CS6230: CAD for VLSI, Project Report, July-Nov 2020

Domain Specific Hardware Accelerators.

Vector Negation and Statistic Minima

Submitted By.

Ashwini Tagadghar (CS19M014)
Shailesh Tiwary (CS20M001)
Sooryakiran (ME17B174)

Domain Specific Hardware Accelerators Project Report.

 July-Nov, 2020

The source code associated with this project is publicly available on GitHub at the following URL: Domain Specific Hardware Accelerator Repository.

Contents.

Overview.

1 Landscape.

Domain Specific Hardware Accelerators are processors designed to perform a specialized task. These tasks extend from accelerators for signal processing to specific matrix multiplication cores concerning neural networks. These devices are power, area, and time optimized for a particular task and exploits parallelism, resulting in significant performance enhancement. Furthermore, these devices have Direct Memory Access, and sometimes these have an internal memory of capacity similar to the primary memory.

Overview.

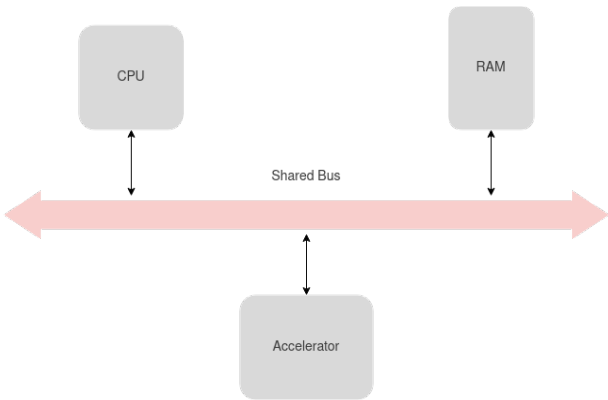

Figure 1. Overview of the system.

The principal objective of our project is to design vector accelerators for elementwise negation and statistics minima. These accelerators are connected to a common bus and have

directory access to the memory, further lessening the CPU workload. The CPU issues instructions to the accelerators, which comprises pointers to data in the memory. There is no direct data transfer between the CPU and the accelerators. The result of the vector computation is written back to the memory. The Control Status Registers in these accelerators let the CPU control and read status from them. Vector instructions on float32, int8, int16 int32 data types are implemented.

Parameterized and Modular Design.

Our design is completely modular. All the modules, i.e., the CPU, Memory, and Accelerators, are parameterized. All the modules can be instantiated with many different combinations of parameters, including wordlength, data length, and bus data & address width subjected to logical constraints.

Testbench, Simulation and Custom Assembler.

To ease the development process with better testing capabilities, we wrote a minimal assembler in Python that converts the sequences of instructions supported by our CPU into machine code, which can be utilized to initialize the instruction memory in Bluespec.

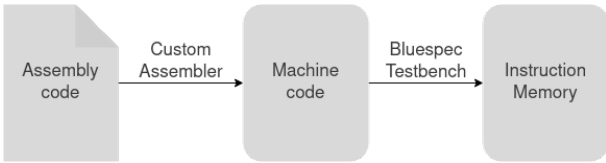

Figure 2. Simulation of the system.

For more details about the instruction set, see section CPU.

2 Central Processing Unit.

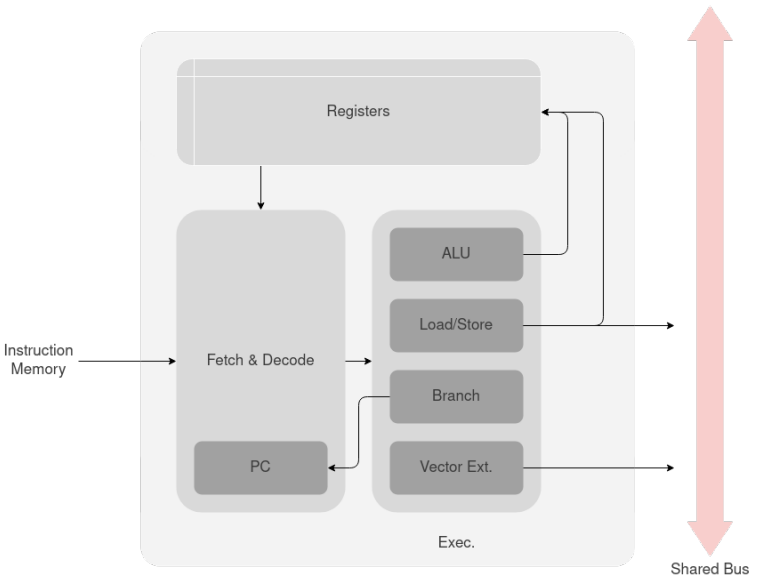

Figure 1. Architecture of the CPU.

We designed a minimal 2 stage pipelined CPU capable of doing basic arithmetic, logic, memory load/store, and custom vector operations. The instruction memory is separated from the data memory. Instructions will be issued in order. For branches, the fetch stage will be flushed, and the program counter will be updated. In the case of vector instructions, the appropriate CSRs of the corresponding accelerators will be updated accordingly. The

general architecture of the CPU is as given in the figure.

The Registers.

Our CPU consists of 8 general-purpose registers R0, R1, R2, ... R7. The capacity of the registers is parameterized. However, our current instruction set supports only 8, 16, and 32-bit values and operations on them. According to the instruction, an individual physical register can be perceived as an 8-bit, a 16-bit, or a 32-bit register. For instance, `ASG_8 R1 4` will consider only the lower 8 bits of R1, whereas `ASG_15 R1 4` will consider the lower 16 bits.

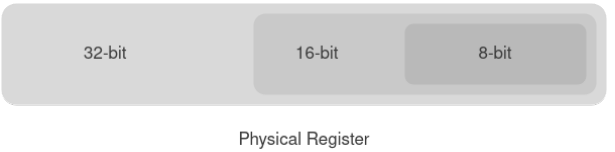

Figure 2. The same register RX can be perceived as 8, 16 or 32-bit register.

The Instruction Set.

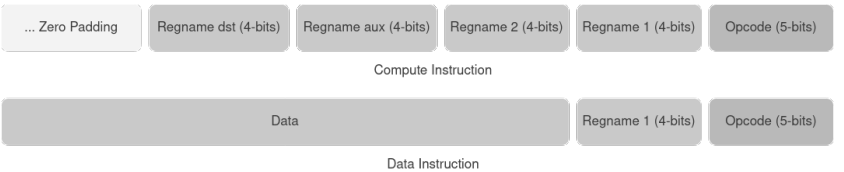

Figure 3. Structure of the instructions.

We devised a minimal custom ISA capable of basic arithmetic, logic, branch, load/store, and custom vector operations. The CPU's must have a minimum word length of 32 bits. The same instructions, padded with zeros, are used for CPUs of higher word lengths. All instructions except the 32-bit Assign instruction in 32-bit CPUs are of the same length equal to the word length. In this case, the instruction (`ASG_32` in 32-bit CPU) is 64-bits long to accommodate the 32-bit value for Assign operation. Our custom ISA is as given below. A small assembler was written in Python to generate machine code in a format supported by Bluespec testbench.

