CSEE 4840: Embedded System Design
Project Report

Martha Barker (mmb2295)

May 14, 2019

Contents

1 Introduction

The aim of this project is to implement a test framework for a garbage collected memory system I designed as part of work done by Martha Kim's and Stephen Edwards' groups at Columbia University on a Haskell to Hardware Compiler [1]. The compiler simplifies custom accelerator design by allowing designers to write functions Haskell and compile them into SystemVerilog. Like the original Haskell applications, the generated circuits have an immutable memory model. While this enables a lot of parallelism, it also means the circuits run through memory very quickly. Automatic garbage collection recycles memory locations that can no longer be accessed to better utilize memory resources.

The hardware component is the garbage collected memory system. It is implemented in an asynchronous dataflow language which is compiled to SystemVerilog. The software component generates random inputs according to user defined parameters and validates the hardware against a software reference program.

2 Design

Figure 1 shows a high level overview of the system architecture.

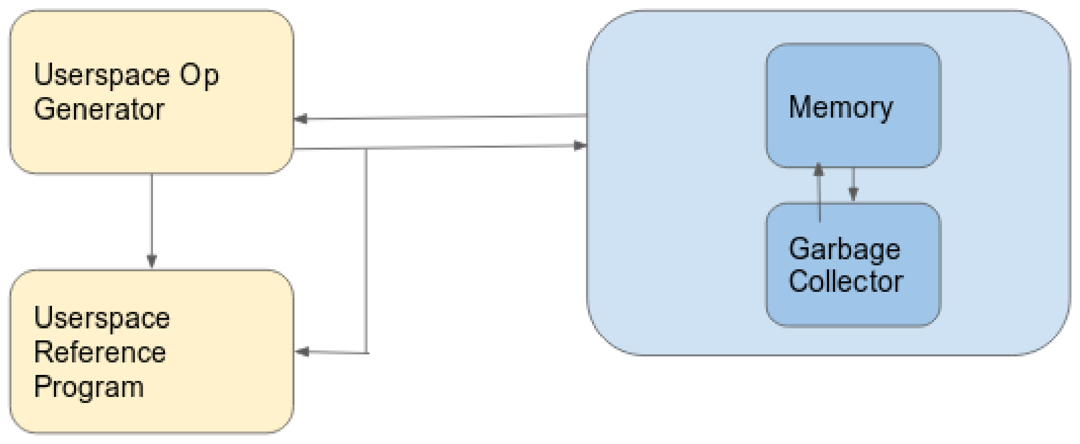


Figure 1: System Architecture

The userspace op generator accepts user defined parameters to specify the memory space then initializes the memory. It then generates random inputs for the hardware. The userspace program talks to the FPGA over the avalon bus. The device driver specifies a 4 bit address and 32 bit data line which is written to or read from the hardware. The hardware module accepts reads and writes produces results. The userspace software polls a status register to know when the hardware is ready to accept inputs or has valid results. After results are read back into the software, they are validated against a software reference to ensure correct operation.

3 Interface

3.1 Avalon Bus

The hardware and software communicate over the avalon bus using the memory mapped peripheral model [Figure 2].

A memory mapped interface implements read and write between master and slave components. The master component drives address, chipselect, and writedata/readdata signals to the slave component. The

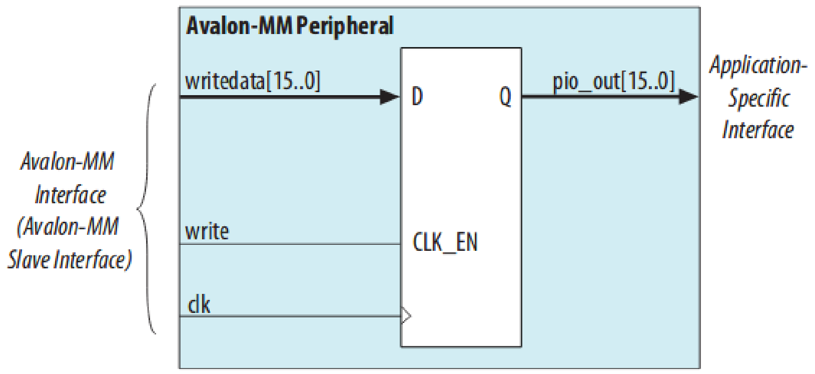

Avalon-MM Peripheral
writedata[15..0]
D
Q
pio_out[15..0]
Application-Specific Interface
Avalon-MM Interface (Avalon-MM Slave Interface)
write
CLK_EN
clk

Figure 2: Basic Slave Component

slave component either reads from or writes to writedata/readdata and uses the address to know which register contains data.

