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 Manual*" by Chad D. Kersey.

Pipeline Block Diagram of the Harmonica Core

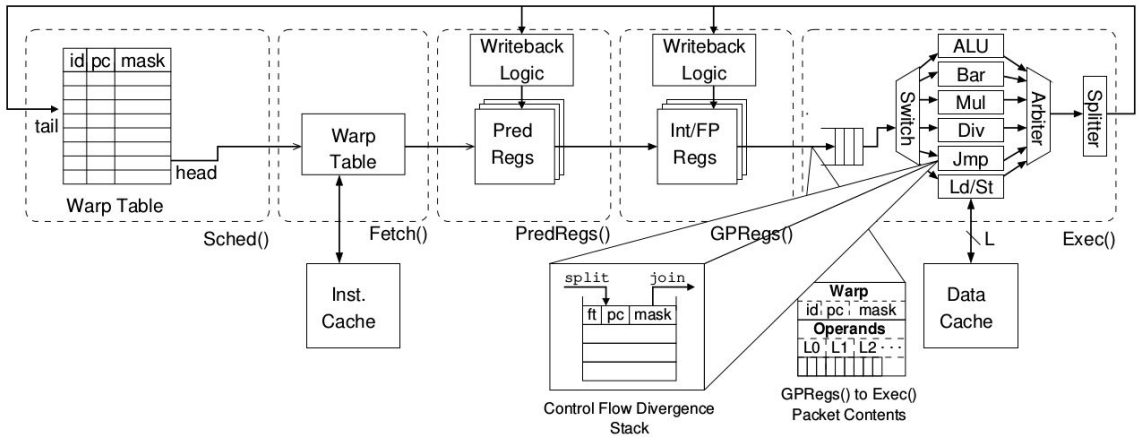

Figure 5: The Harmonica pipeline.

(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
Chad Kersey	Harmonica Architect	cdkersey@gatech.edu
Eric Qin	RTL 0.0 Architect	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 Settings
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)

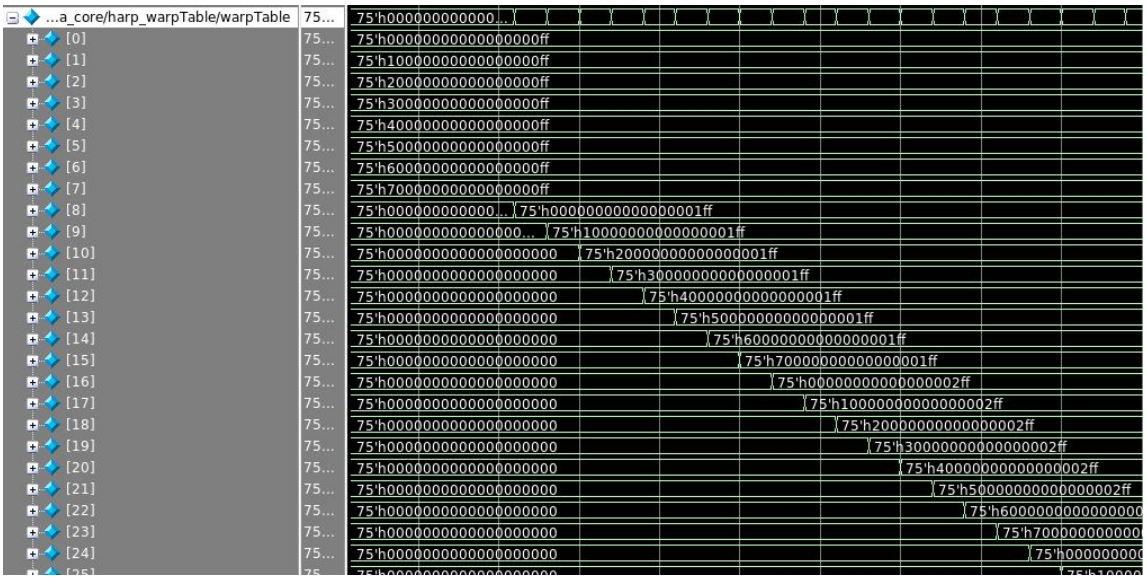

Figure of the initialized warp table (with the starter values) and generated warp table packets after each cycle.