To run the assembler,

```
cd src/asm/
./asm input_file.asm -o output -w 64
```

Where the `-w` argument is the wordlength of the CPU. The argument is optional and the default is as for a 32-bit CPU.

Instruction	Opcode	Example	Description
NOP	0×00	NOP	No op
ASG_8	0×01	AGS_8 R1 11	Assigns int8 11 to Register R1
ASG_16	0×02	ASG_16 R1 11	Assigns int16 int 11 to Register R1
ASG_32	0×03	ASG_32 R1 11	Assigns int32 int 11 to Register R1
		ASG_32 R1 11.0	Assigns float32 11.0 to Register R1
MOV	0×04	MOV R1 R2	Moves content of R1 to R2
ADD_I8	0×05	ADD_I8 R1 R2 R3	int8 addition R3 ← R1 + R2
ADD_I16	0×06	ADD_I16 R1 R2 R3	int16 addition R3 ← R1 + R2
ADD_I32	0×07	ADD_I32 R1 R2 R3	int32 addition R3 ← R1 + R2
ADD_F32	0×08	ADD_F32 R1 R2 R3	float32 addition R3 ← R1 + R2
SUB_I8	0×09	SUB_I8 R1 R2 R3	int8 addition R3 ← R1 + R2
SUB_I16	0×0a	SUB_I16 R1 R2 R3	int16 addition R3 ← R1 - R2
SUB_I32	0×0b	SUB_I32 R1 R2 R3	int32 addition R3 ← R1 - R2
SUB_F32	0×0c	SUB_F32 R1 R2 R3	float32 addition R3 ← R1 - R2
IS_EQ	0×0d	IS_EQ R1 R2 R3	R3 ← (R1 == R2)
JMP	0×0e	JMP R1	Jump PC to address pointed by R1
JMPIF	0×0f	JMPIF R1 R2	Jump to address R1 if R2 is true
LOAD_8	0×10	LOAD_8 R1 R2	Load 8-bits from address pointed by R1 to R2
LOAD_16	0×11	LOAD_16 R1 R2	Load 16-bits from address pointed by R1 to R2
LOAD_32	0×12	LOAD_32 R1 R2	Load 32-bits from address pointed by R1 to R2
STORE_8	0×13	STORE_8 R1 R2	Store 8-bits R1 to address pointed by R2
STORE_16	0×14	STORE_16 R1 R2	Store 16-bits R1 to address pointed by R2
STORE_32	0×15	STORE_32 R1 R2	Store 32-bits R1 to address pointed by R2
VEC_NEG_I8	0×16	VEC_NEG_I8 R1 R2 R3	Bit wise negation of int8 vector starting pointed by R1, length R2, and store back to address in R3
VEC_NEG_I16	0×17	VEC_NEG_I16 R1 R2 R3	Bit wise negation of int16 vector starting pointed by R1, length R2,

			and store back to address in R3
VEC_NEG_I32	0×18	VEC_NEG_I31 R1 R2 R3	Bit wise negation of int32 vector starting pointed by R1, length R2, and store back to address in R3
VEC_NEG_F32	0×19	VEC_NEG_F32 R1 R2 R3	Bit wise negation of float32 vector starting pointed by R1, length R2, and store back to address in R3
VEC_MIN_I8	0×1a	VEC_MIN_I8 R1 R2 R3	Statistics minimum of int8 vector starting pointed by R1, length R2, and store back to address in R3
VEC_MIN_I16	0×1b	VEC_MIN_I16 R1 R2 R3	Statistics minimum of int16 vector starting pointed by R1, length R2, and store back to address in R3
VEC_MIN_I32	0×1c	VEC_MIN_I31 R1 R2 R3	Statistics minimum of int32 vector starting pointed by R1, length R2, and store back to address in R3
VEC_MIN_F32	0×1d	VEC_MIN_F32 R1 R2 R3	Statistics minimum of float32 vector starting pointed by R1, length R2, and store back to address in R3

The Fetch and Decode Stage.

The fetch stage fetches the instructions from the instruction memory, checks for any data dependencies, and looks up the data required for computation from the registers, and enqueues to decoded instruction to the execute stage. In the case of any data dependencies, a `NOP` is sent into the execute stage.

All the data instructions (the assignment instructions) are processed in the fetch stage. This architecture creates a conflict if the execute stage tries to store a value to the same register. The data instruction from the fetch stage is given priority for resolving the conflict. In other words, during a consecutive write after write (WAW) dependency, the first write is disregarded.

The fetch stage is also responsible for decomposing complex vector instructions to a sequence of load/store instructions. For example, the `VEC_NEG_I8` instruction is broken down into a stream of store instructions that stores the pointers to the data to the memory-mapped accelerator and triggers the accelerator. This process creates overhead on the CPU. A potential alternative is for the compiler to decompose the vector functions

and keep the individual instructions short. However, this method results in more large programs.

The Execute Stage.

Since being an inorder processor, the execute stage is relatively straight forward. The opcode is used to send the instructions to the right unit. In the case of a branch, the pipeline is flushed. There is no value forwarding to the fetch stage, and the instructions with data dependencies are kept awaiting. There is also no value forwarding between store and load operations.

For vector instructions, the fetch stage already decomposes the instructions into the corresponding load/store transactions. Therefore the execute stage executes these instructions, idles and pings the accelerator to check for the completion of the instruction. The length of the vector determines the frequency of pings. The pings must not be excessively frequent as it consumes the expensive bus ownership time that the accelerators require, and the pings must not be over sparse as this results in the CPU holding idle for long.

3 Random Access Memory.

Memory access is the bottleneck in almost all of the vector operations. A multi-ported RAM with parallel read/writes coupled with a wide data bus is essential to profit from the vector accelerators' parallel processing capabilities.

Implimentation.

Vectors of EHRs were used to implement the RAM. CReg supplied by Bluespec limited the number of ports to 5. Assuming the smallest addressable unit is 1 byte (this is parameterized in the implementation), this limitation caused the throughput to be capped at 5 bytes per cycle. Another implementation of EHRs from https://web.mit.edu/6.375/install/bsvclib-2007-02-20_19-10/EHR.bsv removed this limitation, and the memory throughput is now only limited by the bus width. The ports followed the priority rules of EHRs. A wrapper was written to coordinate the ports and interface with the bus. To illustrate, if the bus requested only 1 byte, only one port would be active. On the other case, if the bus can transmit 64 bytes per cycle (512 bits) and requests for 512 bits, all 64 ports would be active.