On a read transfer to a slave component, the address is asserted and chipselect goes high [Figure 3]. The readdata is then latched one cycle later to account for the time it takes the slave device to handle the read request.

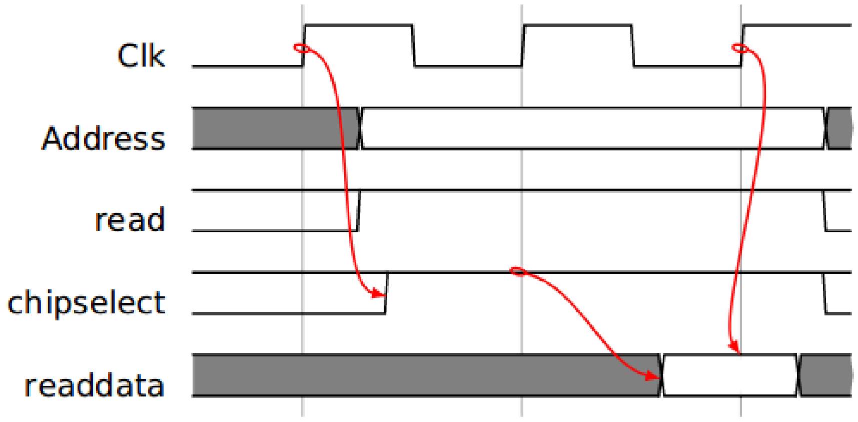


Figure 3: Slave Read Transfer

On a write transfer to a slave component, the address and chipselect are asserted and the slave component latches in the writedata signal in the same clock cycle [Figure 4].

3.2 Implementation

In my design, the hardware module is a slave component connected to the hard processor system using qsys. The interface includes both writedata and readdata signals. I used a 4 bit address to express 8 different registers [Table 1]. The status register shows the status of the hardware module. Num roots, base, bound, and gc threshold are used during memory initialization. The read and write addresses are used both to write operations to the hardware and read the results of those operations back into software. The root address writes a root to the hardware component during garbage collection. The bram address is used to read out the contents of the bram for validation.

My original plan was to implement interrupts so the hardware module would interrupt the hard processor system when the status register changed, but I had difficulty registering the interrupt in my device driver so I changed to a polling implementation.

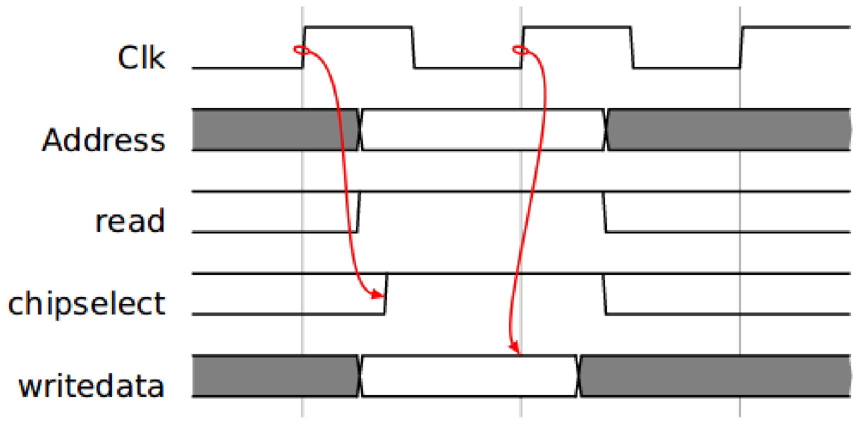


Figure 4: Slave Write Transfer

Address	Function
0	status
1	num roots
2	base
3	bound
4	gc threshold
5	read op/result
6	write op/result
7	root op
8	bram

Table 1: Interface Addresses

4 Hardware

4.1 Hardware Design

The hardware module is an automatic, in-hardware garbage collector for accelerators to decrease thememory footprint of accelerators generated from high level languages. It implements a coat-check style of memory interface where a node is written to memory, and the memory returns the pointer where it can be accessed. An overview of the hardware design is in Figure 5. The garbage collector implements a pointer chasing algorithm where first pointers are recursively marked from a set of external roots, then the entire address space is swept to free all unmarked pointers.