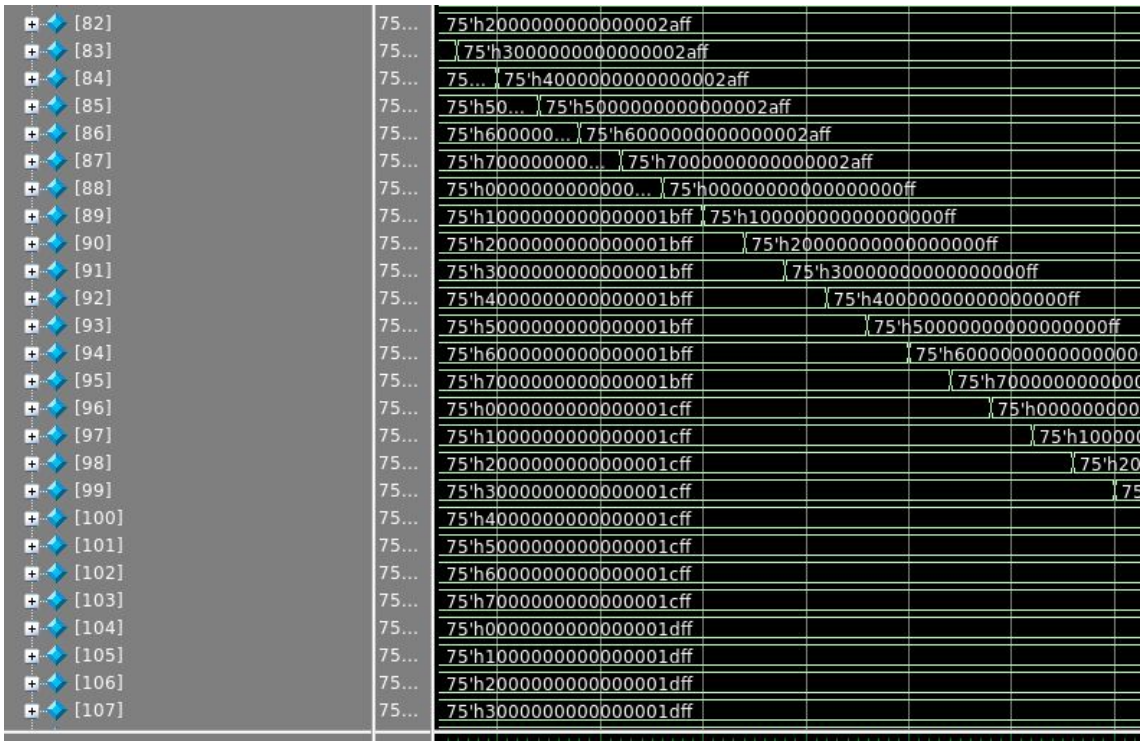

Figure of the warp table with a jump instruction. Notice how the PC value changes from 0x2a to 0.

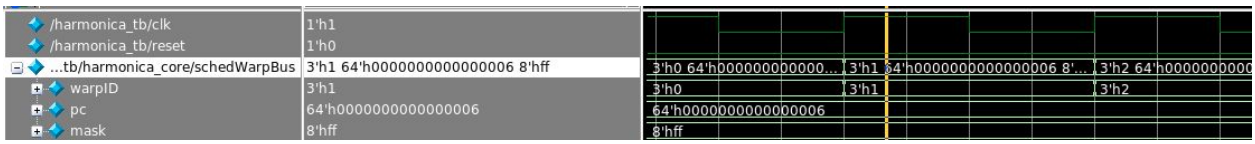

Figure: Assigning the warp table packet to the schedWarpBus structure.