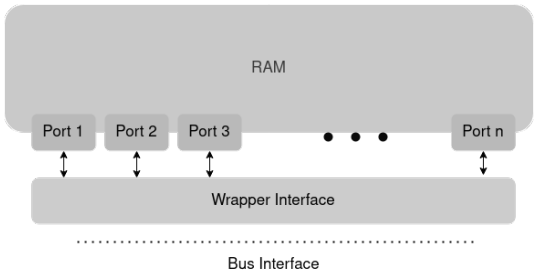

Figure 1. Ram wrapper interfaces bus to the ports.

4 The Bus.

Although Bluespec provides a Config bus (CBus) interface through the CBus and LBus package, top-level test benches have to be written to mediate communication between two or more module interfaces. Employing the CBus package hindered modularity as we insisted our modules to have plug and play interfaces. Bluespec also had implemented in their libraries many of the standard bus protocols like AHB and AXI with TLM interface. These provided full-fledged capabilities that were overkill for use in our system. So, we decided on a simple bus protocol to implement ourselves on Bluespec.

The Bus System.

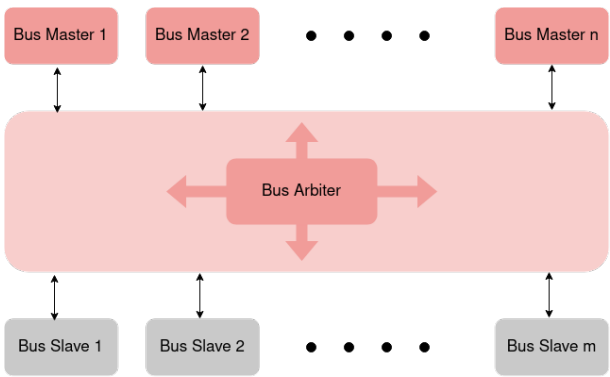

Figure 1. The bus system.

The system consists of a Bus Fabric to which different modules are connected. There are two types of modules, the Bus Masters and the Bus Slaves. Masters issue read or write

requests to which slaves have to respond if the requested address is within its scope. The bus Arbiter determines which master owns the bus. Before sending a request, all masters must request the Arbiter. In the case of multiple requests, the Arbiter uses round-robin scheduling to select the owner of the bus. We used Bluespec's `Arbiter` package for implementing the Arbiter.

The Bus Protocol.

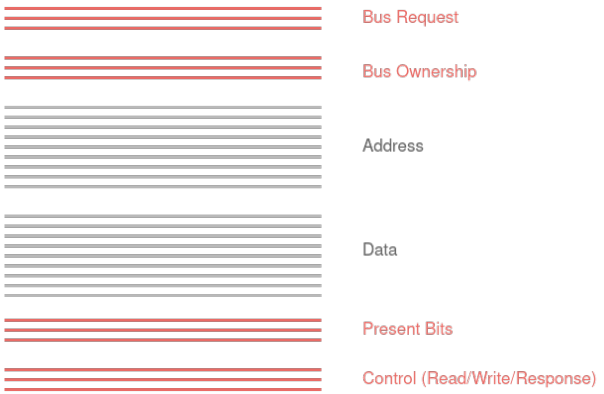

Figure 2. The bus wires.

Every master gets assigned two exclusive lines, one for requesting the Arbiter and another to receive the Arbiter response. All the rest of the Bus lines are shared.

A read request consists of an Address, the present bits indicating the number of units of the smallest addressable blocks expected by the sender, and the control wires in 'Read' mode. For example, to request four consecutive bytes to a RAM whose smallest addressable unit is 1 byte, we have our present bits to have a value of 4. In this way, connected slaves can utilize the entire bus width to the maximum. Similarly, a write request consists of almost the same details except for the data and control in 'Write' mode. Slaves respond with the appropriate 'Response.'

Interfacing the Bus.

The BusMaster and BusSlave modules provide an effortless bridge between the Bus and the associated components. Read/write requests can be enqueued into the BusMaster, and responses can be read using simple GetPut interfaces. Corresponding interfaces also exist towards the BusSlaves.

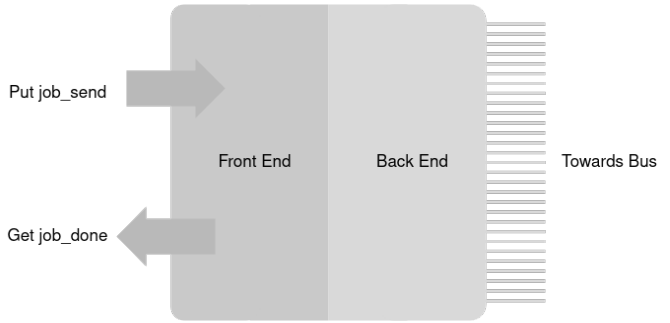

Figure 3. The BusMaster.

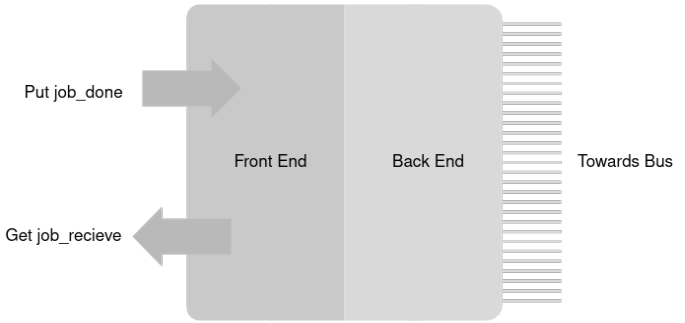

Figure 4. The BusSlave.

For details about using the Bus, see the documentation section.

Vector Extensions.

5 A Brief Introduction .

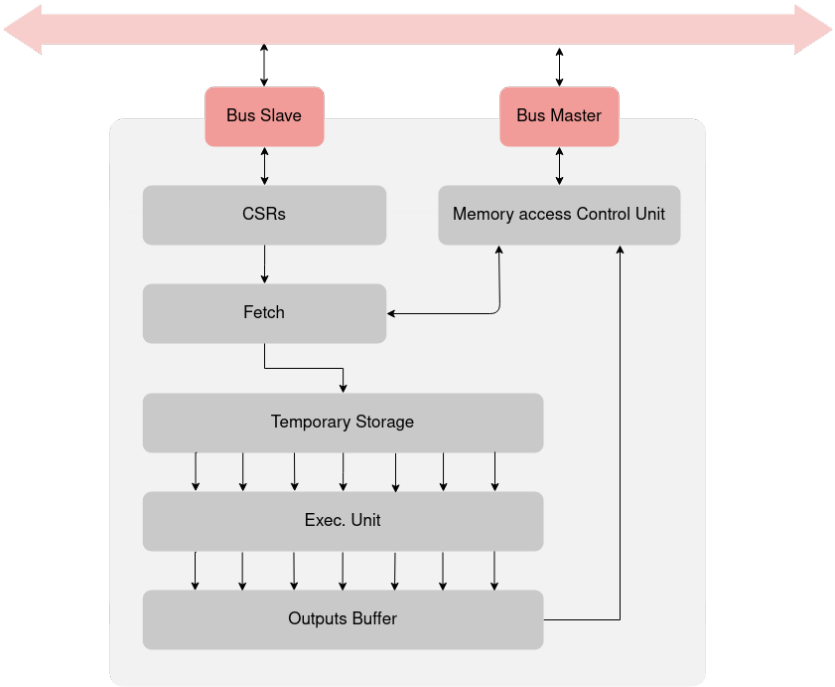

Figure 1. General architecture of our Vector Processing Unit.

