Harmonica Microarchitecture Specifications

Parameterizable SystemVerilog Implementation

RTL 0.0

Computer Architecture and Systems Laboratory

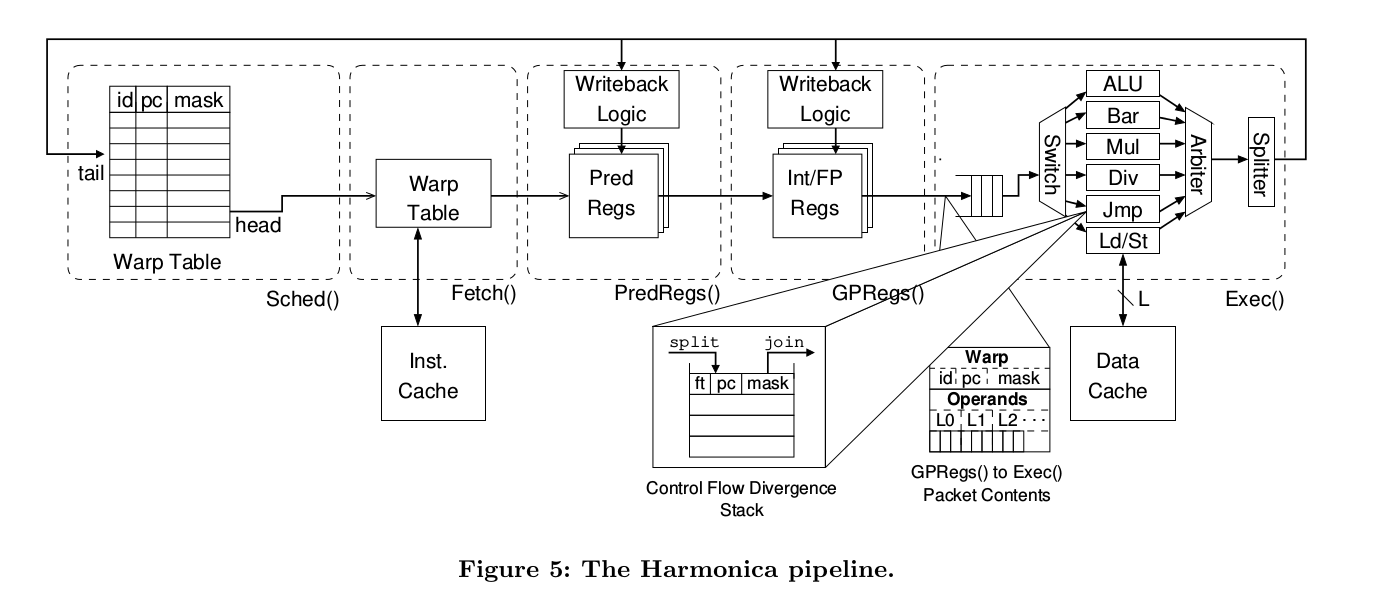
Georgia Institute of Technology

**Section 1 - Harmonica RTL 0.0 Overview**

Harmonica is a SIMT core designed for intelligent 3D Stacked DRAM. This will enable Processing in Memory (PIM). As data transfer energy cost is becoming larger than ALU energy cost, there is a need for a new architecture shift. PIM offers performance and energy benefits as it is able to receive large bandwidth from memory without the need of lengthy bus connections.

The details of the Harmonica architecture is found in “*Lightweight SIMT Core Designs for Intelligent 3D Stacked DRAM*” by Chad D. Kersey, Sudhakar Yalamanchili, and Hyesoon Kim. Another useful document to refer to is “*HARP Instruction Set Manua*l” by Chad D. Kersey.

**Pipeline Block Diagram of the Harmonica Core**



*(Figure taken from “Lightweight SIMT Core Designs for Intelligent 3D Stacked DRAM” )*

Form the pipeline diagram above, we can see that there are 5 stages. The first stage is Sched(), which will contains the Warp Table. The warp table is responsible for keeping track of all of the warp information of different programs, while continuously generating new ones while replacing old ones in a FIFO manner. The second stage is Fetch(), in which the processor will use the warp table information to find the correct program counter and read from the Instruction cache. After that, the hex instruction found in the program counter will get decoded to control the rest of the pipeline. The next two stages include PredRegs() and GPRegs(), which will register files equal to the number of threads supported in the processor. The number of threads is equal to the number of warps multiplied by the number of threads per warp. The final stage of the pipeline is the execution stage which house the arithmetic logic unit, floating point unit, memory interfaces for load word and store word to the data caches, jump instruction,etc. Following the execution unit will generate a new packet to be inserted into the warp table to have a continuous round robin scheduling algorithm. Also at this time, the writeback logic will occur to write any necessary data back into the predicate or general purpose register files.

**Contact Information:**

|  |  |  |
| --- | --- | --- |
| **Name** | **Project Role** | **Email** |
| Sudhakar Yalamanchili | Principal Investigator | [sudha@ece.gatech.edu](mailto:sudha@ece.gatech.edu) |
| Chad Kersey | Harmonica Architect | [cdkersey@gatech.edu](mailto:cdkersey@gatech.edu) |
| Eric Qin | RTL 0.0 Architect | [ecqin@gatech.edu](mailto:ecqin@gatech.edu) |

**RTL Roadmap:**

|  |  |  |
| --- | --- | --- |
| **Milestone** | **Target Completion** | **Completion Status** |
| RTL 0.0 | 1. Create the basic pipeline infrastructure and all of the key RTL components of the Harmonica core 2. Support integer operation instructions 3. Support basic jump, load, and store operations 4. Support predicated instructions 5. Create a verification infrastructure to read the values of the many register files | The target is met. Concerns include potential need for change in the Fetch() stage for timing reasons. (Decoder and instruction fetch are currently in the same stage) . Some variant of jump instructions are yet to be implemented. More test cases and warp executions are needed to confirm functionality. Also, only one configuration combination was tested.  -Eric Qin (Fall 2017 Special Problems) |
| RTL 0.5 | 1. Add control divergence handling (split and join instruction) 2. Update the memory interface 3. Create a test case checker to compare the expected register values with the actual. |  |
| RTL 0.8 | 1. Add floating point functionality and delays 2. Add all remaining instructions that are yet to be implemented 3. Start debugging by enabling different parameters |  |
| RTL 1.0 | 1. Verify design with automatically generated test cases and cross compare between the CHDL implementation and the SystemVerilog implementation 2. Check for corner cases and gather coverage details 3. Upload to FPGA |  |

**Section 2 - Project Files Description**

This section provides a brief overview of every file in the project design. Additional details are found further down the document.