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)

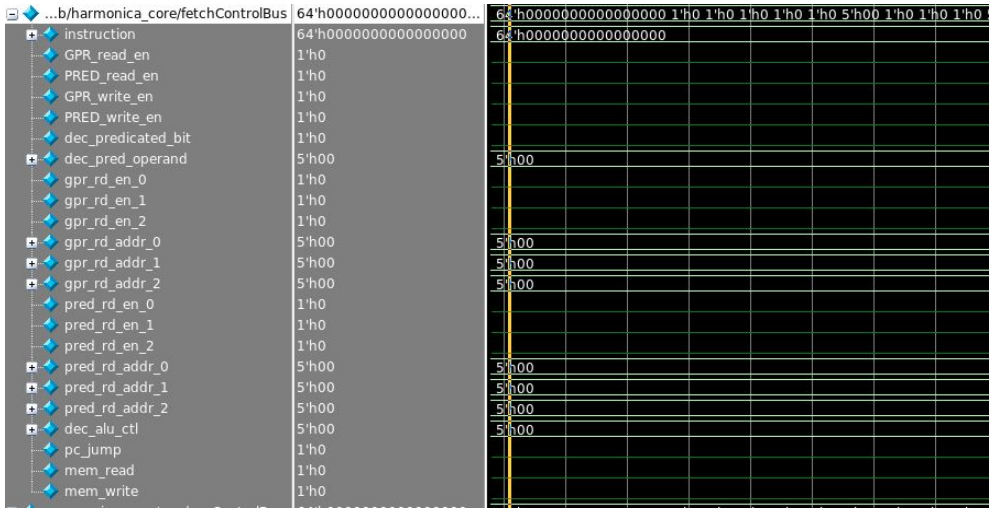

Figure: Decoder output for a nop instruction

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.

Argument Class	Description	Example
AC_NONE	No arguments	`di;`
AC_2REG	2 GPRs, 1 in, 1 out	`neg %r1, %r2;`
AC_2IMM	1 immediate in, 1 GPR out	`ldi %r1, #0xff;`
AC_3REG	3 GPRs, 2 in, 1 out	`add %r1, %r2, %r2;`
AC_3PREG	3 pred. regs, 2 in, 1 out	`andp @p0, @p0, @p1;`
AC_3IMM	GPR in, imm. in, GPR out	`andi %r1, %r3, #3;`
AC_3REGSRC	3 GPRs in	`tlbadd %r0, %r1, %r2;`
AC_1IMM	1 imm in	`jmpi label;`
AC_1REG	1 reg in	`jmpr %r2`
AC_3IMMSRC	2 GPRs in, 1 imm. in	`st %r1, %r2, #10;`
AC_PREG_REG	GPR in, pred. reg. out	`iszero @p0, %r3;`
AC_2PREG	2 pred. regs, 1 in, 1 out	`notp @p0, @p0;`

