Implementation of the RISC-V "P" Instruction Set Extension in Shakti

Aniketh Ganesh (EP20B009)
Guide: Mr. Kotteeswaran Ekambaram
Co-Guide: Prof. Aravind R

1 Dec 2023

1 Introduction

Shakti is an open-source initiative by the Reconfigurable Intelligent Systems Engineering (RISE) group at IIT-Madras. Some of its aims are to produce complete Systems on Chips (SoCs) and a Shakti-based software platform. Shakti processors are based on the RISC-V Instruction Set Architecture. C-Class is a member of the Shakti family of processors with support for various RISC-V ISA extensions like the "F" and "M" standard extensions.

The aim of the project is to implement the proposed RISC-V "P" instruction set extension comprising of Packed Single Instruction, Multiple Data (PSIMD) instructions on Shakti's C-Class SoC. This will enable the processor to run various DSP applications with lower power and higher performance.

All the hardware for Shakti is implemented in Bluespec SystemVerilog, an open-source high-level hardware description language.

2 Packed Single Instruction Multiple Data Extension

Digital Signal Processing (DSP), has emerged as an important technology for modern electronic systems. A wide range of modern applications employ DSP algorithms to solve problems in their particular domains, including sensor fusion, servo motor control, audio decode/encode, speech synthesis and coding, MPEG4 decode, medical imaging, computer vision, embedded control, robotics, human interface, etc.

SIMD is a type of parallel processing technique used to perform the same operation on multiple data elements simultaneously, resulting in accelerated execution of tasks that involve repetitive data operations. These instructions are defined in RISC-V's "P" Extension [1].

3 Integration of pbox with the SoC

Implementing support for PSIMD instructions in the C-class SoC involves two key parts. First, a hardware module called "pbox" must be designed, housing the logic to perform the required computations. This pbox module would essentially function as an ALU for a specific set of instructions. Second, this module must be integrated with the core pipeline. That is, the SoC must decode P-extension instructions correctly, and then route parts of the instruction and the associated operands to the pbox for processing.

Certain PSIMD instructions involve accumulating addition or subtraction operations, which necessitates the retrieval of a value from the register file from the address corresponding to the destination register, denoted as `rd`. Therefore, the inputs required for the pbox module are: the values of the two operands (`rs1` and `rs2`), the destination register (`rd`), and the `funct3` and `funct7` segments of the instruction. Following the computation of the result, the pbox must furnish its output to the SoC for transmission to the subsequent stage of the pipeline. First, we begin with the integration of the pbox into the core.

3.1 Additions in the core pipeline

The pipeline of the Shakti core comprises six stages, numbered from 0 to 5, with each stage instantiated as an individual module. The pbox is also instantiated as a module at the same hierarchical level as these stages. These components are encapsulated within a higher-level module named `mkriscv`, as illustrated in the schematic. The

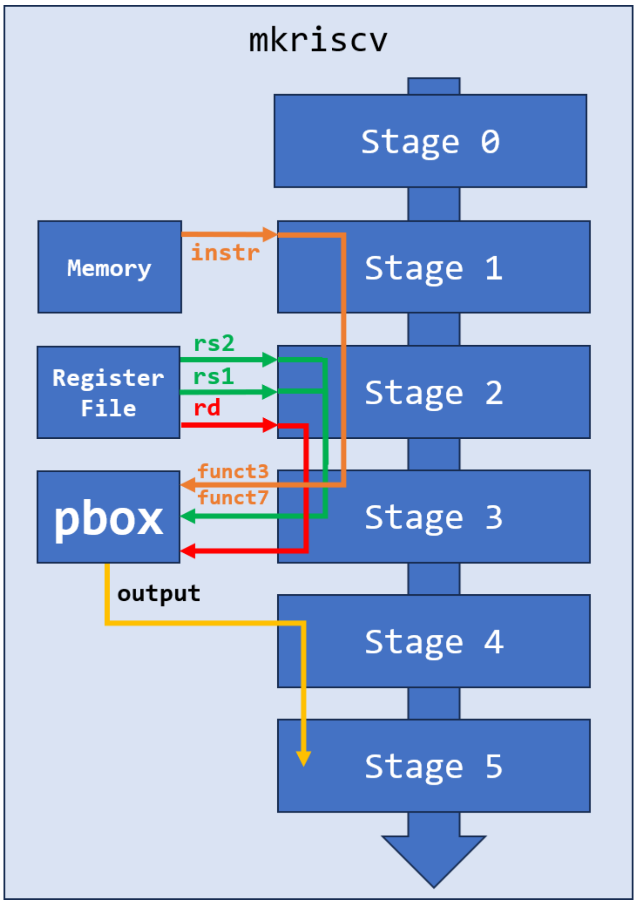


Figure 1: Pipeline of the C-Class core

connections between the interfaces of these modules are implemented within the `mkriscv` module. The following functionality was implemented in the core by making comparisons with the implementation of “mbox”, a similar module that implements RISCV’s “M” standard extension in the processor.