* **decoder.sv**: The decoder will evaluate the HEX instruction and generate control signals that will be used throughout the pipeline. This include read/ write enables, addresses, amd ALU controls.
* **executionUnits.sv**: This is a top level file that instantiate multiple functional units (from the functionUnits.sv file) in the Exec() stage. This is created using a generate loop that parses the input vectors to the correct lane while generating an output vector for writeback purposes.
* **functionalUnits.sv**: Arithmetic unit that calculates integer operations. Multiple of these are instantiated by executionUnits.sv.
* **harmonica.sv**: The heart of the Harmonica RTL. This file connects all of the components together while maintaining timing. Contains all of the data flip flop sin between each stage to maintain instruction flow throughout the pipeline.
* **harmonica\_cfg.sv**: Contains parameters for the user to change such as number of warps and SIMD lanes. Also contains definition and structure declarations.
* **harmonica\_tb.sv** (*Verification component)*: This file is the Harmonica verification infrastructure top level. This instantiate the Harmonica RTL, simple instruction and data caches, and the register trackers.
* **harmonica\_tracker.sv** *(Verification component)*: While running, it will keep track of any changes to the general purpose register block (all of the register files), and print it out to a text file.
* **harmonica\_tracker\_pred.sv** *(Verification component)*: Similar to harmonica\_tracker.sv, but rather tracking the general purpose register block, it tracks the predicate register block.
* **register\_block.sv**: This is a top level file that instantiate multiple register files based on the number of threads supported in the core. This generic file is used for both the general purpose and predicate register blocks in the pipeline.
* **register\_file.sv**: This file mimics a simple register file. It reads at the positive edge of clock while writing on the negative edge of clock.
* **warpTable.sv**: The warp table is a FIFO warp scheduler. It include the starter code to initiate the starting point for each warp, while continuously writing new data into the table and adjusting the head & tail pointer.

**Section 3 - Parameters and Structures Definitions**

This section introduces the basic parameters and structure of the Harmonica RTL design. Note: While creating RTL 0.0, the parameters used were: Architecture Size = 64, Number of registers per register file = 32, Number of SIMD Lanes = 8, Number of Warps = 8.

**Parameters Setting**s

The Harmonica RTL 0.0 aim to support different configurations.

Architecture Size → 32 or 64 bit

Number of registers per register file → 16, 32, or 64

Number of SMD Lanes → 4 or 8

Number of Warps → 4 or 8

For more parameter information, please look in the harmonica\_cfg.sv file. The file will define parameters that are used across all of the files within the Harmonica project. The base parameters will define additional parameters that will be used as macros in many I/O and internal signals. The purpose of this is to make the whole System Verilog implementation parameterizable. This serve as the base implementation; however, more debugging will be needed to make everything work properly.

**Structure Information**

The use of structures is essential for passing warp and decoder output information throughout the pipeline stages. Structures make the code easier to read and debug. Structure are defined in the bottom of the harmonica\_cfg.sv file and are initialized in the harmonica.sv file. Below include detailed tables of the internal signals found in each structure.

|  |  |
| --- | --- |
| **flopControl\_t** – Passes decoder logic throughout the pipeline. For example, register block enables, read enables based on the operand, write enables, and ALU operation control. All of the signals in this structure is generated from the decoder! (Include: fetchControlBus, predregControlBus, gpregControlBus and execControlBus) | |
| **Signal Name and Width** | **Description** |
| instruction, <MACHINE\_WIDTH-1:0> | instruction hex values found in the instruction cache |
| GPR\_read\_en, <0> | enable bit to read from the general purpose register block |
| PRED\_read\_en, <0> | enable bit to read from the predicate register block |
| GPR\_write\_en, <0> | enable bit to write to the general purpose register block |
| PRED\_write\_en, <0> | enable bit to write to the predicate register block |
| dec\_predicated\_bit, <0> | predicate bit for predicate |
| dec\_pred\_operand, <0> | predicate operand value |
| gpr\_rd\_en\_0, <0> | GPR enable bit to read the first register address in the instruction encoding |
| gpr\_rd\_en\_1, <0> | GPR enable bit to read the second register address in the instruction encoding |
| gpr\_rd\_en\_2, <0> | GPR enable bit to read the third register address in the instruction encoding |
| gpr\_rd\_addr\_0, <LOG2\_NUM\_REGS-1:0> | the first register address in the instruction encoding to the general purpose register block |
| gpr\_rd\_addr\_1, <LOG2\_NUM\_REGS-1:0> | the second register address in the instruction encoding to the general purpose register block |
| gpr\_rd\_addr\_2, <LOG2\_NUM\_REGS-1:0> | the third register address in the instruction encoding to the general purpose register block |
| pred\_rd\_en\_0, <0> | PRED enable bit to read the first register address in the instruction encoding |
| pred\_rd\_en\_1, <0> | PRED enable bit to read the second register address in the instruction encoding |
| pred\_rd\_en\_2, <0> | PRED enable bit to read the third register address in the instruction encoding |
| pred\_rd\_addr\_0, <LOG2\_NUM\_REGS-1:0> | the first register address in the instruction encoding to the predicate register block |
| pred\_rd\_addr\_1, <LOG2\_NUM\_REGS-1:0> | the second register address in the instruction encoding to the predicate register block |
| pred\_rd\_addr\_2, <LOG2\_NUM\_REGS-1:0> | the third register address in the instruction encoding to the predicate register block |
| dec\_alu\_ctl <4:0> | control signal for ALU operation select |
| pc\_jump, <0> | instruction program counter jump control signal |
| mem\_read, <0> | data memory read control signal (active high) |
| mem\_write, <0> | data memory write control signal (active high) |