The general architecture for a vector accelerator is as given in the figure. While such a formation is suitable for most vector operations, matrix operations may require a different

kind of design with systolic arrays. Since we intend to devise a design for accelerators concerning vector negation and statistics minimum, we restrict the general design for vector operations. It is to be remarked that slight alterations in dataflow may be required for binary vector operations.

The CPU writes the pointers for the source destination vectors and the vector size into the accelerators' CSRs. Once the CPU instructs the accelerator to start computation by writing to the start flag of the CSRs, the fetch unit requests the memory access controller for the data. The memory access controller handles read/write requests and responses between the accelerator and the bus. The memory access controller uses round-robin scheduling for handling simultaneous read and write requests. Once the fetch unit gets the data corresponding to a portion of the vector, it gets enqueued onto the temporary storage unit.

6 The Control Status Registers.

The Control Status Registers (CSRs) is the only communication path between the processor and the accelerator. There are multiple CSRs, each for a specific purpose. Each CSR has an exclusive address, as indicated in the figure given below. When the CPU encounters a vector instruction, it writes the required operands, i.e., the address of the source, block size of the vector, the Opcode, and the address where the result is to be stored into the CSRs. After these operands are written, the CPU writes one into the Start flag. This action causes the accelerator to start its operation. The Aux CSR is used for any extra data required by any new functionality. This design helps in adding functionality without significantly altering the design. For instance, the pointer to the minima index in vector neg operation utilizes the Aux register.

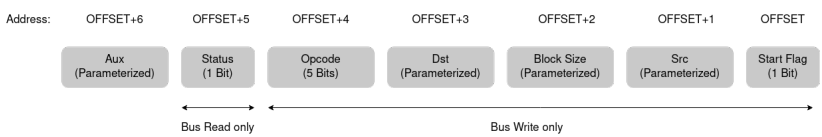

Figure 1. Control status Registers for a Unary Vector Extension.

There is also a 1-Bit status register, which, when equals one, means that the accelerator has completed its operation and is idle.

Interface with the Bus.

The CSRs interface the Bus through a BusSlave. This means that CSRs cannot launch a request by themselves. The CSR's can only respond to read/write requests initiated by the processor or any other masters. So the CPU must periodically ping the accelerator to

know the status of any operation previously initiated. The Start Flag, Src, Block Size, Dst, and Opcode are write-only. A null response is returned if the CPU attempts to read them. Similarly, the CPU can only read the `Status` register. Any attempts to write is ignored.

7 The Vector Fetch Unit.

The vector accelerator's degree of parallelism is upper bounded by the width of the data bus. The fetch unit breaks down a vector into parts of this size and get the data from memory sequentially. The data request is made through the Memory access Control Unit. For instance, suppose the data bus's width is 512 bits, and the vector units can support 512 bits; the degree of parallelism is 64 elements for 8-bit ints, 32 elements for 16-bit, and 16 elements for 32-bit floats & ints. The fetched data is enqueued to a temporary storage unit from where the data goes to a pipelined execution unit.

Temporary Storage Units.

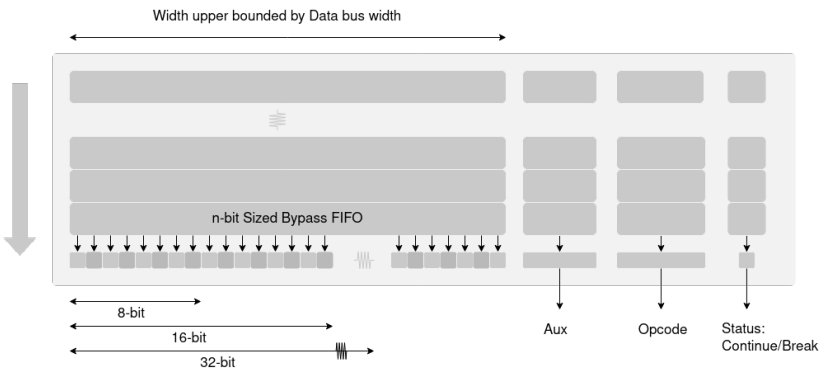

Figure 1. Structural view of the temporary storage units

The temporary storage unit is constructed out of Bypass FIFOs. Bluespec does not support the construction of sized Pipeline FIFOs, but similar functionality can be obtained by having a Pipeline FIFO in series with a sized Bypass FIFO. The outputs can be deciphered according to the Opcode and can be gathered by the appropriate execution unit. The width of the complete unit depends on the highest bandwidth supported through the bus. The functional representation of such a unit is manifested in the following figure.

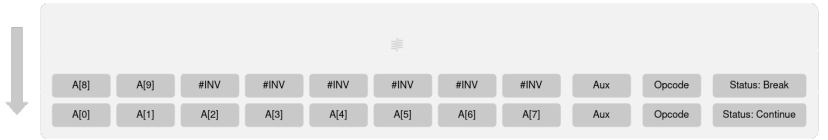

Figure 2. Functional view of a 64-bit wide storage unit storing a vector of int8 A[10]

8 The Vector Execute Units.

The data from the temporary storage units arrive at the right execute units designated by the Opcode through a Pipeline FIFO. The architecture of the two main vector functions that we focus on in this project is given below.

Vector Negation.

Implementing elementwise vector negation was relatively straight forward as there is only one to one correspondence between the inputs and outputs. For integers in 2's complement representation, negation is achieved by inverting all numbers and adding one. However, the abstraction level provided by the implementation of integers in Bluespec libraries hides all details from the writer and can be written directly.

For floating-point values, merely flipping the sign bit of a value results in the number's negative. We employed Bluespec's `FloatingPoint` library, which provided direct arithmetic functionality.

Vector Statistics Minima.

The statistics minima operation is devised to run in logarithmic time complexity. Since every vector is of variable length, usually longer than the execution unit's width, we were required to maintain the system's state. The state contains the value of the minimum from the previous batch. The state is updated if the current batch's minimum is less than the minimum of the previous batch. A status signal of 'Break' resets the states. The indices of the minima are also calculated concurrently along with the minima.