During the first stage of the pipeline, 32-bit instructions are fetched from the instruction memory and transferred to the subsequent stage (Stage 2). Within Stage 2, these instructions undergo decoding, and the required operands are requested from the register file based on the decoded addresses. Additional logic has been implemented in this stage to determine whether an instruction falls within the PSIMD category, based on the `opcode` segment of the instruction. If an instruction is identified as PSIMD, the segments `funct3` and `funct7` of the instruction are extracted and passed on to the subsequent stage (Stage 3).

During stage 3, the relevant inputs are directed to the pbox if the instructions fall in the PSIMD category. Subsequently, the pbox decodes `funct7` and `funct3` and conducts appropriate computations on these inputs, producing a result. The outputs of the computations are provided by the pbox sequentially, in the same order as the inputs, and are subsequently transferred to stage 4.

3.2 Additions to the Register File

A method was written in the register file to allow the reading of an additional address, `rd`. This was used to fetch the value from the register file in stage 2 of the pipeline and was subsequenty transferred to stage 3. Here, `rd` was fed to the pbox along with the other inputs.

4 Implementation of pbox

4.1 Interface of pbox with the pipeline

The inputs received by the pbox from stage 3 are stored in a pipeline FIFO, so that data can be enqued and dequed from it simultaneously when it is full. This enables the pbox to be ready for accepting instructions at every clock cycle. The pbox evalutates a result from these inputs in the same cycle. These outputs are sequentially enqueued in a FIFO buffer. The contents of this FIFO are then read into stage 4, ensuring a faithful transfer of data within the processing pipeline.

4.2 Implementation of PSIMD instructions

Since multiplication operations constitute a significant portion of PSIMD instructions, the implementation commenced with this category of instructions. Typically, these instructions involve calculating the straight or crossed products of the contents of two registers and performing additional computations with these products. This process was divided into three stages for all instructions:

1. Input select: Decode the `funct7` and `funct3` portions of the instruction to decide
   - Which bits of the input registers should be fed to the multiplier (crossed or straight multiplication)
   - Whether signed or unsigned multiplication is to be performed
2. A multiplier of appropriate size that actually computes the signed/unsigned products
3. Additions, subtractions, saturations, and/or rounding of these products