Figure from the HARP Instruction Set Manual: HARP Argument Classes

```
00  "nop"      NONE      01  "di"       NONE      02  "ei"       NONE
03  "tlbadd"   3REGSRC   04  "tlbflush" NONE      05  "neg"      2REG
06  "not"      2REG      07  "and"      3REG      08  "or"       3REG
09  "xor"      3REG      0a  "add"      3REG      0b  "sub"      3REG
0c  "mul"      3REG      0d  "div"      3REG      0e  "mod"      3REG
0f  "shl"      3REG      10  "shr"      3REG      11  "andi"     3IMM
12  "ori"      3IMM      13  "xori"     3IMM      14  "addi"     3IMM
15  "subi"     3IMM      16  "muli"     3IMM      17  "divi"     3IMM
18  "modi"     3IMM      19  "shli"     3IMM      1a  "shri"     3IMM
1b  "jali"     2IMM      1c  "jalr"     2REG      1d  "jmpi"     1IMM
1e  "jmpr"     1REG      1f  "clone"    1REG      20  "jalis"    3IMM
21  "jalrs"    3REG      22  "jmprt"    1REG      23  "ld"       3IMM
24  "st"       3IMMSRC   25  "ldi"      2IMM      26  "rtop"     PREG_REG
27  "andp"     3PREG     28  "orp"      3PREG     29  "xorp"     3PREG
2a  "notp"     2PREG     2b  "isneg"    PREG_REG  2c  "iszero"   PREG_REG
2d  "halt"     NONE      2e  "trap"     NONE      2f  "jmpru"    1REG
30  "skep"     1REG      31  "reti"     NONE      32  "tlbrm"    1REG
33  "itof"     2REG      34  "ftoi"     2REG      35  "fadd"     3REG
36  "fsub"     3REG      37  "fmul"     3REG      38  "fdiv"     3REG
39  "fneg"     2REG      3a  "wspawn"   3REG      3b  "split"    NONE
3c  "join"     NONE      3d  "bar"
```

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.

pseudocode "if (%r1) %r2++ else %r2--;" becomes:

```
        rtop @p0, %r1     // reg. to predicate
@p0 ? addi %r2, %r2, #1 // if @p0, add
        notp @p0, @p0     // complement @p0
@p0 ? subi %r2, %r2, #1 // if @p0, subtract
```

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.

```
        rtop @p0, %r1
@p0 ? split
@p0 ? jmp then
        subi %r2, %r2, #1
        jmp cont
else:   addi %r2, %r2, #1
cont:   join
```

when `split` and `join` are used.

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:

```
always_comb begin
  if (write_en_pred[0] == 1'b1) begin
    block_write_pred_vector[7:0] <= 8'h01;
  end else begin
    block_write_pred_vector[7:0] <= 8'h00;
  end

  if (write_en_pred[1] == 1'b1) begin
    block_write_pred_vector[15:8] <= 8'h01;
  end else begin
    block_write_pred_vector[15:8] <= 8'h00;
  end
```

```
assign reg_write_en = block_write_en_vector & thread_en_vector;
```

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:

```
generate
  genvar reg_index;
  for (reg_index=0; reg_index<NUM_TOTAL_THREADS; reg_index=reg_index+1) begin
    register_file RF(.clk(clk), // input
                     .reset(reset), // input
                     .read_en(reg_read_en[reg_index]), // input
                     .read_en_0(read_en_0), // input
                     .read_en_1(read_en_1), // input
                     .read_en_2(read_en_2), // input
                     .read_addr_0(read_addr_0), // input
                     .read_addr_1(read_addr_1), // input
                     .read_addr_2(read_addr_2), // input
                     .write_en(reg_write_en[reg_index]), // input
                     .write_addr(write_addr), // input
                     .write_data(write_data_vector_0), // input
                     .read_data_0(read_data_block_0[reg_index]), // output
                     .read_data_1(read_data_block_1[reg_index]), // output
                     .read_data_2(read_data_block_2[reg_index])); // output
  end
endgenerate

 // Signal to the next for data vector generation
 always @ (posedge clk) begin
   warp_number_read_ff <= warp_number_read;
 end


 assign read_data_vector_0 = {
       read_data_block_0[warp_number_read_ff],
       read_data_block_0[warp_number_read_ff+8],
       read_data_block_0[warp_number_read_ff+16],
       read_data_block_0[warp_number_read_ff+24],
       read_data_block_0[warp_number_read_ff+32],
       read_data_block_0[warp_number_read_ff+40],
       read_data_block_0[warp_number_read_ff+48],
       read_data_block_0[warp_number_read_ff+56]};
```

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.