The hardware module was designed as an asynchronous dataflow network. This means that instead of values on wires, we use tokens in buffers. The dataflow network consists of actors connected via point-to-point fifo channels. An actor "fires" and tokens move through the system when the upstream actor asserts that its data is valid and the downstream actor asserts that it is ready for input [Figure 6]. When an actor fires, it consumes one or more tokens from its input channels and produces tokens on zero or more of its output channels.

4.2 Status register

The hardware communicates with the software component over the avalon bus, and uses a status register to communicate its status [Table 2]. The status register is reset every time it is read. Each bit is a one hot indication of a particular status. Bits 0 and 1 indicate that the hardware is ready for a new read op or write op to be written. Bits 2 and 3 indicate that the hardware has a valid read or write response to be read. Bit 4 indicates that the memory needs to be garbage collected and roots should be the next input. Bit

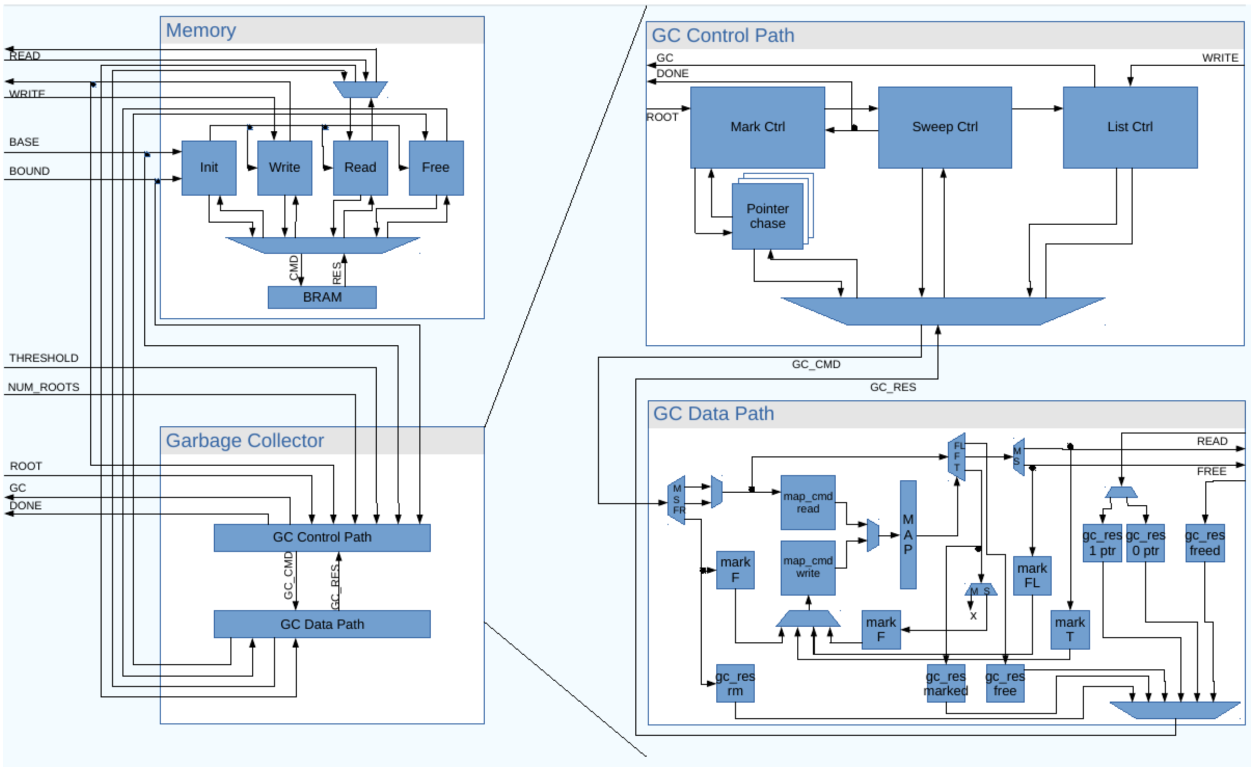
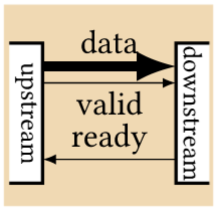


