Arrow Programmer’s Guide

[Controller (Control Unit + Decoder)](#_kmsz6tkcggyq)

[Register File](#_xe6f62qe7bsk)

[OffsetGen.vhd](#_c6n68kkoxpmc)

[Bank1.vhd (Bank.vhd)](#_97vdje35z8ez)

[RegisterFile.vhd](#_phwkm8st5665)

[RegFile\_OffsetGen.vhd](#_z0zt407sq3b8)

[ALU](#_yo46odlz5vxu)

[ALU\_lane.vhd](#_zbet7nw4gjym)

[MV\_Block.vhd](#_p1vznhjqxbcm)

[ALU\_with\_pipeline.vhd](#_bg1d3sy9tcyj)

[RegFile\_ALU.vhd](#_wpws6gdhk79h)

[Memory Unit](#_n1z0z9b54eco)

[AXI Side](#_xrwg444if11)

[MEM\_Bank.vhd](#_4p2tbev820vx)

[MemGen.vhd](#_ym8u9e4vey69)

[MEM\_Unit.vhd](#_enwv9y9al2kw)

[Miscellaneous](#_9xskfe6lds5k)

[Packages.vhd](#_93cw7oe7c3rv)

[ShiftRegister.vhd](#_u53ur8n29ctf)

[Comments and Remaining Tasks](#_jx4gzzwm7psl)

**Note**: Arrow is a work in progress. Every component we implemented was testbenched on its own, then tested again as part of the system. Though we think our implementation is correct, there might be some things that could be done differently or better. **Do not take our code for granted** and assume that everything is right. Also, always make sure that the code you are writing synthesizes properly (using Vivado tools) This can be done through a quick google search if you are unsure. Remember: this is a **HARDWARE** description language and you should be able to predict the hardware that will be inferred from your code, to avoid mistakes or unwanted behaviors.

# Controller (Control Unit + Decoder)

* The **decoder** (Decoder.vhd) divides the incoming vector instructions (received from MicroBlaze or scalar processor through AXI interface) into the respective fields according to the spec. Nothing fancy. Synthesizes to a few wires. The decoder also generates the lane idx, i.e. the lane where the instruction will be executed, based on the most significant bit (MSB) of the destination register. It is done this way because currently, the register banks are dedicated to their corresponding lanes (i.e. lower 16 registers or bank 0 to lane 0 and upper 16 regs or bank 1 to lane 1). This scheme was chosen for its simplicity but might need to be changed later to something more dynamic.
* The **control unit** (Control\_Unit.vhd) generates the appropriate control signals (still no concept of lanes here) for decoded instruction, based on opcode, funct3 and mew for memory. CSRs are located here for now but they can be later moved to the scalar core.
  + There are 3 processes
    - 1 clocked process for reading/writing CSRs. Writes require a CSR\_WEN (write enable) to be performed. vstart here needs to be reset at the end of each instruction according to spec, but we haven’t implemented this yet (keeping it 0 always).
    - 1 combinational process to generate appropriate control signals
    - 1 clocked process for vsetvl/vsetvli. Clocked because it might write to CSRs.
* The **controller** (Controller.vhd) combines the control unit, decoder and logic to send signals to their appropriate **lanes**. We did this by having output ports the size of the signal multiplied by the number of lanes. That way, we write to the part of the port related to the appropriate lane. Note that all our code was written in a generic way so any change in the number of lanes, number of registers, etc. should not affect the correctness of our code. However, changing the number of lanes for example will lead to less register per lanes[[1]](#footnote-0) and may thus affect the correctness indirectly. Again, please do not take our code for granted.
  + newInst is necessary to tell the lane that a new instruction is coming so it can reset its counters and prepare for it.