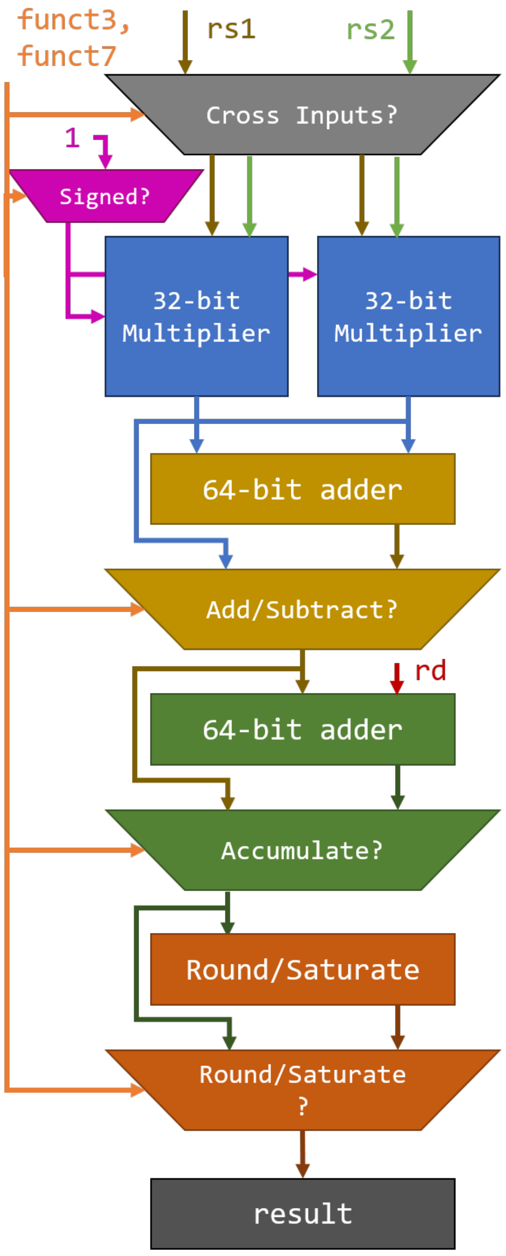


Figure 2: Schematic of 32-bit signed multiply-accumulate instructions

4.3 Sign-configurable multiplier

Multipliers occupy the most area among all the logic modules necessary for the execution of multiply instructions. Hence, the design must attempt to maximize their re-usability.

The execution of multiply instructions involves handling 8-bit, 16-bit, or 32-bit signed as well as unsigned integers. This necessitates both signed and unsigned multipliers in each of these sizes. In order to achieve this efficiently with minimal wasted hardware, it was decided to design a single configurable signed/unsigned multiplier capable of performing both kinds of operations.

The binary arithmetic was worked out for the unsigned and 2's complement signed multiplications of an n-bit integer with an m-bit one. The extra steps required in signed multiplication, compared to unsigned, are:

1. The addition of 1 in positions $n-1$, $m-1$, and $m+n-1$
2. The negation of certain partial products

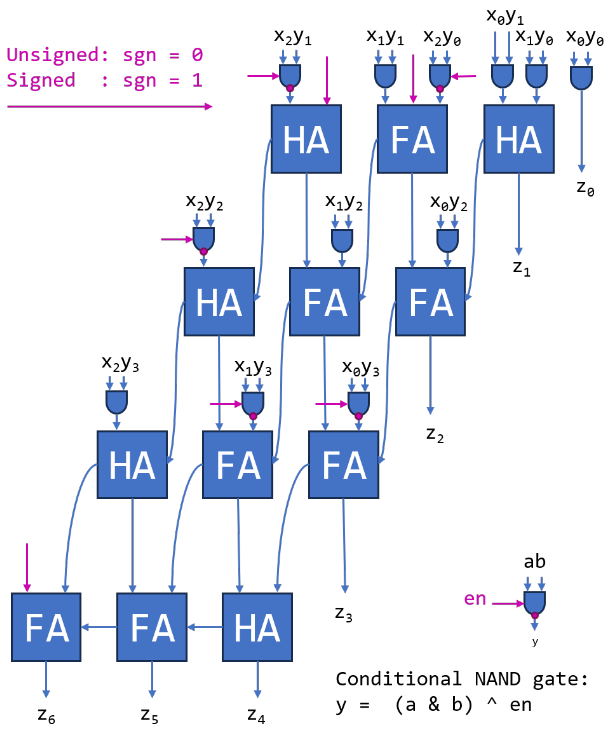


Figure 3: Schematic of a 3-bit * 4-bit configurable signed/unsigned multiplier

Leveraging its simple architecture, this was implemented using a modified array multiplier with carry-saving to reduce the critical path delay. The first requirement was achieved by supplying the sign-configure signal as an additional input to the appropriate adders. For the second requirement, the relevant partial products were generated through a gate that implements the function $y = (a \cdot b) \oplus en$. i.e, the gate works as an AND gate when the enable signal is 0 (for unsigned multiplication) and NAND when it is 1 (for signed multiplication). The diagram illustrates the working of a 3-bit * 4-bit sign-configurable multiplier.

This was implemented as a function parameterized by the sizes n and m to allow the code to be reused for inputs of different sizes. For example, the 8-bit multiply instructions would use eight 8-bit*8-bit multipliers, 16-bit multiply instructions would employ four 16-bit*16-bit multipliers, etc.

4.4 Decoding the funct7 and funct3 segments

The central idea was to use multiplexers to order the inputs and route them to appropriate combinational blocks according to the instruction. Hence, the instructions were methodically grouped based on properties they shared, such as crossing of the inputs, addition of the products to one another, negation of a product, accumulation of a product to the destination value, etc.

The multiplexers would be placed sequentially, and hence this grouping of instructions was done at multiple levels. For example, a set A of 32-bit multiplication instructions would be grouped as "accumulate instructions," i.e., the products obtained by multiplying 32-bit halves of the operands will have to be added to or subtracted from the destination register `rd`. Now, a subset B of A would require the product to be negated before this addition (ie, it is to be subtracted from `rd`), while another subset C of A might require the addition with `rd` to be Q63-saturating, etc. The aim is to construct boolean functions of the individual bits of `funct3` and `funct7` to represent these sets.