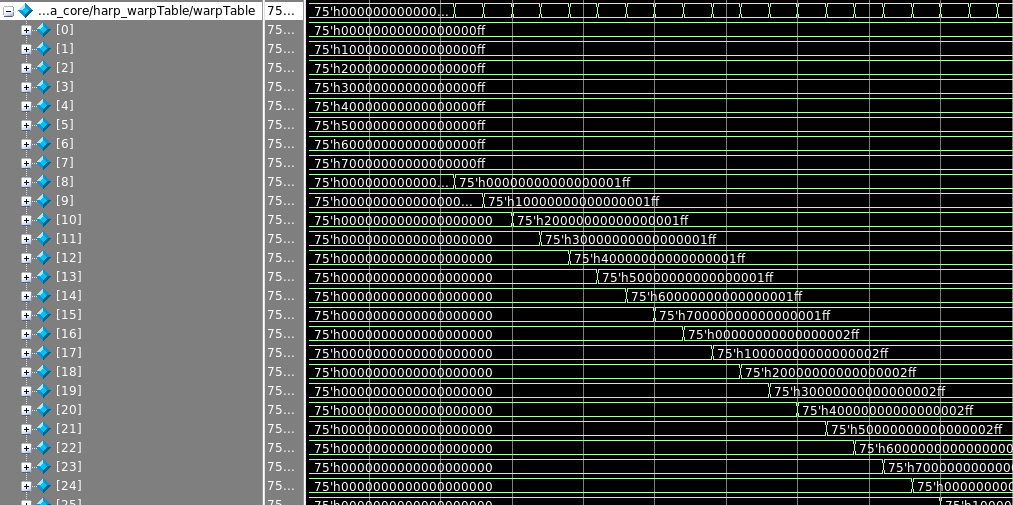
|  |  |
| --- | --- |
| **flopWarpData\_t** – Passes warp information throughout the pipeline. This will be essential for keeping track which warp is in what stage of the pipeline. It also aid the timing of the warp table. (Include: schedWarpBus, fetchWarpBus, predregWarpBus, gpregWarpBus, execWarpBus) | |
| **Structure Signal Name and Width** | **Description** |
| warpID, <LOG2\_NUM\_WARPS-1:0> | informs what warp the stage is working on |
| pc, <MACHINE\_WIDTH-1:0> | informs what pc the stage is working on |
| mask, <NUM\_THREADS\_PER\_WARP-1:0> | determines what thread of the warp to mask out (for future branch divergence implementation) |

**Section 4 – Schedule Stage RTL Components Overview**

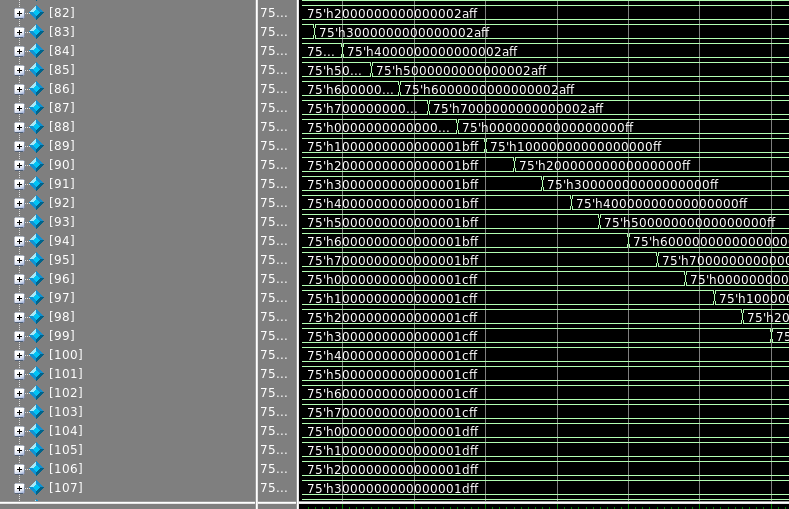
Whenever reset is enabled, the warp table (warpTable.sv) will initialize to its starter code. It will enable eight warps all starting at the PC value of 0x0 as the boot address. The boot address can be changed by the user. Then, each warp will continue going to the next entry in the instruction cache. A new packet will be inserted to the next entry of the warp table during the cycle after the execute stage. The packet includes the warp ID, pc, and mask. Tail and head pointers are adjusted accordingly to model a FIFO scheduler structure. The pointer go in a circular fashion to reduce warp table size. In RTL 0.0 implementation, the FIFO depth is set to 128. The warp controls and structure declaration are found in the schedule portion of the harmonica.sv file. The schedWarpBus will be assigned to the head warp table packet, and will be flopped to the next cycle which is the fetchWarpBus.

|  |  |  |
| --- | --- | --- |
| **Warp Table I/O signals** | | |
| **Signal Name and Direction** | **Width** | **Description** |
| clk (input) | [0] | clock signal |
| reset (input) | [0] | reset signal |
| read\_packet\_en (input) | [0] | read packet enable |
| read\_packet\_data (output) | [LOG2\_NUM\_WARPS+MACHINE\_WIDTH+NUM\_THREADS\_PER\_WARP-1:0] | read packet data, which includes the warp id, pc, and the mask |
| write\_packet\_en (input) | [0] | write packet enable |
| write\_packet\_data (input) | [LOG2\_NUM\_WARPS+MACHINE\_WIDTH+NUM\_THREADS\_PER\_WARP-1:0] | write packet data, which includes the warp id, pc, and the mask |
| fifo\_is\_empty (output) | [0] | signal when fifo is empty |
| fifo\_is\_full (output) | [0] | signal when fifo is full |

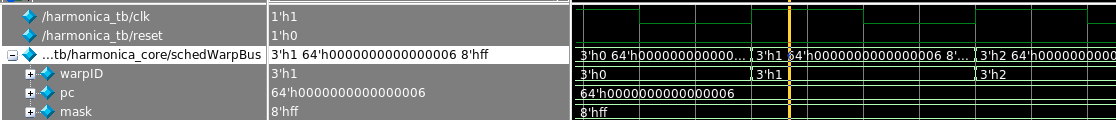
The warps will be executed in a round robin fashion to avoid dependency hazards. Unless there is a jump instruction, the next PC value will be next PC in the instruction cache. (Only jmpi was implemented in RTL 0.0)



*Figure of the initialized warp table (with the starter values) and generated warp table packets after each cycle.*