Figure 5: Garbage Collected Memory



valid	ready	Meaning
1	0	Token valid; not consumed
1	1	Token transferred
0	–	No token to transfer

Figure 6: Handshake Protocol

5 indicates that garbage collection is complete and normal operation can resume. Bit 6 indicates that the memory is ready for a new root input.

4.3 Interface Interaction

I use the existing ready and valid signals from the handshake protocol to implement the status register. I use a register for the memory inputs to hold the next input until the memory is ready to accept it. When the token is consumed by the memory, I then update the status register to get a new input from the software. When an output is valid, a register holds the valid result until it is read by the software and that register is cleared for the next valid result.

The access the bram for validation, I keep a counter. Every time read is asserted with the bram address, the hardware returns the bram entry at the counter address and increments the counter. When the counter reaches the maximum address it resets to the minimum address. Because reads over the avalon bus have a one cycle delay, I only increment the counter every other clock cycle where read and the bram address are asserted.

Bit	Meaning
0	ready for read op
1	ready for write op
2	read response valid
3	write response valid
4	start gc
5	gc complete
6	ready for root

Table 2: Status Register

4.4 Limitations

A limitation of my hardware module is that it needs to be rebooted between every run for correctness. To do a reset without reboot I would have had to clear every token out of the system. Most of the SystemVerilog code in my hardware module is generated from a dataflow language which is difficult to change in such a manner.

5 Software

5.1 GC Memory Driver

The device driver writes values to 8 device registers which correspond to the 8 addresses in hardware. I write 32 bit wide data, so the addresses are multiplied by 4 because registers are byte addressed. Userspace programs access the driver fuctions using ioctl(). On writes the driver copies data from userspace to kernel space and uses iowrite32() to write data. On reads the driver uses ioread32() to read data then copies the data from kernel space to user space on completion.

5.2 Op Generation

The main body of the op generation code polls the status register to update its status. If the status indicates that it is time for garbage collection, the software switches to garbage collection mode. In garbage collection mode, no new inputs can be written. Instead, the code waits for all outstanding ops to complete so that it can begin garbage collection. If there are no outstanding ops, then the code begins garbage collection by writing roots as the hardware module is ready for them. It then waits until the status register indicates garbage collection is complete before switching back to regular mode.

In regular mode, the software checks the status of read op, write op, read res, and write res, and writes ops and reads results accordingly. The number of ops types of ops depend on the user inputs or default to 20,000 ops where 20% are writes and 80% are reads. Write ops can only include pointers to existing roots and read ops can only read existing roots.

Because in actual accelerators pointers are not held on to continually, I have implemented a lifespan for pointers in the user space program. Each new pointer is assigned a random lifespan between 2 and 102 which is decremented every time a write op occurs. This allows memory to be freed during garbage collection because it is no longer a root or reachable from a root.

5.3 Validation

I implement a software reference to validate the hardware results against. The reference contains a bram model, a list of roots, and the freelist head and tail. When writing an op to hardware, I execute the same op on the software model. When a result is read from hardware I check it against the model. If the results are ever different, the program prints the error and stops.

I also have a function to garbage collect the model using the same algorithm as the hardware. At the time of garbage collection, the main program reads out the hardware bram and validates it against the model. It then does garbage collection on both the hardware and the softwre model. Finally, it reads out the hardware bram and validates it again.

Correct operation is very anti-climactic because almost nothing appears to have happened. If an error occurs, the error is printed and the program stops. All of the randomness used in the program is seeded, so every situation is reproducible for debugging as long as the seed does not change.

6 Conclusion

In the project, I implemented a hardware/software system to test a memory system with in-hardware garbage collection. I had previously tested the system in simulation to validate every memory output. I also tested the system using real application circuits and validated the circuit output, rather than the memory outputs. I am pleased that the hardware/software system confirms that the hardware module is working correctly.