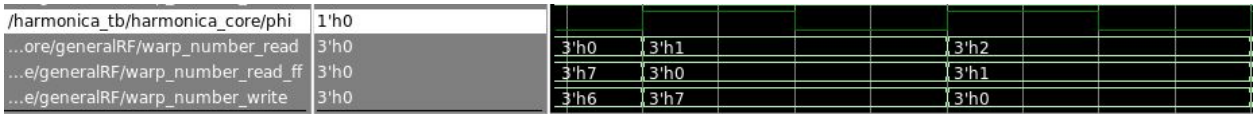

Figure: Screenshot of the warp number differences between read, read_ff, and write for GPR

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.

```
.alu_out(alu_out_vector[lane_index*MACHINE_WIDTH+MACHINE_WIDTH-1:lane_index*MACHINE_WIDTH]));
```

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.

```
.dec_predicated_bit(dec_predicated_bit), // output
.dec_pred_operand(dec_pred_operand), // output
```

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.

```
always_comb begin // prediate bit logic to determine what to read from the pred register block
  PRED_ctl_block_en <= (predregControlBus.PRED_read_en | predregControlBus.dec_predicated_bit);
  if (predregControlBus.dec_predicated_bit == 1'b1) begin
    pred_ctl_rd_addr_0 <=  predregControlBus.dec_pred_operand;
    pred_ctl_rd_en_0 <= 1'b1;
  end else begin
    pred_ctl_rd_addr_0 <= predregControlBus.pred_rd_addr_0;
    pred_ctl_rd_en_0 <= predregControlBus.pred_rd_en_0;
  end
end

.read_en_0(pred_ctl_rd_en_0), // input
.read_en_1(predregControlBus.pred_rd_en_1), // input
.read_en_2(predregControlBus.pred_rd_en_2), // input
.read_addr_0(pred_ctl_rd_addr_0), // input
.read_addr_1(predregControlBus.pred_rd_addr_1), // input
.read_addr_2(predregControlBus.pred_rd_addr_2), // input
```

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)

```
if (predregControlBus.dec_predicated_bit == 1'b1) begin
  //if (pred_data_vector_0[(NUM_LANES-(NUM_LANES-1))*MACHINE
  if (pred_data_vector_0[63:0] != 'b0) begin
    pred_warp_thread_en_vector[0] <= 1'b1;
  end else begin
    pred_warp_thread_en_vector[0] <= 1'b0;
  end

  if (pred_data_vector_0[127:64] != 'b0) begin
    pred_warp_thread_en_vector[1] <= 1'b1;
  end else begin
    pred_warp_thread_en_vector[1] <= 1'b0;
  end

  if (pred_data_vector_0[191:128] != 'b0) begin
    pred_warp_thread_en_vector[2] <= 1'b1;
  end else begin
    pred_warp_thread_en_vector[2] <= 1'b0;
  end

  if (pred_data_vector_0[255:192] != 'b0) begin
    pred_warp_thread_en_vector[3] <= 1'b1;
  end else begin
    pred_warp_thread_en_vector[3] <= 1'b0;
  end
```

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)

```
.write_en_pred(pred_warp_thread_en_vector_e_ff),
```

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

```
always_comb begin
  if (write_en_pred[0] == 1'b1) begin
    block_write_pred_vector[7:0] <= 8'h01;
  end else begin
    block_write_pred_vector[7:0] <= 8'h00;
  end

  if (write_en_pred[1] == 1'b1) begin
    block_write_pred_vector[15:8] <= 8'h01;
  end else begin
    block_write_pred_vector[15:8] <= 8'h00;
  end

  if (write_en_pred[2] == 1'b1) begin
    block_write_pred_vector[23:16] <= 8'h01;
  end else begin
    block_write_pred_vector[23:16] <= 8'h00;
  end


assign block_write_en_vector = ((block_write_pred_vector << warp_number_write) & ({MACHINE_WIDTH{write_en}}
);
assign reg_write_en = block_write_en_vector & thread_en_vector;
```

```
.write_en(reg_write_en[reg_index]),
```