# 

# Register File

## OffsetGen.vhd

* **Short description:** OffsetGen is responsible for deciding which bytes of the 64-bit data chunk are to be written. Given the 64-bit data chunks, we have 8 bytes hence the 8-bit size of the WriteEnSel vector, where each bit represents 1 byte of data. If a bit is a 0, the data isn’t to be written.
* **In-depth process explanation:**
  + The process’s function is to set the bits of the WriteEnSel vector each cycle.
  + First, we get the first element to start from using v\_start. We also find the number of elements either from vl or vlmax.
  + The number of elements per transfer is 64/SEW **at most**. (For example, if SEW=16 bits, we have at most 4 elements in our 64-bit data chunk.) We will need this value as our upper bound.
  + In the process, we treat each transfer case differently.
  + If it is the first transaction: the lower bound is v\_start. Else, it is 0.
  + If elements left is greater than number of elements per transfer: more than 1 transfer is left, so the upper bound is number of elements per transfer.
  + If elements left is less than number of elements per transfer: then it could be the last transfer or the first and last transfer (if total number of elements for this instruction was less than the number of elements possible per transfer). Hence, the upper bound is elements left and **NOT** number of elements per transfer.
  + The loop: considers the SEW values and sets the WriteEnSel bits accordingly. This is where masking is taken into consideration. If masking is enabled and the mask bit is 0, we don’t set WriteEnSel bits. The loop goes from 0 to 7, since a 64-bit chunk can have 8 elements at max (when SEW is minimum which is 8-bits).
  + Illustration example:
    - Assume SEW=16 bits, and in the current transfer we have 1 enabled element, followed by 2 disabled elements and 1 enabled element
    - WriteEnSel would be as follows: **11000011**
    - Red: first element
    - Green: second element
    - Yellow: third element
    - Blue: fourth element
    - Notice that each element is represented by 2 bits of WriteEnSel, since SEW=16-bits which is 2 8-bits.
    - The process keeps on going until there are no elements left.

## Bank1.vhd (Bank.vhd)

* **Short description:** Each bank contains a subset of the vector registers. In our case, since we have 2 banks, each bank has 16 vector register. The reason we have 2 vhd files instead of 1 is that they are asymmetric; Bank1.vhd has the mask register(v0) as output which the other banks in order to read.
* **Important note:** the bank is blind to the concept of an element size. It only sees an incoming 64-bit data chunk and based on WriteEnSel decides if to write a certain byte and based on w\_offset\_int decides where to write it. w\_offset\_int is basically the hop size **IN BYTES**.
* **In-depth writing process explanation:**
  + The process’s function is to decide **WHERE** to write the data based on the WriteEnSel bits and w\_offset\_int.
  + Illustration example:
  + Assume WriteEnSel is 11111111
  + If w\_offset\_int is 0, we write incoming 64 bits in a certain vector register WriteDest at an offset 0.
  + If w\_offset\_int is 8, we write the incoming 64 bits at an offset of 8 bytes, which is adjacent to the 8 bytes we previously wrote when w\_offset\_int was 0.
  + The reason cases were written instead of a neater loop is due to synthesis purposes.
* **In-depth reading process explanation:**
  + The process’s function is to read data based on the WriteEnSel bits and r\_offset\_int. Note that reading is **NOT** clock dependent unlike writing.
  + The reason cases were written instead of a neater loop is due to synthesis purposes.
  + It is built in the same fashion as the writing process.

## RegisterFile.vhd

* **Short description:** RegisterFile is the top level containing two banks. One bank is Bank1.vhd, and the other is a Bank.vhd instances. If we were to change the number of banks to 4 for example, we would still have 1 instance of Bank1.vhd and 3 instances of Bank.vhd.
* Code-wise, it is pretty straight forward. The code is composed component declarations and generates.
* The mapping of the RegisterFile ports to each of the bank has the following scheme:
  + Illustration example:
    - sew: in STD\_LOGIC\_VECTOR (3\*NB\_LANES-1 downto 0); is one of RegisterFile’s ports.
    - sew vector is a 6-bit vector assuming 2 lanes, which is double the size of the SEW field.
    - Lane 0 takes the lower 3 bits, and lane 1 takes the upper 3 bits. In our scheme, Lane 0 always takes the lower x bits of a certain port.

## RegFile\_OffsetGen.vhd

* **Short description:** RegFile\_OffsetGen is the top level containing RegisterFile (the two banks) and two OffsetGen instances, one OffsetGen for each Bank. One bank is Bank1.vhd, and the other is a Bank.vhd instances. If we were to change the number of banks to 4 for example, we would still have 1 instance of Bank1.vhd and 3 instances of Bank.vhd.
* Code-wise, it is pretty straight forward. The code is composed component declarations and generates.