7 References

[1] Richard Townsend and Martha A. Kim and Stephen A. Edwards. From Functional Programs to Pipelined Dataflow Circuits. In Proceedings of Compiler Construction (CC), pages 76-86, Austin, Texas, February 2017.

8 File Listings

8.1 Hardware

- GC_Memory.sv

8.2 Software

- GC_Memory.c
- GC_Memory.h
- main.c
- hardware.c
- hardware.h
- reference.c
- reference.h

A GC_Memory.sv

```
module GC_Memory(input logic clk,
                 input logic         reset,
                 input logic [31:0]  writedata,
                 input logic         write,
                 input               chipselect,
                 input logic [3:0]   address,
                 output logic [31:0] readdata,
                 output logic [9:0]  LEDR
                 );

   logic [11:0]                      set_num_roots, set_base, set_bound,
      set_threshold;
   logic [6:0]                       interrupt_status;
   logic [11:0]                      read_op, write_res;
   logic [24:0]                      write_op, read_res;
   logic [12:0]                      root_op;
   logic [11:0]                      bram_addr;
   logic [24:0]                      bram_line;

   assign irq = |interrupt_status;

   initial begin
      LEDR = 10'b0000000000;
   end

   always_ff @(posedge clk) begin
      if (chipselect) begin
         case (address)
           4'h0 : LEDR[3:0] <= 4'h0 ;
           4'h1 : LEDR[3:0] <= 4'h1 ;
           4'h2 : LEDR[3:0] <= 4'h2 ;
           4'h3 : LEDR[3:0] <= 4'h3 ;
           4'h4 : LEDR[3:0] <= 4'h4 ;
           4'h5 : LEDR[3:0] <= 4'h5 ;
           4'h6 : LEDR[3:0] <= 4'h6 ;
           4'h7 : LEDR[3:0] <= 4'h7 ;
           4'h8 : LEDR[3:0] <= 4'h8 ;
           4'h9 : LEDR[3:0] <= 4'h9 ;
           4'ha : LEDR[3:0] <= 4'ha ;
           4'hb : LEDR[3:0] <= 4'hb ;
           4'hc : LEDR[3:0] <= 4'hc ;
           4'hd : LEDR[3:0] <= 4'hd ;
           4'he : LEDR[3:0] <= 4'he ;
           4'hf : LEDR[3:0] <= 4'hf ;
         endcase
      end
   end // always_ff @

   always_ff @(posedge clk) begin
      LEDR[8:4] <= bram_addr;
   end
```

```
logic increment;

always_ff @(posedge clk) begin
   if (reset) begin
      set_num_roots <= 12'd0;
      set_base <= 12'd0;
      set_bound <= 12'd0;
      set_threshold <= 12'd0;
      read_op <= 12'd0;
      write_op <= 25'd0;
      root_op <= 13'd0;
   end else if (chipselect) begin
      case (address)
        4'h0 : if (!write) begin
           readdata[6:0] <= interrupt_status;
        end
        4'h1 : if (write) set_num_roots <= writedata[11:0];
        4'h2 : if (write) set_base <= writedata[11:0];
        4'h3 : if (write) set_bound <= writedata[11:0];
        4'h4 : if (write) set_threshold <= writedata[11:0];
        4'h5 : if (write) read_op <= writedata[11:0]; else if (!write)
            readdata <= {7'b0, read_res};
        4'h6 : if (write) write_op <= writedata[24:0]; else if (!write)
            readdata <= {20'd0, write_res};
        4'h7 : if (write) root_op <= writedata[12:0];
        4'h8 : if (!write) readdata <= {7'b0, bram_line};
        default : ;
      endcase
   end // if (chipselect)

   if (chipselect && write && (address == 4'h2))
     bram_addr <= writedata[11:0];
   else if (chipselect && !write && (address == 4'h8)) begin
      if (increment) begin
         bram_addr <= bram_addr + 12'd1;
         increment <= ~increment;
      end else begin
         increment <= ~increment;
      end
   end
   else if (bram_addr >= set_bound)
     bram_addr <= set_base;
end

logic num_roots_rdy, base_rdy, bound_rdy, threshold_rdy;
initial begin
   num_roots_rdy = 1'b0;
   base_rdy = 1'b0;
   bound_rdy = 1'b0;
   threshold_rdy = 1'b0;
end

always_ff @(posedge clk) begin
   if (chipselect && write && (address == 4'h1)) begin
```

```
      num_roots_rdy <= 1'b1;
   end else if (num_roots_r && num_roots_rdy) begin
      num_roots_rdy <= 1'b0;
   end

   if (chipselect && write && (address == 4'h2)) begin
      base_rdy <= 1'b1;
   end else if (base_r && base_rdy) begin
      base_rdy <= 1'b0;
   end

   if (chipselect && write && (address == 4'h3)) begin
      bound_rdy <= 1'b1;
   end else if (bound_r && bound_rdy) begin
      bound_rdy <= 1'b0;
   end

   if (chipselect && write && (address == 4'h4)) begin
      threshold_rdy <= 1'b1;
   end else if (threshold_r && threshold_rdy) begin
      threshold_rdy <= 1'b0;
   end
end

assign num_roots_d = {set_num_roots, 1'b1} & {13{num_roots_rdy}};
assign base_d = {set_base, 1'b1} & {13{base_rdy}};
assign bound_d = {set_bound, 1'b1} & {13{bound_rdy}};
assign threshold_d = {set_threshold, 1'b1} & {13{threshold_rdy}};

logic write_rdy, read_rdy, write_res_rdy, read_res_rdy, root_rdy;
initial begin
   write_rdy = 1'b0;
   read_rdy = 1'b0;