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)

```
always_comb begin
  if (execControlBus.gpr_rd_en_1 == 1'b1) begin
    exe_A_vector <= gpr_data_vector_1;
  end else if (execControlBus.pred_rd_en_1 == 1'b1) begin
    exe_A_vector <= pred_data_vector_1_ff;
  end

  if (execControlBus.gpr_rd_en_2 == 1'b1) begin
    exe_B_vector <= gpr_data_vector_2;
  end else if (execControlBus.pred_rd_en_2 == 1'b1) begin
    exe_B_vector <= pred_data_vector_2_ff;
  end else begin // Take intermediate values
    exe_B_vector <= execControlBus.instruction[36:0];
  end
end
```

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.

```
====================================================
Simulation Time          --->               14500

Register (0,4) Value: 0000000000000003
Register (8,4) Value: 0000000000000003
Register (16,4) Value: 0000000000000003
Register (24,4) Value: 0000000000000003
Register (32,4) Value: 0000000000000003
Register (40,4) Value: 0000000000000003
Register (48,4) Value: 0000000000000003
Register (56,4) Value: 0000000000000003

====================================================




====================================================
Simulation Time          --->               16100

Register (0,5) Value: ffffffffffffffff
Register (8,5) Value: ffffffffffffffff
Register (16,5) Value: ffffffffffffffff
Register (24,5) Value: ffffffffffffffff
Register (32,5) Value: ffffffffffffffff
Register (40,5) Value: ffffffffffffffff
Register (48,5) Value: ffffffffffffffff
Register (56,5) Value: ffffffffffffffff

====================================================
```

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.

```
initial begin
  for (i = 0 ; i < 10000 ; i = i + 1) begin
    @ (posedge clk);
    // pick which warp to trace // WARPEDIT
    if ( warp7_sig == 1'b1 ) begin // change warpX_sig to select
```