*Figure of the warp table with a jump instruction. Notice how the PC value changes from 0x2a to 0.*

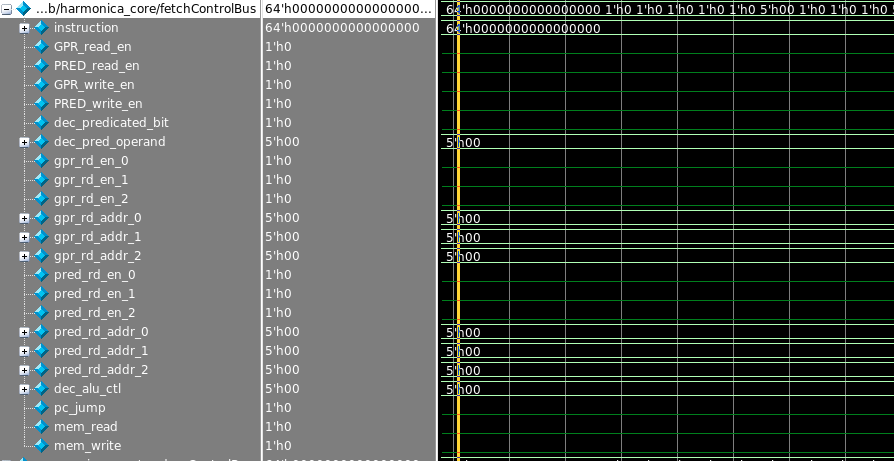
**

*Figure: Assigning the warp table packet to the schedWarpBus structure.*

**Section 4 – Fetch Stage RTL Components Overview (And Implemented Instructions)**

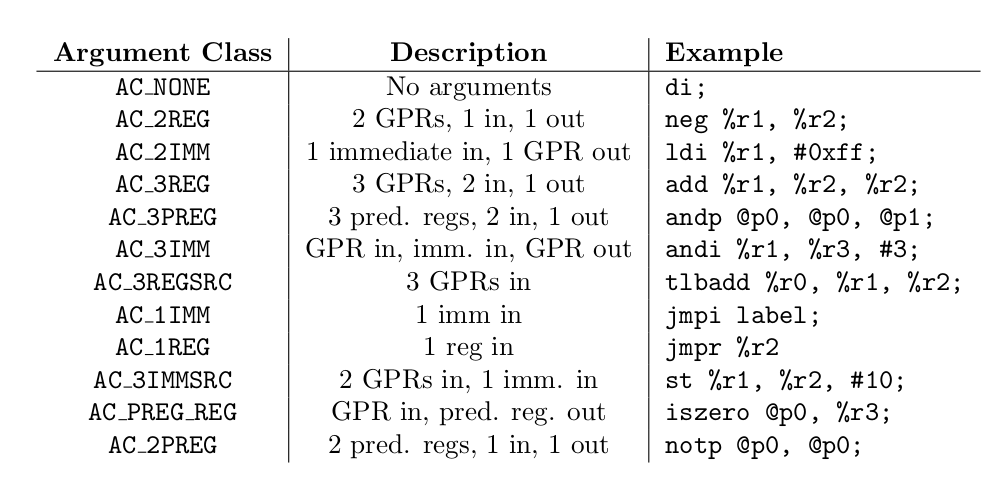
In the fetch stage, the first step is to get the PC counter value from the warp data structure. (found in the harmonica.sv file) With the PC counter, it sends an instruction cache request to get the back the hex encodings. The instruction cache is found harmonica\_tb.sv. The hex encodings are generated through the Harptool disassembler. Details are found later in the document. Then, the hex instruction will go into the decoder (found in decoder.sv file) and outputs essential control signals to the fetchControlBus. The control and warp structures then flop to the next stage.

|  |  |  |
| --- | --- | --- |
| **Decoder I/O signals** | | |
| **Signal Name and Direction** | **Width** | **Description** |
| dec\_instruction (input) | [MACHINE\_WIDTH-1:0] | the instruction encoding to be decoded |
| GPR\_read\_en (output) | [0] | enable bit to read from the general purpose register block |
| PRED\_read\_en (output) | [0] | enable bit to read from the predicate register block |
| GPR\_write\_en (output) | [0] | enable bit to write to the general purpose register block |
| PRED\_write\_en (output) | [0] | enable bit to write to the predicate register block |
| dec\_predicated\_bit (output) | [0] | predicate bit for predicate dependent instructions |
| dec\_pred\_operand (output) | [LOG2\_NUM\_REGS-1:0] | predicate operand value |
| gpr\_rd\_en\_0 (output) | [0] | enable bit to read the first register address in the instruction encoding to the general purpose register block |
| gpr\_rd\_en\_1 (output) | [0] | enable bit to read the second register address in the instruction encoding to the general purpose register block |
| gpr\_rd\_en\_2 (output) | [0] | enable bit to read the third register address in the instruction encoding to the general purpose register block |
| gpr\_rd\_addr\_0 (output) | [LOG2\_NUM\_REGS-1:0] | the first register address in the instruction encoding to the general purpose register block |
| gpr\_rd\_addr\_1 (output) | [LOG2\_NUM\_REGS-1:0] | the second register address in the instruction encoding to the general purpose register block |
| gpr\_rd\_addr\_2 (output) | [LOG2\_NUM\_REGS-1:0] | the third register address in the instruction encoding to the general purpose register block |
| pred\_rd\_en\_0 (output) | [0] | enable bit to read the first register address in the instruction encoding to the predicate register block |
| pred\_rd\_en\_1 (output) | [0] | enable bit to read the second register address in the instruction encoding to the predicate register block |
| pred\_rd\_en\_2 (output) | [0] | enable bit to read the third register address in the instruction encoding to the predicate register block |
| pred\_rd\_addr\_0 (output) | [LOG2\_NUM\_REGS-1:0] | the first register address in the instruction encoding to the predicate register block |
| pred\_rd\_addr\_1 (output) | [LOG2\_NUM\_REGS-1:0] | the second register address in the instruction encoding to the predicate register block |
| pred\_rd\_addr\_2 (output) | [LOG2\_NUM\_REGS-1:0] | the third register address in the instruction encoding to the predicate register block |
| dec\_alu\_ctl (output) | [4:0] | basic ALU operation select |
| pc\_jump (output) | [0] | instruction program counter jump control signal |
| mem\_read (output) | [0] | data memory read control signal (active high) |
| mem\_write (output) | [0] | data memory write control signal (active high) |