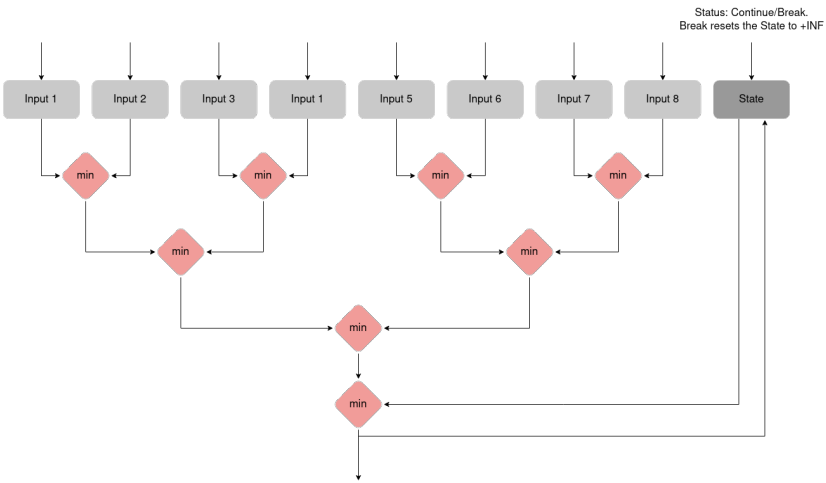

Figure 1. Execute unit for computing statistics minima.

Demo.

9 CPU Demo, The Fibonacci Series.

To demonstrate the capabilities of the CPU and the accelerator, we have included a few demo test benches. These can be run from `src/Demo/Demo.bspec`. There are 2 bsv test benches, `Demo1.bsv and Demo2.bsv`. `Demo1.bsv` contains only the CPU whereas `Demo2.bsv` contains both CPU and the accelerator.

The Assembly Code.

The assembly code for printing the first ten elements of the CPU is located in `src/asm/fibonacci_asm.asm`. The code snippet is as given below.

```
NOP                     ; A test code to print that prints the Fibonacci series
ASG_32 R7 128           ; Assign the address of the console
ASG_32 R1 1             ; Initialize first two elements
ASG_32 R2 1             ;
ASG_32 R6 8             ; Loop count. Print 10 (first 2 + 8)
ASG_32 R5 0             ; Loop initialise
ASG_32 R4 1             ; Loop increment
STORE_32 R1 R7          ; Print first 2 values through memory mapped console
STORE_32 R2 R7          ;
loop: ADD_I32 R1 R2 R3      ; Calculate the next element
    ASG_32 R7 128           ; Assign the console address again
    STORE_32 R3 R7          ; Print the new term
    MOV R2 R1               ; Forget the past and move ahead
    MOV R3 R2               ;
    ADD_I32 R5 R4 R5        ; Loop increment
    IS_EQ R5 R6 R3          ;
```

```
    ADD_I32 R3 R4 R3          ; IS_EQ outputs 1 if true. But we need 1 to jump.
    ASG_32 R7 $loop           ; Jump branch destination
    JMPIF R7 R3               ; Conditional Jump
```

To generate machine code run,

```
cd src/asm
./asm fibonacci_asm.asm -o fibonacci -w 64
```

Here `-o fibonacci` is the output file name and `-w 64` tells the assembler than we are generating code for a 64-bit machine.

To run the simulation, execute,

```
cd src/Demo
./compile_and_sim.sh Demo1.bsv
```

If you have already compiled once, just run,

```
cd src/Demo
./out
```

If no errors occur, you will get the final outputs as,

```
...
Running ./out
Warning: file '../asm/fibonacci' for memory 'my_core_imem_c_memory' has a gap
↪  at addresses 19 to 18446744073709551615.
CONSOLE 00000001 |           1
CONSOLE 00000001 |           1
CONSOLE 00000002 |           2
CONSOLE 00000003 |           3
CONSOLE 00000005 |           5
CONSOLE 00000008 |           8
CONSOLE 0000000d |          13
CONSOLE 00000015 |          21
CONSOLE 00000022 |          34
CONSOLE 00000037 |          55
```

Looking at Demo1.bsv.

Here is a snippet from `Demo1.bsv` that specifies the machine parameters.

```
`define WORD_LENGTH 64  // We are making a 64 bit machine
`define DATA_LENGTH 32  // The data size of our machine
`define BUS_DATA_LEN 32 // Data bus width
`define ADDR_LENGTH 20  // Addr bus width

`define GRANULARITY 8   // Smallest addressible unit (1 Byte at every address)
`define RAM_BYTES 64    // Ram size (number of addressible units)
`define RAM_PORTS 4     // 4 ports, 4 x 8 for 32 bit bus

`define RAM_ADDRESS_OFFSET 1000 // Address of the RAM
`define CONSOLE_ADDRESS 128     // Address of the Console
```

The console is just a slave connected to the bus that prints everything written to its address. It is made only for debugging purposes. Notice that we are writing to address 128 in the above assembly code to print the value.

```
CPU #(`WORD_LENGTH,
      `DATA_LENGTH,
      `BUS_DATA_LEN,
      `ADDR_LENGTH,
      `GRANULARITY)
      my_core <- mkCPU(1, "../asm/fibonacci"); // CPU_ID 1, Initialize IMEM
```

We transfer the generated machine code to the CPU to initialize the instruction memory while instantiating.

```
DRAMSlave #(`GRANULARITY,
            `RAM_BYTES,
            `RAM_ADDRESS_OFFSET,
            `BUS_DATA_LEN,
            `ADDR_LENGTH,
            `RAM_PORTS) my_dram    <- mkDRAMSlave(0);

Console #(`BUS_DATA_LEN,
          `ADDR_LENGTH,
          `GRANULARITY)      my_console <- mkConsole(1, `CONSOLE_ADDRESS);

Vector #(1, BusMaster #(`BUS_DATA_LEN,
                        `ADDR_LENGTH,
                        `GRANULARITY)) master_vec;
```

```
Vector #(2, BusSlave  #(`BUS_DATA_LEN,
                        `ADDR_LENGTH,
                        `GRANULARITY)) slave_vec;


master_vec[0] = my_core.bus_master;
slave_vec[0]  = my_dram.dram_slave;
slave_vec[1]  = my_console.bus_slave;

Bus #(1, 2,             // 1 master, 2 slaves
      `BUS_DATA_LEN,
      `ADDR_LENGTH,
      `GRANULARITY) bus <- mkBus(master_vec, slave_vec);

mkConnection (master_vec, bus);
mkConnection (slave_vec, bus);
```

Here you can see how all BusMaster and BusSlave interfaces are connected to the main Bus. The vectors of all BusMasters and Slaves are passed to the `mkBus(..)`. Also `mkConnection (..)` is used to connect the interfaces.

10 Demo, Vector Negate.

In this demo, we will be generating a 32-bit CPU and attach it with a vector accelerator. `src2.bsv` contains the required setup concerning this.

Looking at Demo2.bsv.