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.

```
  64'h0000000000000000,             //nop  0
  64'h0000000000000000,             //nop  1
  64'h0000000000000000,             //nop  2
  64'h0000000084000110,             //andi %r1, %r1, #0 -- 3 // r1 should be 0x0
  64'h0005000084000112,             //andi %r5, %r1, #5 -- 4 // r5 should be 0x0
  64'h0000000008000051,             //neg %r2 %r2 -- 5 // r2 should be 0xfffffffffffffffe
  64'h0000000084000060,             //not %r1 %r1 -- 6 // r1 should be 0xffffffffffffffff

  64'h000000000c600072,             //and %r4 %r3 %r3 -- 7 // r4 should be 0x3
  64'h0000000090400082,             //or %r5 %r4 %r2 -- 8 // r5 should be 0xffffffffffffffff
  64'h0000000090400092,             //xor %r5 %r4 %r2 -- 9 // r5 should be 0xfffffffffffffffd
  64'h000000008c2000a2,             //add %r5 %r3 %r1 -- a // r5 should be 0x2
  64'h000000009c0000b2,             //sub %r5 %r7 %r0 -- b // r5 should be 0x7

  64'h00000000250000c5,             //mul %r10 %r9 %r8 -- c // r10 should be 0x48
  64'h00000000290000d5,             //div %r10 %r10 %r8 -- d // r10 should be 0x9
  64'h00000000ffc000e0,             //mod %r1 %r31 %r30 -- e // r1 should be 0x1
  64'h00000000042000f1,             //shl %r2 %r1 %r1 -- f // r2 should be 0x2
  64'h0000000008200101,             //shr %r2 %r2 %r1 -- 10 // r2 should be 0x1

  64'h0001000080000122,             //ori %r5 %r0 #1 -- 11 // r5 should be 0x1
  64'h0001000094000132,             //xori %r5 %r5 #1 -- 12 // r5 should be 0x0
  64'h0001000094000142,             //addi %r5 %r5 #1 -- 13 // r5 should be 0x1
  64'h0001000094000152,             //subi %r5 %r5 #1 -- 14 // r5 should be 0x0
  64'h0002000084000162,             //muli %r5 %r1 #2 -- 15 // r5 should be 0x2
  64'h0002000094000172,             //divi %r5 %r5 #2 -- 16 // r5 should be 0x1
  64'h0001000018000183,             //modi %r6 %r6 #1 -- 17 // r6 should be 0x0
  64'h0002000004000193,             //shli %r6 %r1 #2 -- 18 // r6 should be 0x4
  64'h00010000180001a3,             //shri %r6 %r6 #1 -- 19 // ~ 43695 ns with the value of 0x2
'////// Debug Breakpoint 1 Start
  // Memory needs to be updated TODO
  64'h0001000080000235,             // ld %r11, %r0, #1 -- RE 1a //  r11 should be 0xfffffffffffffff1
  64'h0001000000000245,             // st %r10, %r0, #1 -- not completely correct, FIXME

  64'h000000000c800271,             //andp @p2, @p3, @p4;  -- 1c // p2 should be 0x0
  64'h0000000090a00281,             //orp @p3, @p4, @p5 -- 1d // p3 should be 0x5
  64'h0000000014c00292,             //xorp @p4, @p5, @p6 -- 1e // p4 should be 0x3
  64'h00000000000002a0,             //notp @p0 @p0 -- 1f // p0 should be 0xffffffffffffffff
  64'h00000000040002b0,             //isneg @p0 %r1 -- 20 // p0 should be 0x0
  64'h00000000000002c0,             //iszero @p0, %r0 -- 21 // p0 should be 0x1  ~ 57700 ns

  64'h0000000084008910,             //@p2 ? andi %r1, %r1, #0  // @p2 is 0 // no change for r1
  64'h00000000d400851a,             //@p1 ? andi %r21, %r21, #0  // @p1 is 1 // r21 change to 0
  64'h0000000000000000,             //nop  0
  64'h0000000000000000,             //nop  1
  64'h0000000000000000,             //nop  2
  64'h0000000000000000,             //nop  3
  64'h0000000000000000,             //nop  4
  64'h0000000000000000,             //nop  5


'////// Debug Breakpoint 1 End
  64'h00000000000001d0,             //jmpi index 0 --> to NOP! -- checked
```

Figure: Directed test cases for Harmonica RTL 0.0

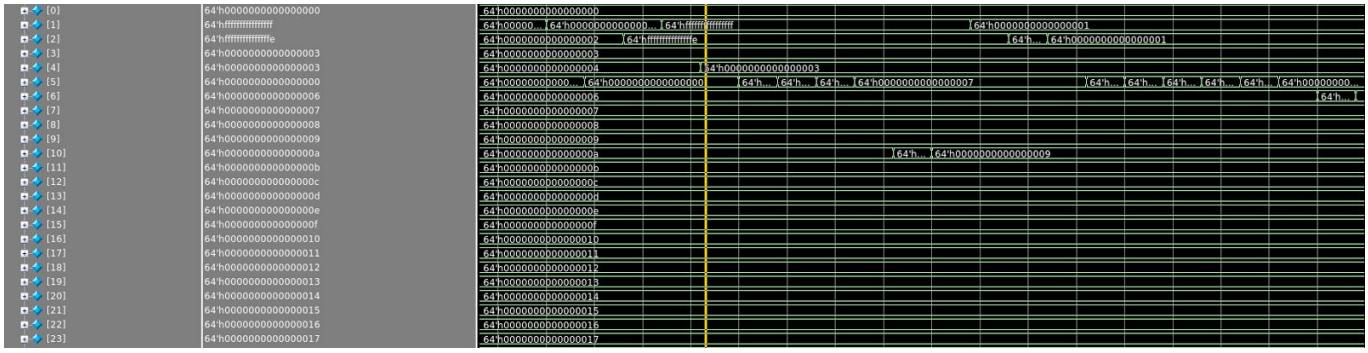

Figure: Waveform of the register changings in one of the register files.