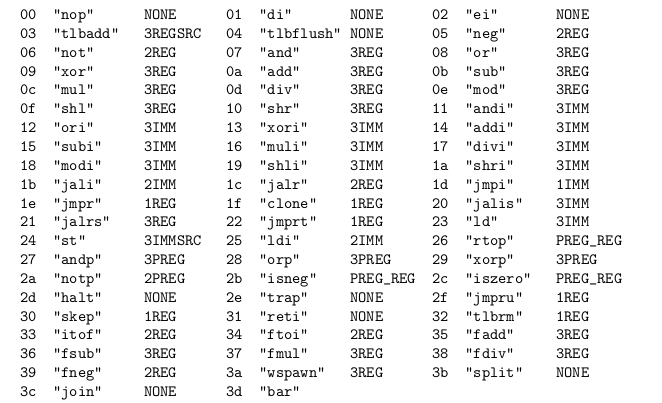


*Figure: Decoder output for a nop instruction*

The figures below show the argument classes and instruction set of the Harmonica core. The control output from the decoder depends on the instruction’s argument class and operation.



*Figure from the HARP Instruction Set Manual: HARP Argument Classes*



*Figure from the HARP Instruction Set Manual: HARP Instructions*

The supported instructions for RTL 0.0 are listed below. Most have been tested with one directed test case. However, due to the scale of the RTL project, much more tests will be needed to ensure coverage and correct functionality.

**Implemented Integer Operations:**

NEG (2REG) , NOT (2REG) , AND (3REG) , OR (3 REG) , XOR (3 REG), ADD (3 REG) , SUB (3 REG) , MUL (3 REG) , DIV (3 REG), MOD (3 REG) , SHL (3 REG), SHR (3 REG) , ANDI (3 IMM), ORI (3 IMM), XORI (3 IMM) , ADDI (3 IMM), SUBI (3 IMM), MULI (3 IMM), DIVI (3 IMM) , MODI (3 IMM) , SHLI (3 IMM) , SHRI (3 IMM)

**Implemented Floating Point Operations:**

None implemented in RTL 0.0.

**Implemented Control Operations:**

JMPI (1 IMM)

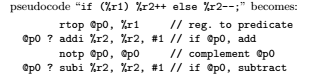
**Data Operations (Will need a new memory subsystem):**

LD (3 IMM), ST (3 IMMSRC)

**Divergence and Predicate Operations**

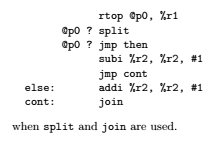
ANDP (3 PREG) , ORP (3 PREG) , XORP (3 PREG), NOTP (2 PREG) , ISNEG (PREG\_REG), ISZERO (PREG\_REG)

Additional, the if/then/else predication is implemented. The instruction style is shown below.



*Figure from Lightweight SIMT Core Designs Paper: Predication example*

The next step is to handle branch divergence with split and join instructions with predication. This has not yet been correctly implemented in RTL 0.0.



*Figure from Lightweight SIMT Core Designs Paper: Split and Join methodology*

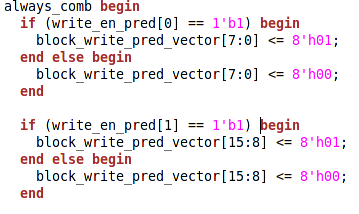
**Section 5 – Predicate and General Purpose Stages RTL Components Overview**

The predicate and general purpose are separate stages, with the general purpose stage one cycle later than the predicate stage. The reason why they are described in the same section is because they share many similarities. For example, they both instantized a lot of register files.

|  |  |  |
| --- | --- | --- |
| **Register Block I/O signals (Used for General Purpose and Predicate Register Block)** | | |
| **Signal Name and Direction** | **Width** | **Description** |
| clk (input) | [0] | clk signal |
| reset (input) | [0] | reset signal |
| warp\_number\_read (input) | [LOG2\_NUM\_WARPS-1:0] | the warp identifier for which register files to read from |
| warp\_number\_write (input) | [LOG2\_NUM\_WARPS-1:0] | the warp identifier for which register files to write to |
| block\_read\_en (input) | [0] | enables reading to the register block |
| thread\_en\_vector (input) | [NUM\_TOTAL\_THREADS-1:0] | enable bit for each register file based on if the thread is valid or not (handle branch divergence) |
| read\_en\_0 (input) | [0] | enable reading for the first entry |
| read\_en\_1 (input) | [0] | enable reading for the second entry |
| read\_en\_2 (input) | [0] | enable reading for the third entry |
| read\_addr\_0 (input) | [LOG2\_NUM\_REGS-1:0] | address of the first entry to be read |
| read\_addr\_1 (input) | [LOG2\_NUM\_REGS-1:0] | address of the second entry to be read |
| read\_addr\_2 (input) | [LOG2\_NUM\_REGS-1:0] | address of the third entry to be read |
| write\_en (input) | [0] | write enable to the register block |
| write\_en\_pred (input) | [NUM\_LANES-1:0] | write enable based on the predicate mask |
| write\_addr (input) | [LOG2\_NUM\_REGS-1:0] | write address to the register block |
| write\_data\_vector\_0 (input) | [NUM\_LANES\*MACHINE\_WIDTH-1:0] | write data vector to the register block |
| read\_data\_vector\_0 (output) | [NUM\_LANES\*MACHINE\_WIDTH-1:0] | read data vector 1 from the register block |
| read\_data\_vector\_1 (output) | [NUM\_LANES\*MACHINE\_WIDTH-1:0] | read data vector 2 from the register block |
| read\_data\_vector\_2 (output) | [NUM\_LANES\*MACHINE\_WIDTH-1:0] | read data vector 3 from the register block |

Register block instantiates multiple register files based on the total number of threads the Harmonica micro-architecture is configured to. The value will be the product of the number of supported warps and the number of lanes in the system. The register block logic will also determine the mapping of which warp ID and position in the warp will each register file be responsible for.