`Demo2.bsv` is different from `Demo1.bsv`. It contains our vector accelerator attached to the Bus.

```
`include <VX_Address.bsv> // Location where Accelerator is memory mapped

`define WORD_LENGTH 32   // Here we are generating a 32 bit CPU
`define DATA_LENGTH 32
`define BUS_DATA_LEN 128 // When changing bus width, remember to increase
↪  memory ports
`define ADDR_LENGTH 20
`define VECTOR_DATA_SIZE `BUS_DATA_LEN
`define VX_STORAGE_SIZE 2

`define GRANULARITY 8    // Smallest addressible unit (1 byte)
`define RAM_BYTES 64     // Ram size (number of addressible units)
`define RAM_PORTS 16     // 16 ports, 1 byte per port for 128 bit bus
`define RAM_ADDRESS_OFFSET 1024

`define CONSOLE_ADDRESS 128
```

The vector accelerator is defined as attached as follows,

```
VectorUnary #(`DATA_LENGTH,
              `VECTOR_DATA_SIZE,
              `BUS_DATA_LEN,
              `ADDR_LENGTH,
              `GRANULARITY) vec_Unary <- mkVectorUnary (`VX_ADDRESS,
              ↪  `VX_STORAGE_SIZE, 7);
                                    // VX_ADDRESS <- Memory mapped address
                                    ↪  of Accelerator
                                    // VX_STORAGE_SIZE <- Depth of
                                    ↪  temporary storage FIFOs
Vector #(2, BusMaster #(`BUS_DATA_LEN,
                        `ADDR_LENGTH,
                        `GRANULARITY)) master_vec;

Vector #(3, BusSlave  #(`BUS_DATA_LEN,
                        `ADDR_LENGTH,
                        `GRANULARITY)) slave_vec;

...
...
...

slave_vec[2]  = vec_Unary.bus_slave;

Bus #(2, 3, `BUS_DATA_LEN,
            `ADDR_LENGTH,
            `GRANULARITY) bus <- mkBus(master_vec, slave_vec);

mkConnection (master_vec, bus);
mkConnection (slave_vec, bus);
```

Running Demo Files.

Four separate assembly files are provided for demonstrating vector negation on four different datatypes, `int8`, `int16`, `int32` `float32`. These files are located at,

```
src/asm/vec_neg_f32_demo.asm
src/asm/vec_neg_i8_demo.asm
```

```
src/asm/vec_neg_i16_demo.asm
src/asm/vec_neg_i32_demo.asm
```

Here is a snippet of `src/Demo/vec_neg_i32_demo.asm`. It initializes a vector of `int_32`, of length 10 with numbers from 0 to 9. Vector negation is done and the output is printed.

```
ASG_32 R1 0          ; Initialisation
ASG_32 R2 10         ; Num loops
ASG_32 R3 1024       ; Address
ASG_32 R4 1          ; Value increment delta
ASG_32 R5 4          ; Address increment delta
ASG_32 R0 128        ; Address of console
loop_init: STORE_32 R1 R3      ;
    STORE_32 R1 R0       ; Print console
    ADD_I32 R1 R4 R1     ; Increment value
    ADD_I32 R3 R5 R3     ; Increment address
    ASG_32 R7 1
    IS_EQ R1 R2 R6       ; Compare
    ADD_I32 R6 R7 R6
    ASG_32 R7 $loop_init
    JMPIF R7 R6          ; Jump if not Eq
ASG_32 R3 1024
VEC_NEG_I32 R3 R2 R3     ; Vector negation
ASG_32 R1 0
loop_print: LOAD_32 R3 R4    ; Load from memory
    STORE_32 R4 R0           ; Print
    ADD_I32 R3 R5 R3         ; Incement address
    ASG_32 R4 1
    ADD_I32 R1 R4 R1         ; Increment Loop
    ASG_32 R7 1
    IS_EQ R1 R2 R6           ; Compare
    ADD_I32 R6 R7 R6
    ASG_32 R7 $loop_print
    JMPIF R7 R6              ; Jump if not Eq
```

Running the Simulation.

Generate machine code by running,

```
cd /src/asm
./asm vec_neg_i32_demo.asm -o vector
```

Since, we are generating for a 32-bit CPU in this example, we do not have to specify the `-w N` option. 32-bit is the default value. Also note that the output file name is consistent with the input taken from `Demo2.bsv`

If compiling `Demo2.bsv` for the first time, run,

```
cd src/Demo/
./compile_and_sim.sh Demo2.bsv
```

Else, if you have already compiled once, then run,

```
cd src/Demo/
./out
```

If no errors occur, you will get output similar to as shown below. Note that if you are running the demo on `float32`, the outputs are in IEEE 754 format hex and appear to not convey any legible information in `int`.

```
CONSOLE 00000000 |           0
CONSOLE 00000001 |           1
CONSOLE 00000002 |           2
CONSOLE 00000003 |           3
CONSOLE 00000004 |           4
CONSOLE 00000005 |           5
CONSOLE 00000006 |           6
CONSOLE 00000007 |           7
CONSOLE 00000008 |           8
CONSOLE 00000009 |           9
CONSOLE 00000000 |           0
CONSOLE ffffffff |          -1
CONSOLE fffffffe |          -2
CONSOLE fffffffd |          -3
CONSOLE fffffffc |          -4
CONSOLE fffffffb |          -5
CONSOLE fffffffa |          -6
CONSOLE fffffff9 |          -7
```

```
CONSOLE fffffff8 |          -8
CONSOLE fffffff7 |          -9
```

11 Demo, Statistics Minima.

In this demo, we will initialize a vector with some values, and we will print its minima and minimum index. The assembly files required for this demo are located at,

```
src/asm/vec_min_f32_demo.asm
src/asm/vec_min_i8_demo.asm
src/asm/vec_min_i16_demo.asm
src/asm/vec_min_i32_demo.asm
```

Here is a snippet from `vec_min_i8_demo.asm`

```
NOP                 ; Store Array [13, 12, 11, -13, 15]
ASG_32 R3 128       ; Console address
ASG_8 R1 13         ; Value
ASG_32 R2 1024      ; Address
STORE_8 R1 R2       ; Store to RAM
STORE_8 R1 R3       ; Print
ASG_8 R1 12         ; Value
ASG_32 R2 1025      ; Address
STORE_8 R1 R2       ; Store to RAM
STORE_8 R1 R3       ; Print
ASG_8 R1 11         ; Value
ASG_32 R2 1026      ; Address
STORE_8 R1 R2       ; Store to RAM
STORE_8 R1 R3       ; Print
ASG_8 R1 -13        ; Value
ASG_32 R2 1027      ; Address
```

```
STORE_8 R1 R2       ; Store to RAM
STORE_8 R1 R3       ; Print
ASG_8 R1 15         ; Value
ASG_32 R2 1028      ; Address
STORE_8 R1 R2       ; Store to RAM
STORE_8 R1 R3       ; Print
NOP
ASG_32 R1 1024      ; Vector starting location
ASG_32 R2 5         ; Vector size
ASG_32 R4 1024      ; Minimum dst
ASG_32 R5 1025      ; Argmin dst
VEC_MIN_I8 R1 R2 R4 R5  ; Vec op
LOAD_8 R4 R6        ; Load minima
STORE_8 R6 R3       ; Print minima
LOAD_8 R5 R7        ; Load argmin
STORE_8 R7 R3       ; Print argmin
```

Running the Simulation.