In the P-extension's documentation, similar instructions are typically assigned the same value of `funct3` and similar values of `funct7` (separated by low hamming distances). All instructions were assigned a decimal number equivalent to their `funct7` value, as shown. The sets mentioned above were identified in this grid, and the decimal numbers in them were noted. The function corresponding to this set must be such that it returns 1 for all these terms (i.e., these would be the minterms of the function) and 0 for all terms not in this subset, yet part of the immediate superset of the set in consideration (i.e., all terms outside this immediate superset can be treated as don't-cares).

A Python script was written to generate the list of minterms, zeros, and don't cares corresponding to a given subset and superset of instructions. With the help of K-maps, this was used to derive a boolean function in terms of the bits of `funct7`. The final select signals to the multiplexers were evaluated based on a combination of the `funct3` values and these boolean functions.

4.5 Combinational circuits for other computations

Some instructions require Qn saturation; that is, if the result of any intermediate computation requires more than the number of bits n available for it, overflow must be avoided and the result saturated at -2^{n-1} or $2^{n-1} - 1$. For performing Qn saturation, a function parameterized by the size n was implemented.

Similarly, some instructions require the final n bit result to be rounded and stored in $n/2$ bits. A function parameterized by the sizes n and m was implemented to round an n-bit integer to m bits. Using the decoding logic, the sign configurable multiplier, and these rounding and saturation circuits, a total of 58 instructions comprising 16-bit multiply, 16-bit multiply-accumulate, 16-bit multiply-saturate, 32-bit multiply-accumulate, and 32-bit multiply instructions have been implemented.

5 Testing

5.1 Building the assembler

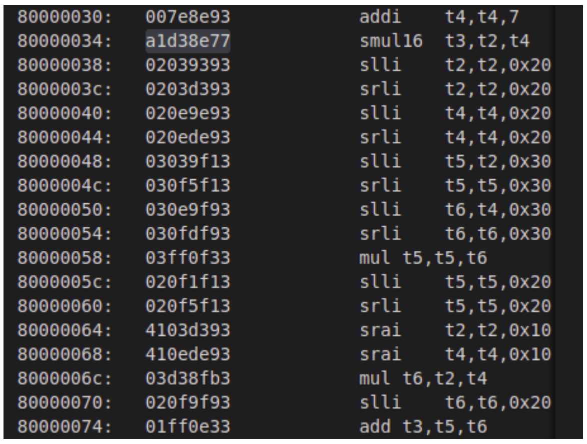
Since the PSIMD extension has not yet been ratified, there is no official assembler available from RISC-V. For testing, an assembler was prepared by building the RISC-V GNU Compiler Toolchain [2] with experimental repositories for gcc [3] and binutils-gdb [4].

5.2 Verification of pbox functionality

An assembly program, `psimd.S`, containing PSIMD opcodes was written. It was assembled into an ELF file using the assembler described above, and subsequently converted to a hexfile using the `elf2hex` utility. This hexfile would be the instruction memory for the processor and was placed in the `bin` directory of c-class. The entire setup was simulated using Verilator. The list of committed instructions, along with the value written into the destination register, would be logged in `rtl.dump`. Thus, the correct functionality of the implemented instructions was verified.