   write_res_rdy = 1'b0;
   read_res_rdy = 1'b0;
   root_rdy = 1'b0;
end

always_ff @(posedge clk) begin
   if (chipselect && write && (address == 4'h6)) begin
      write_rdy <= 1'b1;
   end else if (wr_r && write_rdy) begin
      write_rdy <= 1'b0;
      interrupt_status[1] <= 1'b1;
   end else if (chipselect && !write && (address == 4'h0)) begin
      interrupt_status[1] <= 1'b0;
   end

   if (chipselect && write && (address == 4'h5)) begin
      read_rdy <= 1'b1;
   end else if (rd_r && read_rdy) begin
      read_rdy <= 1'b0;
      interrupt_status[0] <= 1'b1;
   end else if (chipselect && !write && (address == 4'h0)) begin
```

```
            interrupt_status[0] <= 1'b0;
        end

        if (~write_res_rdy && wr_ptr_dout[0]) begin
            write_res_rdy <= 1'b1;
            interrupt_status[3] <= 1'b1;
            write_res <= wr_ptr_dout[12:1];
        end else if (chipselect && !write && (address == 4'h6)) begin
            write_res_rdy <= 1'b0;
        end else if (chipselect && !write && (address == 4'h0)) begin
            interrupt_status[3] <= 1'b0;
        end

        if (~read_res_rdy && rd_node_dout[0]) begin
            read_res_rdy <= 1'b1;
            interrupt_status[2] <= 1'b1;
            read_res <= rd_node_dout[25:1];
        end else if (chipselect && !write && (address == 4'h5)) begin
            read_res_rdy <= 1'b0;
        end else if (chipselect && !write && (address == 4'h0)) begin
            interrupt_status[2] <= 1'b0;
        end

        if (start_gc_dout[0]) begin
            interrupt_status[4] <= 1'b1;
        end else if (chipselect && !write && (address == 4'h0)) begin
            interrupt_status[4] <= 1'b0;
        end

        if (gc_done_d[0]) begin
            interrupt_status[5] <= 1'b1;
        end else if (chipselect && !write && (address == 4'h0)) begin
            interrupt_status[5] <= 1'b0;
        end

        if (chipselect && write && (address == 4'h7)) begin
            root_rdy <= 1'b1;
        end else if (root_r && root_rdy) begin
            root_rdy <= 1'b0;
            interrupt_status[6] <= 1'b1;
        end else if (chipselect && !write && (address == 4'h0)) begin
            interrupt_status[6] <= 1'b0;
        end
    end

    assign wr_d = {write_op, 1'b1} & {26{write_rdy}};
    assign rd_d = {read_op, 1'b1} & {13{read_rdy}};
    assign wr_ptr_rout = (chipselect && !write && (address == 4'h6));
    assign rd_node_rout = (chipselect && !write && (address == 4'h5));
    assign start_gc_rout = (chipselect && !write && (address == 4'h0));
    assign gc_done_r = (chipselect && !write && (address == 4'h0));
    assign root_d = {root_op, 1'b1} & {14{root_rdy}};

    \Word#_t num_roots_d;
```

```
logic                          num_roots_r;
Root_t root_d;
logic                          root_r;
\Word#_t base_d;
logic                          base_r;
\Word#_t bound_d;
logic                          bound_r;
Go_t gc_done_d;
logic                          gc_done_r;
Pointer_t rd_d;
logic                          rd_r;
Node_t rd_node_dout;
logic                          rd_node_rout;
Node_t wr_d;
logic                          wr_r;
Pointer_t wr_ptr_dout;
logic                          wr_ptr_rout;
Go_t force_gc_d;
logic                          force_gc_r;
Go_t start_gc_dout;
logic                          start_gc_rout;
Raise_Err_t gc_error_dout;
logic                          gc_error_rout;
Node_t mem_error_d;
logic                          mem_error_r;
assign mem_error_r = 1'b1;
\Word#_t threshold_d;
logic                          threshold_r;

Pointer_t gc_read_d;
logic                          gc_read_r;
Node_t gc_node_d;
logic                          gc_node_r;
Pointer_t gc_free_d;
logic                          gc_free_r;
Go_t gc_ack_d;
logic                          gc_ack_r;
Pointer_t mem_read_d;
logic                          mem_read_r;
Node_t mem_node_d;
logic                          mem_node_r;
Go_t mem_ack_d;
logic                          mem_ack_r;
\Word#_t mem_base_d;
logic                          mem_base_r;
\Word#_t mem_bound_d;
logic                          mem_bound_r;
\Word#_t gc_base_d;
logic                          gc_base_r;
\Word#_t gc_bound_d;
logic                          gc_bound_r;
Pointer_t mem_ref_dout;
logic                          mem_ref_rout;
Pointer_t gc_ref_d;
```

```
logic                    gc_ref_r;
Go_t no_gc_dout;
logic                    no_gc_rout;

parameter POINTER_BITS = 13;
parameter NODE_BITS = 26;

WordFork2 #(.POINTER_BITS(POINTER_BITS),
            .NODE_BITS(NODE_BITS)) base_fork(.clk(clk),
                      .input_d(base_d),
                      .input_r(base_r),
                      .output1_dout(mem_base_d),