Generate machine code by running,

```
cd /src/asm
./asm vec_min_i8_demo.asm -o vector
```

Since, we are generating for a 32-bit CPU in this example, we do not have to specify the `-w N` option. 32-bit is the default value. Also note that the output file name is consistent with the input taken from `Demo2.bsv`

If compiling `Demo2.bsv` for the first time, run,

```
cd src/Demo/
./compile_and_sim.sh Demo2.bsv
```

Else, if you have already compiled once, then run,

```
cd src/Demo/
./out
```

If no errors occur, you will get output similar to as shown below. Note that if you are running the demo on `float32`, the outputs are in IEEE 754 format hex and appear to not convey any legible information in `int`. The last 2 lines represent the minimum value and the index of the minimum value (starting from 0) respectively. If multiple minima exists, the index with the lower value is outputted.

```
CONSOLE 0d |   13
CONSOLE 0c |   12
CONSOLE 0b |   11
CONSOLE f3 |  -13
CONSOLE 0f |   15
CONSOLE f3 |  -13
CONSOLE 03 |    3
```

Documentation.

12 Project Structure.

Directory Structure.

The directory structure of the project is as shown here. The directory `docs` contains all the documentation and its sources. The assembler and the sample assembly codes are present in the `asm` directory inside the `src`. The packages `Bus`, `CPU`, `DebugConsole`, `RAM` and `VectorProcessor` are located in their corresponding directories. The folder `Demo` contains the demo files demonstrating the usage and capablities.

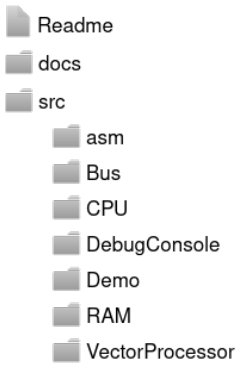

Figure 1. The directory structure.

Assembler.

The assembler is located at the following path: `src/asm/asm`.

Demo Testbenches.

The demo testbenches are located at the following paths: `src/Demo/Demo1.bsv & src/Demo/Demo2.bsv`

Git Repository.

The git repository of the project is located at the following url: https://github.com/Sooryakiran/Domain-Specific-Hardware-Accelerator-VLSI-CAD-Project/

13 Bus.

Packages.

```
import Bus ::  * ;
```

Description.

The Bus library includes interface, transactor, and function definition to fulfill a minimal Bus with Bluespec System Verilog. The implementation of BusMasters and Slaves can be attached to other devices employing naive Get and Put connections.

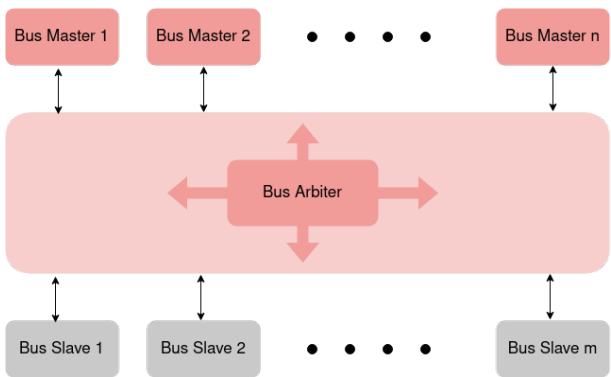

Figure 1. The bus system.

Data Structures.

Within the Bus, the data is composed of the following data structures: the `ControlSignal` contains information about the type of the signal, i.e, `Read, Write & Response`. The `Chunk`

datatype contains the entire signal.

Chunk.

The Chunk is a structure containing all the information to be transacted.

Chunk.

Member	Datatype	Valid Values
control	ControlSignal	Read/Write/Response
data	Bit #(datasize)	..
addr	Bit #(addrsize)	..
present	Bit #(PresentSize # (datasize, granularity))	..

```
typedef enum {Response, Read, Write} ControlSignal deriving (Bits, Eq, FShow);

typedef TLog #(TAdd #(TDiv #(datasize, granularity), 1))
PresentSize #(numeric type datasize,
              numeric type granularity);

typedef struct {ControlSignal control;
                Bit #(datasize) data;
                Bit #(addrsize) addr;
                Bit #(PresentSize #(datasize, granularity)) present;}

                Chunk #(numeric type datasize,
                        numeric type addrsize,
                        numeric type granularity) deriving (Bits, FShow);
```

Bus Interfaces.

The package includes two major bus interfaces `BusMaster` and `BusSlave`. The interface for the bus fabric is `Bus`.

BusMaster.

The BusMaster interface issues `Read/Write` requests and recieves `Responses`. The bus master connects with the master device through simple `Get/Put` interfaces.

```
// An interface for the bus master
// Param datasize   : The width of the databus
// Param addrsize   : The width of the addressbus
// Param granularity: Size of the smallest addressable unit
interface BusMaster #(numeric type datasize,
                      numeric type addrsize,
                      numeric type granularity);
    // Frontend
    interface Put #(Chunk #(datasize, addrsize, granularity)) job_send;
    interface Get #(Chunk #(datasize, addrsize, granularity)) job_done;

    // Backend
    method Bool valid;
    method Action granted (Bool permission);
    method Action available (Bool availability);
    interface Put #(Chunk #(datasize, addrsize, granularity)) put_states;
    interface Get #(Chunk #(datasize, addrsize, granularity)) get_states;
endinterface
```

BusSlave.

The BusSlave gets `Read/Write` requests and issues `Responses`.

```
// An interface for the bus slave
interface BusSlave #(numeric type datasize,
                     numeric type addrsize,
                     numeric type granularity);
    // Front end
    interface Get #(Chunk #(datasize, addrsize, granularity)) job_recieve;
    interface Put #(Chunk #(datasize, addrsize, granularity)) job_done;

    // Backend
    method Bool is_address_valid (Bit #(addrsize) addr);
    interface Put #(Chunk #(datasize, addrsize, granularity)) put_states;
    interface Get #(Chunk #(datasize, addrsize, granularity)) get_states;
endinterface
```

Bus.

The Bus is the medium of communication. It contains the `Arbiter` which mediates the transactions.

```
// An interface for the bus fabric
interface Bus #(numeric type masters,
                numeric type slaves,
                numeric type datasize,
                numeric type addrsize,
                numeric type granularity);
    interface Put #(Chunk #(datasize, addrsize, granularity)) write_to_bus;
    interface Put #(Chunk #(datasize, addrsize, granularity))
    ↪  write_to_bus_slave;
    interface Get #(Chunk #(datasize, addrsize, granularity)) read_from_bus;
endinterface
```

The `Bus` is connectable with `BusMaster BusSlave`. The bus is also connectable with vectors of `BusMaster` and `BusSlave`.

