note that SIMD is not supported by the compiler toolchain; the compiler only provides SIMD intrinsics.

New operand fields:

- `pimm6s` – Xpulp 6-bit immediate encoded in the bits `25:20` and sign-extended to 8 or 16 bits;
- `pimm6u` – Xpulp 6-bit immediate encoded in the bits `25:20` and zero-extended to 8 or 16 bits.

In the following tables, the second operand is referred to as `op2`, and it varies based on the mentioned execution modes. Also, the indices of the packed-SIMD arrays `rd`, `rs1` and `op2` range from `0` to `1` fro 16-bit operations and from `0` to `3` for 8-bit operations:

- index = `0` – stands for bits `15:0` for 16-bit operations, or bits `7:0` for 8-bit operations;
- index = `1` – stands for bits `31:16` for 16-bit operations, or bits `15:8` for 8-bit operations;
- index = `2` – stands for bits `23:16` for 8-bit operations;
- index = `3` – stands for bits `31:24` for 8-bit operations;

SIMD ALU instructions

`pv.add[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` + `op2[i]`
`pv.sub[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` − `op2[i]`
`pv.avg[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = (`rs1[i]` + `op2[i]`) >> 1 Shift is arithmetic
`pv.avgu[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	`rd[i]` = (`rs1[i]` + `op2[i]`) >> 1 Shift is logical
`pv.min[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` ≤ `op2[i]` ? `rs1[i]` : `op2[i]` Comparison is signed
`pv.minu[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	`rd[i]` = `rs1[i]` ≤ `op2[i]` ? `rs1[i]` : `op2[i]` Comparison is unsigned
`pv.max[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` > `op2[i]` ? `rs1[i]` : `op2[i]` Comparison is signed
`pv.maxu[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	`rd[i]` = `rs1[i]` > `op2[i]` ? `rs1[i]` : `op2[i]` Comparison is unsigned
`pv.srl[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` >> `op2[i]` Shift is logical
`pv.sra[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` >> `op2[i]` Shift is arithmetic
`pv.sll[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` << `op2[i]` Shift is logical
`pv.or[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` | `op2[i]`
`pv.xor[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` ^`op2[i]`
`pv.and[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd[i]` = `rs1[i]` & `op2[i]`
`pv.abs.{h,b} rd, rs1`	`rd[i]` = `rs1[i]` ≤ 0 ? −`rs1[i]` : `rs1[i]` Comparison is signed

SIMD Dot-product instructions

`pv.dotup[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	`rd` = $\sum_i$ `rs1[i]` · `op2[i]` All operations are unsigned
`pv.dotusp[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd` = $\sum_i$ `rs1[i]` · `op2[i]` `rs1[i]` are treated as unsigned, `op2[i]` are treated as signed
`pv.dotsp[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd` = $\sum_i$ `rs1[i]` · `op2[i]` All operations are signed
`pv.sdotup[.sc[i]].{h,b} rd, rs1, {rs2,pimm6u}`	`rd = rd +` $\sum_i$ `rs1[i]` · `op2[i]` All operations are unsigned
`pv.sdotusp[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd = rd +` $\sum_i$ `rs1[i]` · `op2[i]` `rs1[i]` are treated as unsigned, `op2[i]` are treated as signed
`pv.sdotsp[.sc[i]].{h,b} rd, rs1, {rs2,pimm6s}`	`rd = rd +` $\sum_i$ `rs1[i]` · `op2[i]` All operations are signed

SIMD support instructions

`pv.extract.h rd, rs1, pimm6s`	`rd` = *Sext*(`rs1[i]`) With `i` = `pimm6u[0]`
`pv.extract.b rd, rs1, pimm6s`	`rd` = *Sext*(`rs1[i]`) With `i` = `pimm6u[1:0]`
`pv.extractu.h rd, rs1, pimm6s`	`rd` = *Zext*(`rs1[i]`) With `i` = `pimm6u[0]`
`pv.extractu.b rd, rs1, pimm6s`	`rd` = *Zext*(`rs1[i]`) With `i` = `pimm6u[1:0]`
`pv.insert.h rd, rs1, pimm6s`	`rd[i]` = `rs1[15:0]` With `i` = `pimm6u[0]`; the rest of the bits of `rd` are untouched

`pv.insert.b rd, rs1, pimm6s`	`rd[i]` = `rs1[7:0]` With `i` = `pimm6u[1:0]`; the rest of the bits of `rd` are untouched
`pv.shuffle2.h rd, rs1, rs2`	`rd[i]` = `src[j]` Where `src` = `rs2[i][1]` == 1 ? `rs1` : `rd`, while `j` = rs2[i][0]
`pv.shuffle2.b rd, rs1, rs2`	`rd[i]` = `src[j]` Where `src` = `rs2[i][2]` == 1 ? `rs1` : `rd`, while `j` = rs2[i][1:0]