                      .output1_rout(mem_base_r),
                      .output2_dout(gc_base_d),
                      .output2_rout(gc_base_r)
                      );

WordFork2 #(.POINTER_BITS(POINTER_BITS),
            .NODE_BITS(NODE_BITS)) bound_fork(.clk(clk),
                       .input_d(bound_d),
                       .input_r(bound_r),
                       .output1_dout(mem_bound_d),
                       .output1_rout(mem_bound_r),
                       .output2_dout(gc_bound_d),
                       .output2_rout(gc_bound_r)
                       );

RefFork2 #(.POINTER_BITS(POINTER_BITS),
           .NODE_BITS(NODE_BITS)) ref_fork(.clk(clk),
                  .input_d(mem_ref_dout),
                  .input_r(mem_ref_rout),
                  .output1_dout(wr_ptr_dout),
                  .output1_rout(wr_ptr_rout),
                  .output2_dout(gc_ref_d),
                  .output2_rout(gc_ref_r)
                  );


Memory #(.POINTER_BITS(POINTER_BITS),
         .NODE_BITS(NODE_BITS)) memory(.clk(clk),
                                       .dat_d(wr_d),
                                       .dat_r(wr_r),
                                       .ref_dout(mem_ref_dout),
                                       .ref_rout(mem_ref_rout),
                                       .rd_d(mem_read_d),
                                       .rd_r(mem_read_r),
                                       .rd_node_dout(mem_node_d),
                                       .rd_node_rout(mem_node_r),
                                       .free_d(gc_free_d),
                                       .free_r(gc_free_r),
                                       .init_start_d(mem_base_d),
                                       .init_start_r(mem_base_r),
                                       .init_end_d(mem_bound_d),
                                       .init_end_r(mem_bound_r),
                                                    .free_ack_dout(gc_ack_d),
                                                    .free_ack_rout(gc_ack_r),
                                                    .error_mem_d(mem_error_d),
                                                    .error_mem_r(mem_error_r),
                                                    .bram_addr(bram_addr),
                                                    .bram_line(bram_line)
                                                    );

    Garbage_Collection  #(.POINTER_BITS(POINTER_BITS),
                         .NODE_BITS(NODE_BITS)) gc(.clk(clk),
                          .num_roots_d(num_roots_d),
                          .num_roots_r(num_roots_r),
                          .root_d(root_d),
                          .root_r(root_r),
                          .base_d(gc_base_d),
                          .base_r(gc_base_r),
                          .bound_d(gc_bound_d),
                          .bound_r(gc_bound_r),
                          .done_dout(gc_done_d),
                          .done_rout(gc_done_r),
                          .node_d(gc_node_d),
                          .node_r(gc_node_r),
                          .ack_d(gc_ack_d),
                          .ack_r(gc_ack_r),
                          .read_dout(gc_read_d),
                          .read_rout(gc_read_r),
                          .free_dout(gc_free_d),
                          .free_rout(gc_free_r),
                          .alloc_d(gc_ref_d),
                          .alloc_r(gc_ref_r),
                          .force_gc_d(force_gc_d),
                          .force_gc_r(force_gc_r),
                          .start_gc_dout(start_gc_dout),
                          .start_gc_rout(start_gc_rout),
                          .gc_error_dout(gc_error_dout),
                          .gc_error_rout(gc_error_rout),
                          .threshold_d(threshold_d),
                          .threshold_r(threshold_r)
                          );

    RefNodeMerge2  #(.POINTER_BITS(POINTER_BITS),
                    .NODE_BITS(NODE_BITS)) read_merge(.clk(clk),
                             .cmd0_d(gc_read_d),
                             .cmd0_r(gc_read_r),
                             .cmd1_d(rd_d),
                             .cmd1_r(rd_r),
                             .res_in_d(mem_node_d),
                             .res_in_r(mem_node_r),
                             .res0_dout(gc_node_d),
                             .res0_rout(gc_node_r),
                             .res1_dout(rd_node_dout),
                             .res1_rout(rd_node_rout),
                             .cmd_out_dout(mem_read_d),
                             .cmd_out_rout(mem_read_r)
```

```
                                            );


endmodule // Toplevel
```