# ALU

## ALU\_lane.vhd

* **Short description:** ALU\_lane.vhd is the entity that performs the arithmetic instructions. Only a subset of the instructions was implemented. Please check the report for further details.
* ALU\_lane is not sensitive to the clock. It is just combinational logic.
* The functions were implemented as cases due to synthesis purposes.
* Based on sew\_int, we decide how many bits to operate on from the incoming 64-bit data chunks.
  + Illustrative example:
    - Assume sew\_int = 16, and the arithmetic instruction is an add
    - The lane basically performs 4 adds in parallel, 4 being 64-bit/16-bit (sew\_int). The first 16-bit of the first operand is added with the first 16-bits of the second operand, then the second 16-bits, etc.
* The implementations of the instructions are pretty straightforward.

## MV\_Block.vhd

* **Short description:** MV\_Block.vhd implements the merge and move instructions. Check the ISA for an in-depth description of the merge and move instructions.
* We decided to implement these two instructions in a component of their own rather than with ALU\_lane.
* In our previous implementation of 1 element per cycle, MV\_Block was very straight forward. However, with the new 64-bit data chunk/transfer implementation**, MV\_Block requires more refinement to correctly implement the merge and move instructions**. It is currently implemented such that the instructions are partially correct just for the sake of moving data to populate the vector registers in our testing phase.

## ALU\_with\_pipeline.vhd

* **Short description:** ALU\_with\_pipeline.vhd is the entity containing two instances of ALU\_lane.vhd and MV\_Block.vhd each.
* **Pipelining:**
  + Since we have various inputs coming at different clock cycles (e.g., the data being read from the register file, the offsets and WriteEnSel coming from OffsetGen …etc.) we pipeline the incoming signals such the signals coming from OffsetGen are pipelined only once since they already take 1 cycle to be outputted from OffsetGen, and other signals (such as that coming from Controller.vhd) are pipelined 2 cycles to be in line with the signals coming from OffsetGen and RegFile\_OffsetGen.vhd.
  + We pipeline the WriteEnSel and w\_offset since they will be used in the write-back to the banks once the results from ALU\_lane or MV\_Block are ready.
* **The done signal should be used as a way to flush the signals once an instruction is done, and it is also used in software testing, but it needs refinement as to when it is actually set and when it actually flushes.**

## RegFile\_ALU.vhd

* **Short description:** RegFile\_OffsetGen is the top level containing RegFile\_OffsetGen and ALU\_with\_pipeline.
* Code-wise, it is pretty straight forward. The code is composed component declarations and mappings.

# Memory Unit

Memory has 2 “sides”: AXI side and Arrow side. We still did not connect both sides, and do not have a fully working memory system yet.

## AXI Side

* The AXI side of memory is attached as an IP that can be directly added to Vivado. It is located in a zip in src/AXI/.
* The AXI side interfaces with the memory system through the AXI bus. It receives the request from Arrow through direct connections (simple port map) then generates the AXI memory requests using a custom AXI master interface and sends them to the MIG. MIG takes care of translating the AXI request to be compatible with the DDR memory. Note that the MIG is shared with the MicroBlaze, so you can write from MicroBlaze and read from Arrow and vice versa.
  + The AXI master is based on <https://github.com/Architech-Silica/Designing-a-Custom-AXI-Master-using-BFMs>
  + It was modified to work without the FIFO (commented all code related to FIFO in AXI\_master\_transaction.vhd).

Note that a modified version of the FIFO might be needed to support data transfers of varying width (varying sew), but this will require supporting narrow instructions and implementing them according to the AXI spec. I prefer not to share the code for the modified FIFO that I had started working on because it is very scratch-like and not properly tested. However, I can confirm that the AXI master interface without the FIFO works well.