Register Block to Register File Indexing Writing Design:

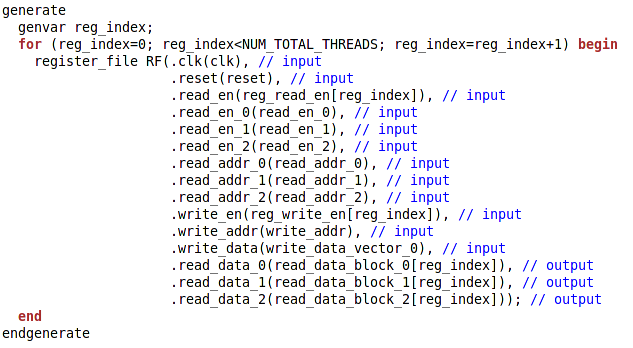


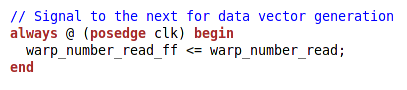


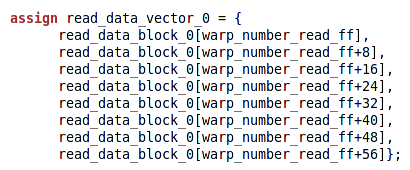


The logic snippet above show the writing algorithm. Based on the write\_en\_pred input signal (hardcoded to 8h’FF for the predicate register block and assigned to pred\_warp\_thread\_en\_vector\_e\_ff for the general purpose register block), the block\_write\_pred\_vector signal changes. Then, there will be a shift based on the warp\_number\_write signal.

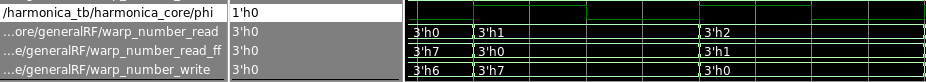
Register File to Register Block Reading Design:







Above are some relevant code snippets. The read data from each register file gets stored into a buffer. From that buffer, it will read the value corresponding to the warp number. It is important to not that the warp\_number\_read gets delayed one cycle. This is because when reading from the register file, there is a one cycle delay. Hence why there is an offset of warp\_number\_read\_ff.



*Figure: Screenshot of the warp number differences between read, read\_ff, and write for GPR*

|  |  |  |
| --- | --- | --- |
| **Register File I/O signals (Called from General Purpose and Predicate Register Block)** | | |
| **Signal Name and Direction** | **Width** | **Description** |
| clk (input) | [0] | clk signal |
| reset (input) | [0] | reset signal |
| read\_en (input) | [0] | top level read enable signal (master signal) |
| read\_en\_0 (input) | [0] | enable reading for the first entry |
| read\_en\_1 (input) | [0] | enable reading for the second entry |
| read\_en\_2 (input) | [0] | enable reading for the third entry |
| read\_addr\_0 (input) | [LOG2\_NUM\_REGS-1:0] | address of the first entry to be read |
| read\_addr\_1 (input) | [LOG2\_NUM\_REGS-1:0] | address of the second entry to be read |
| read\_addr\_2 (input) | [LOG2\_NUM\_REGS-1:0] | address of the third entry to be read |
| write\_en (input) | [0] | write enable to the register file |
| write\_addr (input) | [LOG2\_NUM\_REGS-1:0] | write address to the register file |
| write\_data (input) | [MACHINE\_WIDTH-1:0] | write data to the register file |
| read\_data\_0 (output) | [MACHINE\_WIDTH-1:0] | read data 1 from the register file |
| read\_data\_1 (output) | [MACHINE\_WIDTH-1:0] | read data 2 from the register file |
| read\_data\_2 (output | [MACHINE\_WIDTH-1:0] | read data 3 from the register file |

The register file is very generic. There are three read ports for the three maximum operands in the HARP ISA. The read happen i the positive edge of clock, while the write happens in the negative edge of clock. For implementing this on certain types of FPGA, it may be necessary to change the write edge from negative to positive.

**Section 5 – Execution Stage RTL Components Overview**

The Execution State contains all of the ALU units, data cache, jump logic, etc. The executionUnits.sv file generates multiple functional units (found in functionalUnit.sv). This is to create the number of lanes for the SIMD processor. Using the input ALU vectors, it is able to partition to the correct functional unit.

|  |  |  |
| --- | --- | --- |
| **Execution Units I/O Signals** | | |
| Signal Name and Direction | Width | Description |
| clk (input) | [0] | clk signal |
| reset (input) | [0] | reset signal |
| enable\_vector (input) | [NUM\_LANES-1:0] | enable vector that include enable bits for each lane |
| alu\_op (input) | [4:0] | alu operation signal |
| alu\_A\_vector (input) | [NUM\_LANES\*MACHINE\_WIDTH-1:0] | vector including the first input value |
| alu\_B\_vector (input) | [NUM\_LANES\*MACHINE\_WIDTH-1:0] | vector including the second input value |
| mem\_read (input) | [0] | enable for memory read |
| mem\_write (input) | [0] | enable for memory write |
| alu\_out\_vector (output) | [NUM\_LANES\*MACHINE\_WIDTH-1:0] | output vector from the ALU |

Execution Units module instantiates multiple functional units. The number of functional units is based on the number of lanes. The logic inside the file will correctly map the input vectors to the correct lane. The is done with a generate loop with the NUM\_LANES parameter. The generate variable lane\_index is then used to index the correct signals to each functional unit.