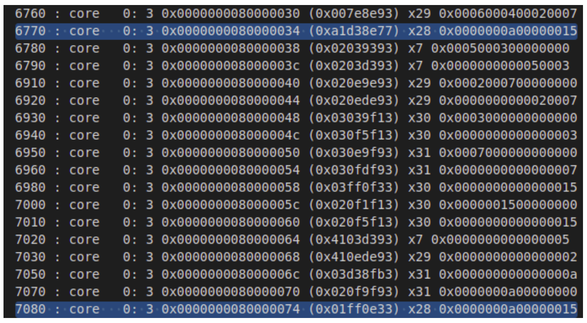
5.3 Performance analysis

In order to analyse the improvement in performance achieved by using PSIMD instructions over regular RISC-V assembly, an equivalent assem-



```
80000030:   007e8e93            addi   t4,t4,7
80000034:   a1d38e77            smul16  t3,t2,t4
80000038:   02039393            slli   t2,t2,0x20
8000003c:   0203d393            srli   t2,t2,0x20
80000040:   020e9e93            slli   t4,t4,0x20
80000044:   020ede93            srli   t4,t4,0x20
80000048:   03039f13            slli   t5,t2,0x30
8000004c:   030f5f13            srli   t5,t5,0x30
80000050:   030e9f93            slli   t6,t4,0x30
80000054:   030fdf93            srli   t6,t6,0x30
80000058:   03ff0f33            mul t5,t5,t6
8000005c:   020f1f13            slli   t5,t5,0x20
80000060:   020f5f13            srli   t5,t5,0x20
80000064:   4103d393            srai   t2,t2,0x10
80000068:   410ede93            srai   t4,t4,0x10
8000006c:   03d38fb3            mul t6,t2,t4
80000070:   020f9f93            slli   t6,t6,0x20
80000074:   01ff0e33            add t3,t5,t6
```

(a) Instruction memory disassembly: The P instruction $SMUL16$ is followed by equivalent M instructions



```
6760 : core   0: 3 0x0000000080000030 (0x007e8e93) x29 0x0006000400020007
6770 : core   0: 3 0x0000000080000034 (0xa1d38e77) x28 0x0000000a00000015
6780 : core   0: 3 0x0000000080000038 (0x02039393) x7  0x0005000300000000
6790 : core   0: 3 0x000000008000003c (0x0203d393) x7  0x0000000000050003
6910 : core   0: 3 0x0000000080000040 (0x020e9e93) x29 0x0002000700000000
6920 : core   0: 3 0x0000000080000044 (0x020ede93) x29 0x0000000000020007
6930 : core   0: 3 0x0000000080000048 (0x03039f13) x30 0x0003000000000000
6940 : core   0: 3 0x000000008000004c (0x030f5f13) x30 0x0000000000000003
6950 : core   0: 3 0x0000000080000050 (0x030e9f93) x31 0x0007000000000000
6960 : core   0: 3 0x0000000080000054 (0x030fdf93) x31 0x0000000000000007
6980 : core   0: 3 0x0000000080000058 (0x03ff0f33) x30 0x0000000000000015
7000 : core   0: 3 0x000000008000005c (0x020f1f13) x30 0x0000001500000000
7010 : core   0: 3 0x0000000080000060 (0x020f5f13) x30 0x0000000000000015
7020 : core   0: 3 0x0000000080000064 (0x4103d393) x7  0x0000000000000005
7030 : core   0: 3 0x0000000080000068 (0x410ede93) x29 0x0000000000000002
7050 : core   0: 3 0x000000008000006c (0x03d38fb3) x31 0x000000000000000a
7070 : core   0: 3 0x0000000080000070 (0x020f9f93) x31 0x0000000a00000000
7080 : core   0: 3 0x0000000080000074 (0x01ff0e33) x28 0x0000000a00000015
```

(b) RTL dump: The first column indicates the clock cycle (scaled by 10) at which the instruction was committed

Figure 4: Performance analysis of $SMUL16$ instruction

bly program was written using instructions from the "M" Standard Extension to implement certain PSIMD operations. The number of cycles taken in both cases was compared. On average, PSIMD instructions were observed to offer a 30x speedup compared to implementations using "M" instructions for the same computations.

6 Conclusion

A module was made to implement RISC-V's PSIMD extension and integrated with the pipeline of Shakti's c-class processor. A sign-configurable multiplier was designed for performing PSIMD computations efficiently. A generalized system was devised to generate the boolean logic for decoding the `funct3` and `funct7` portions of the instructions. Using these, 58 instructions have been implemented, and the implementation of the remaining instructions has been reduced to directly applying the methods developed here. The correct functioning of the integration was verified through Verilator simulations. The implementation of this extension provides, on average, a 30x speedup for relevant SIMD computations, enabling Shakti to perform DSP and AI computations significantly faster.

The source code for this project can be accessed at `https://github.com/aniketh-g/p-box/`

References

[1] RISC-V (2022) riscv-p-spec [Documentation]. `https://github.com/riscv/riscv-p-spec`

[2] RISC-V (2022) RISC-V GNU Compiler Toolchain [Source Code]. `https://github.com/riscv-collab/riscv-gnu-toolchain`

[3] Sinan, L (2022) GNU Compiler Collection [Source Code]. `https://github.com/linsinan1995/riscv-gcc/tree/riscv-gcc-experiment-p-ext`

[4] Sinan, L (2022) GNU development tools [Source Code]. `https://github.com/linsinan1995/riscv-binutils-gdb/tree/riscv-binutils-experiment-p-ext`