* + The way memory is supposed to work now, is that Arrow sends the starting memory address to load from and the number of bytes to load. The number of bytes is translated to beats to be able to perform the load in bursts, thus making the operation faster. The problem with this approach is that it assumes that data are located contiguously; in the case where they are not, this will lead to wasted cycles (because it’s fetching useless data). The AXI master also supports strided accesses but they have not been tested by us.
  + For testing purposes, we modified “mem\_address\_gen\_with\_FIFO\_v1\_0.vhd” (e.g. port map on line 314) to test specific functionalities without worrying about other components generating the correct signals.

## MEM\_Bank.vhd

* **Short description:** MEM\_Bank.vhd is just an array whose purpose is to simulate the memory banks. **DO NOT INCLUDE WHEN PACKAGING THE IP**. Only use in testing.

## MemGen.vhd

* **Short description:** MemGen.vhd is the entity responsible for outputting the memory addresses and offsets for the data to be fetched from the memory banks. MemGen.vhd currently supports masked and unmasked operations.
* Ideally, MemGen.vhd should support **unit-strided, strided, and indexed.** Refer to the spec sheet to find the definitions of each mode.
* **Unit-strided** and **strided** have been implemented. **Indexed** is not yet implemented fully, but a rough structure has been written but not tested.
* Note that MEMWIDTH has a different encoding than SEW.
* A key idea to keep in mind is that we still are dealing with a 64-bit data chunk, where WriteEnMemSel is an 8-bit vector such that each bit corresponds to a byte of the 64-bit data chunk. Our goal is to set the WriteEnMemSel bits depending on:
  + Stride: is it a unit stride (one byte) or some custom number?
  + MEMWIDTH: this determines how many WriteEnMemSel bits we will flick “on” everytime we have an element to write.
* **Process Explanation:**
  + **Unit-stride**: works similar to OffsetGen.vhd. Refer to the process explanation in OffsetGen.vhd for more details.
  + **Strided:** the important concept to understand here is FRAGMENTS and when they occur:
    - Illustration example:

Assume we have 16-bit (2-byte) elements and a 24-bit (3-byte stride).

WriteEnMemSel will look something like this: 01100011

Where the red is the first element, green is the first stride, blue is the second element, and the purple is a **FRAGMENT** of the stride. This means that on the next cycle, we have to keep track of the remaining fragment (i.e, two zeroes) before we continue with our elements.

Next cycle: ...1100 where the purple is the remainder of the stride from the previous cycle, and the beige is the third element.

## MEM\_Unit.vhd

* **Short description:** MEM\_Unit.vhd is to be used for testing purposes only, as it includes MEM\_Bank.vhd. MEM\_Unit.vhd is just a toplevel including an instance of MemGen.vhd and MEM\_Bank.vhd each.

# Miscellaneous

## Packages.vhd

* **Short description:** Packages.vhd contains the generic declarations utilized throughout the project. We opted for this file to reduce redundancies across our files and use only this file as our reference.

## ShiftRegister.vhd

* **Short description:** ShiftRegister.vhd **COULD** be utilized in testing the memory unit with RegFile\_ALU\_Controller to simulate the number of cycles needed by the memory interface to get the needed data. The number of cycles incurred by ShiftRegister.vhd can be changed using the variable found in the file.
* This file still hasn’t been used in a full fledged system simulation, but it might come in handy.

# Comments and Remaining Tasks

* Looking back, we think we made a mistake by including pipelines in the components themselves (e.g. in ALU or controller). A better, more traditional way, is to do the components alone, and separate them with pipeline registers of appropriate size. To understand what I mean, check out the implementation of a standard 5-stage pipeline RISC-V scalar core and see how it is implemented. Our biggest mistake was that we dived straight into coding without looking at other implementations or reading the literature. We managed to overcome this problem but the code we wrote might still have some silly mistakes that were overlooked or ignored, even after revision. We invite you to look at the code with a critical eye.
* **Chaining**: To support chaining, we need to be able to read from other banks and this will require changing the concept of dedicated banks to lanes. The banks and register file will require some modification. A controller needs to be implemented to manage chaining. This may be a different component or part of controller.vhd.

1. Since our banks are dedicated to lanes and since we don’t use virtual registers. [↑](#footnote-ref-0)