The code above show a snippet of how the alu vectors are partitioned based on the lane index. The table below is the IO signals table for the functional unit module from functionUnit.sv.

|  |  |  |
| --- | --- | --- |
| **Functional Units I/O Signals (Called from Execution Units)** | | |
| **Signal Name and Direction** | **Width** | **Description** |
| clk (input) | [0] | clk signal |
| reset (input) | [0] | reset signal |
| enable (input) | [0] | enable for the functional unit |
| alu\_op (input) | [4:0] | alu operation signal |
| alu\_A (input) | [MACHINE\_WIDTH-1:0] | first input signal |
| alu\_B(input) | [MACHINE\_WIDTH-1:0] | second input signal |
| mem\_read (input) | [0] | enable for memory read |
| mem\_write (input) | [0] | enable for memory write |
| alu\_out\_vector (output) | [MACHINE\_WIDTH-1:0] | output vector from the ALU |

The table below shows what functionality each alu\_op do:

|  |  |
| --- | --- |
| **ALU\_OP** | **Operation** |
| 5’b00000 | nop |
| 5’b00001 | neg |
| 5’b00010 | not |
| 5’b00011 | and |
| 5’b00100 | or |
| 5’b00101 | xor |
| 5’b00110 | add |
| 5’b00111 | sub |
| 5’b01000 | mul |
| 5’b01001 | div |
| 5’b01010 | mod |
| 5’b01011 | shl |
| 5’b01100 | shr |
| 5’b01101 | isneg |
| 5’b01110 | iszero |

**Section 6 – Connecting Everything Together**

To understand how everything gets connected together, do look at harmonica.sv. The beginning include the input and output signals of the Harmonica core. Some of the external signals are yet to be implemented. The next part of the code is internal signal declaration. The remaining of the code contain internal signal connections and flip flops. Below include short descriptions on some of the important microarchitecture:

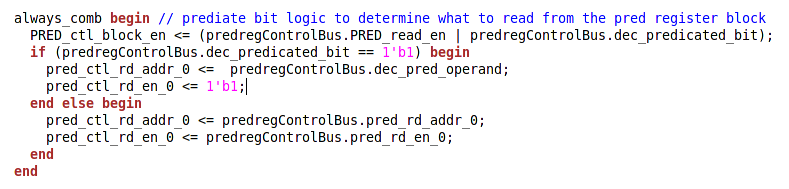
Predicated Instruction (@p ? ) Logic

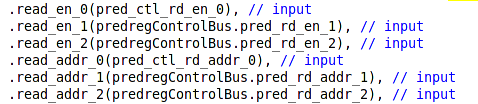
Enabling predicated instruction is different from other instructions because it enables reads from both the predicate and general register block.

Traversing through the pipeline, a predicated instruction will have the value of 1’b1 at its most significant bit. The decoder will set the dec\_predicated\_bit signal to 1’b1 and will also output the predicate register index to evaluate in the dec\_pred\_operand signal.

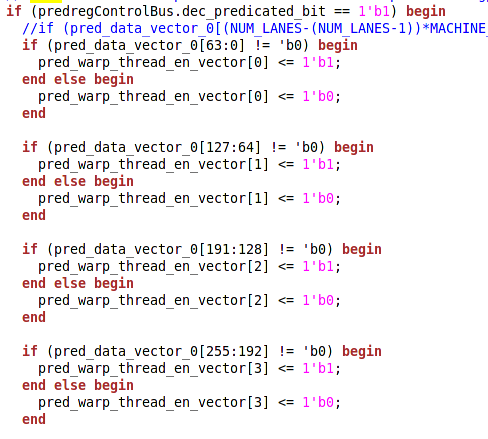


The decoder outputs get put into the fetchControlBus structure and will get flopped after each stage. When it reaches the predicate stage, it checks if it is a predicated instruction and set the read enable and address accordingly. If it is not a predicated instruction, but rather an instruction that alters the predicate registers, it will leave it as it is. The pred\_ctl\_rd\_addr\_0 and pred\_ctl\_rd\_en\_0 signals then get connected to the predicate register block.





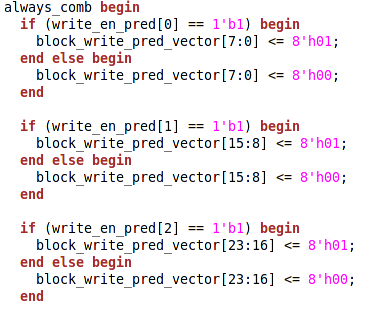
The predicated mask can now be used with the pred\_data\_vector output. If the partition of the pred\_data\_vector is not equal to 0, then the thread enable is true. Otherwise it is false. (Logic below found in harmonica.sv)

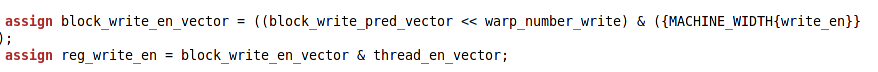


The pred\_warp\_thread\_en\_vector will then get flopped a number of cycles and will go into the write\_en\_pred port of the general purpose register block. (Logic below found in harmonica.sv)



With this signal, it generates the correct mapping and send it to the correct register. (Logic below found in register\_block.sv)

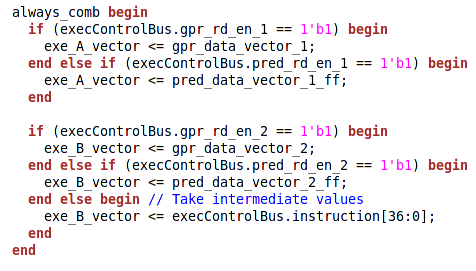






ExecutionUnits ALU Vector Logic

Inputs ALU\_A\_VECTOR and ALU\_B\_VECTOR are dependent on what type of registers the instruction require, and if it requires an intermediate value. The code snippet is shown below. (This is found in harmonica.sv)



**Section 7 – Gathering Instruction Hex Encodings form Harptool Tutorial**

Creating the hex encodings can be quite tedious. Luckily, Harptool, created by Chad, disassembles the assembly language to hex.

Instructions to get started with Harptool:

$ git clone<https://github.com/cdkersey/harptool>

$ cd harptool/src

# In order for the next step to work, you MUST have installed the "flex" package

# on your system, as well as g++

$ make -j 4

$ cd test

$ make run

$ cat simple.out

Edit any assembly file (\*.s file) and generate the binaries.

I personally saved an original binary file, changed the assembly file, rerun it, and diff the outputs. Something similar to:

make clean ; make run ; hexdump simple.bin | & tee simplehex.txt ; diff simplehex.txt simplehexori.txt

**Section 8 – Harmonica Tracker (Initial Debug Tool)**

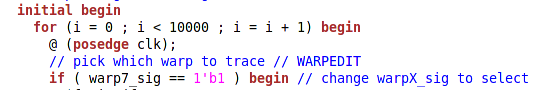
Harmonica has a lot of internal signals and registers that require to be checked for debugging purposes. A Harmonica tracker is undergoing development to provide information for the user to gauge RTL validity. This will be particularly useful for debugging multiple Register Files and warp states within each pipe stage.