Modules.

The following constructors are used to create the bus.

mkBusSlave.

Creates the BusSlave interface.

```
// Module defintion for a BusSlave
// Param lower_bound : Lower bound of the Slave address
// Param upper_bound : Upper bound of the Slave address
// Param id          : Integer id for the slave
module mkBusSlave #(Bit #(addrsize) lower_bound,
                    Bit #(addrsize) upper_bound,
                    Integer id) (BusSlave #(datasize, addrsize, granularity));
```

mkBusMaster.

Creates the BusMaster interface.

```
// Module defintion for a BusMaster
// Param id          : Integer id for the master
module mkBusMaster #(Integer id)
                    (BusMaster #(datasize, addrsize, granularity));
```

mkBus.

Creates the Bus interface.

```
// Module defintion for a Bus fabric
// Param master_vec : A vector of BusMasters
// Param slave_vec  : A vector of BusSlaves
module mkBus #(Vector #(masters, BusMaster #(datasize, addrsize, granularity))
                       master_vec,
               Vector #(slaves, BusSlave #(datasize, addrsize, granularity))
                       slave_vec)
              (Bus #(masters, slaves, datasize, addrsize, granularity));
```

14 CPU.

Packages.

```
import CPU ::  * ;
```

Description.

The CPU package includes a 2 stage pipelined inorder CPU.

CPU Interfaces.

The package includes a `CPU` interface that wraps around everything.

CPU.

The `CPU` interface consists of a `BusMaster` to connect with the Bus. It also has an `Imem` Interface.

```
// An interface ot out CPU to connect with the Instruction Memory and the Bus
// Param wordlength     : Wordlength of out CPU, 32-Bit onwards supported
// Param datalength     : Length of the data registers
// Param busdatalength  : Width of the databus for the bus interface
// Param busaddrlength  : Width of the addressbus for the bus interface
// Param granularity    : Size of the smallest addresable unit. eg 1 Byte in
↪  RAMs
interface CPU #(numeric type wordlength,
                numeric type datalength,
                numeric type busdatalength,
                numeric type busaddrlength,
```

```
                numeric type granularity);
    interface Imem #(wordlength) imem;
    interface BusMaster #(busdatalength,
                          busaddrlength,
                          granularity) bus_master;
endinterface

typedef Server #(Bit #(wordlength), Bit #(wordlength)) Imem #(numeric type
↪  wordlength);
```

Modules.

The CPU can be constructed using,

mkCPU.

Creates the CPU interface.

```
// Creates a minimal 2 stage inorder pipelined CPU
// Param cpu_id : ID of the CPU (only for identification during debug)
// Param rom    : A string containing the path of the init IMEM
module mkCPU #(Integer cpu_id, String rom) (CPU #(wordlength,
                                                  datalength,
                                                  busdatalength,
                                                  busaddrlength,
                                                  granularity))

    provisos (Add# (na, 32, datalength),
              Add# (nb, 32, busdatalength),
              Add# (nc, 16, datalength),
              Add# (nd, 16, busdatalength),
              Add# (ne, 8,  datalength),
              Add# (nf, SizeOf #(Opcode),  datalength),
              Add# (ng, 8,  busdatalength),
              Add# (nh, 1,  busdatalength),
              Add# (ni, busaddrlength, TAdd#(TMax#(datalength, busaddrlength),
              ↪  1)),
              Add# (wordlength,0, SizeOf #(Instruction #(wordlength))),
              Add# (nj, 16, TAdd#(wordlength, datalength)));
```

15 Vector Accelerators.

Packages.

```
import VectorUnary ::  * ;
```

Description.

The package includes a vector accelerator for unary operations.

Vector Interfaces.

The package includes a `VectorUnary` interface that wraps around everything.

VectorUnary.

The `VectorUnary` interface consists of a `BusMaster` to connect with the Bus and issue `Read/Write` requests to the memory. It also consists a `BusSlave` to respond to the requests from the CPU.

```
// Interface of the Vector accelerator
// Param datasize       : Datasize of the Registers
// Param vectordatasize : Number of bits that can be parallelly operated upon
// Param busdatasize    : Width of the databus
// Param busaddrsize    : Width of the address bus
// Param granularity    : The smallest addressable unit size
interface VectorUnary #(numeric type datasize,
                        numeric type vectordatasize,
                        numeric type busdatasize,
                        numeric type busaddrsize,
```

```
                        numeric type granularity);

    interface BusMaster #(busdatasize, busaddrsize, granularity) bus_master;
    interface BusSlave  #(busdatasize, busaddrsize, granularity) bus_slave;

endinterface
```

Modules.

The VectorUnary can be constructed using,

mkVectorUnary.

Creates the VectorUnary interface.

```
// Creates a vector unary accelerator
// Param address           : Memory mapped address of the accelerator
// Param temp_storage_size : Size of the temp. data storage FIFOFs
// Param id                : ID of the unit
module mkVectorUnary #(Bit #(busaddrsize) address,
                       Integer temp_storage_size,
                       Integer id) (VectorUnary #(datasize,
                                                  vectordatasize,
                                                  busdatasize,
                                                  busaddrsize,
                                                  granularity))
    provisos (Add #(na, datasize, busdatasize),
                Add #(nb, 1,        busdatasize),
                Add #(nc, SizeOf #(Opcode), busdatasize),
                Add #(nd, vectordatasize, busdatasize),
                Mul #(ne, granularity, vectordatasize),
                Add #(nf, PresentSize #(vectordatasize, granularity),
                ↪  PresentSize #(busdatasize, granularity)),
                Add #(ng, 8, vectordatasize),
                Add #(nh, 16, vectordatasize),
                Add #(ni, 32, vectordatasize),
                Add #(nj, 8,  busdatasize),
                Add #(nk, 16, busdatasize),
                Add #(nl, 32, busdatasize));
```

References.

References.

1. Memory-mapped I/O, Wikipedia, https://en.wikipedia.org/wiki/Memory-mapped_I/O

2. Machine Language, nand2tetris, https://www.nand2tetris.org/project04

3. Assembler, nand2tetris, https://www.nand2tetris.org/project06

4. AHB, Bluespec, https://github.com/B-Lang-org/bsc-contrib/blob/master/Libraries/AMBA_TLM2/AHB/AHB.pdf

5. TLM, Bluespec, https://github.com/B-Lang-org/bsc-contrib/blob/master/Libraries/AMBA_TLM2/TLM/TLM.pdf

6. Vector Processors, UIC, Page 16-21, https://www.cs.uic.edu/~ajayk/c566/VectorProcessors.pdf

7. Bluespec Reference Guide, Bluespec, https://github.com/B-Lang-org/Documentation/blob/master/Language_Spec/bsv-reference-guide.pdf

8. BSV by Example, Bluespec, https://github.com/B-Lang-org/Documentation/blob/master/Tutorials/BSV_by_Example_Book/bsv_by_example.pdf