*Figure: Sample output of the tracker feature. First parentheses value tells whatwarp the register that is changing is mapped to, while the second parentheses value tells what register in the register file that is being changed*

Despite the horrible code infrastructure (used a C++ program to generate repetitive statements and copied & pasted it to the tacker files; for some reason, genvar and the for statement did not work for me), it is obviously a good feature that will need additional work. The files are harmonica\_tracker.sv and harmonica\_tracker\_pred.sv. Harmonica\_tracker.sv tracks the general purpose registers while harmonica\_tracker\_pred.sv track the predicate register files.

The output file will be h\_tracker.txt and h\_tracker\_pred.txt. This tool is a framework for the user to change based on what they want to debug. For example, to analyze a different warp, the code snippet below need to be changed. The parameter to change is warpX\_sig. It currently support warp0\_sig, warp1\_sig…. to warp7\_sig. Any changes to the tracker will require additional copy & paste generation, so a better infrastructure is needed.

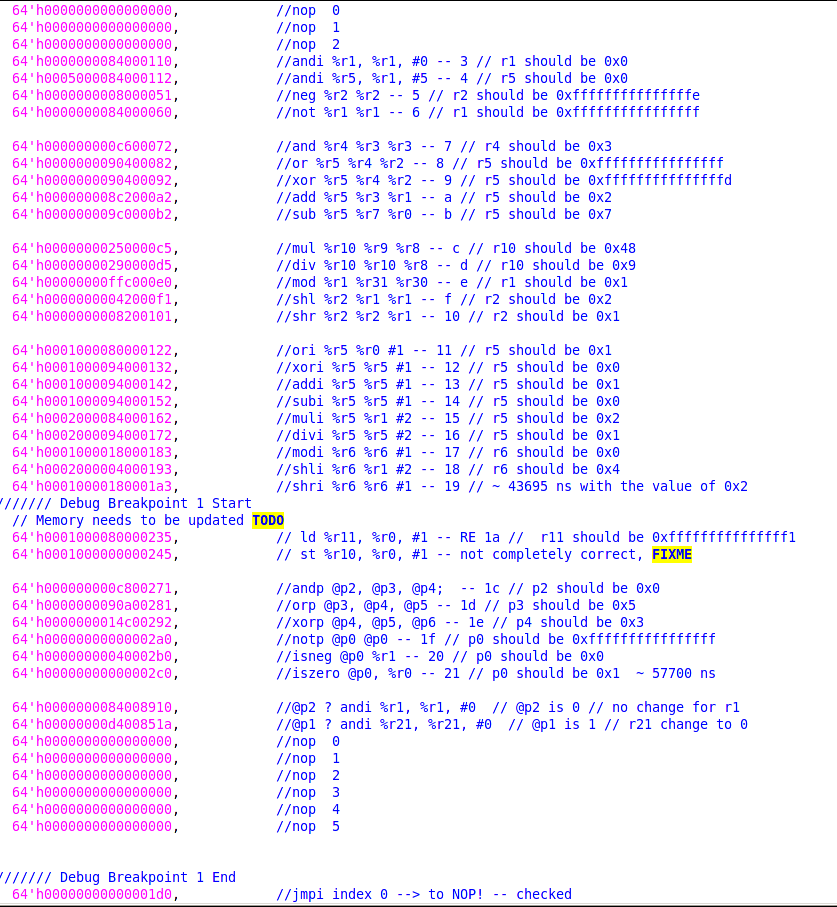


Potential Future Development

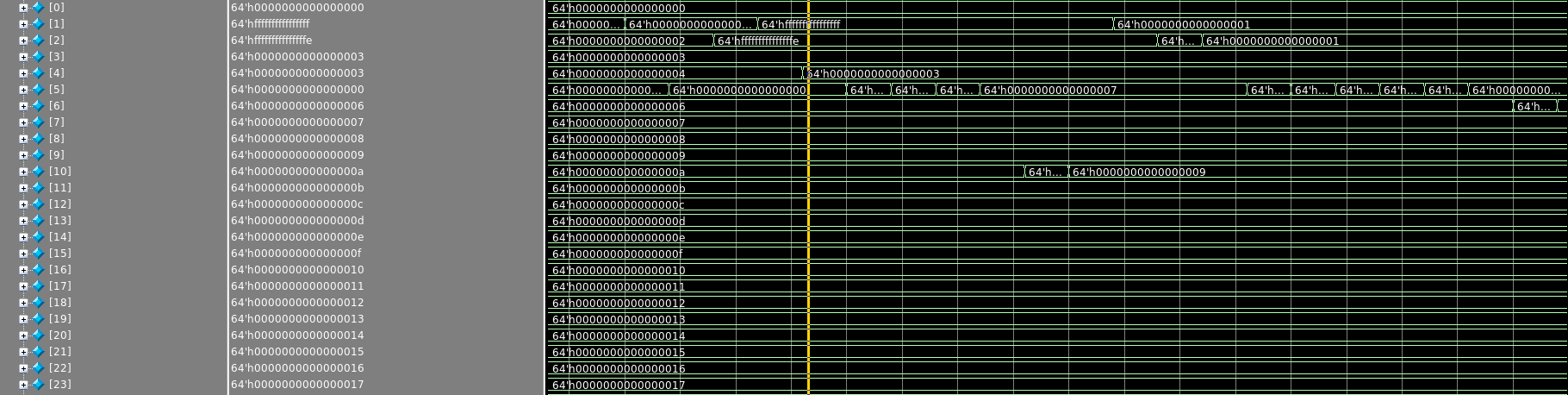
Besides the much needed infrastructure overhaul, there additional information that should be included to help users debug. One example is to include the assembly instruction of what as executed besides the timestamp. Also, including a checker and will make a more robust harmonica verification infrastructure. The checker will compare the expected outcome and the actual outcome and print out a TEST FAIL or TEST PASS to the user. If the test fail, it will highlight the timestamp of the register mismatch.

**Section 9 – Harmonica RTL 0.0 Verification Status**

The testbench below is found in the instruction cache portion of the harmonica\_tb.sv. Do look at the waveform and the tracker output files.



*Figure: Directed test cases for Harmonica RTL 0.0*



*Figure: Waveform of the register changings in one of the register files.*