B GC_Memory.c

```
/* * Device driver for the GC Memory module
 *
 * A Platform device implemented using the misc subsystem
 *
 * Adapted from vga_ball:
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod GC_Memory.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree GC_Memory.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "GC_Memory.h"

#define DRIVER_NAME "gc_memory"
#define GC_MEMORY_IRQ_NUM 72

/* Device registers */
#define BG_STATUS(x) (x)
#define BG_NUM_ROOTS(x) ((x)+1*4)
#define BG_BASE(x) ((x)+2*4)
#define BG_BOUND(x) ((x)+3*4)
#define BG_THRESHOLD(x) ((x)+4*4)
#define BG_READ(x) ((x)+5*4)
#define BG_WRITE(x) ((x)+6*4)
#define BG_ROOT(x) ((x)+7*4)
#define BG_BRAM(x) ((x)+8*4)

/*
 * Information about our device
 */
struct gc_memory_dev {
  struct resource res; /* Resource: our registers */
  void __iomem *virtbase; /* Where registers can be accessed in memory */
  gc_memory_status_t status;
} dev;

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long gc_memory_ioctl(struct file *f, unsigned int cmd, unsigned long
    arg)
{
  gc_memory_arg_t vla;
  pr_info(DRIVER_NAME ": ioctl");

  switch (cmd) {
  case GC_MEMORY_NUM_ROOTS:
    pr_info(DRIVER_NAME ": set num_roots");
    if (copy_from_user(&vla, (gc_memory_arg_t *) arg,
                       sizeof(gc_memory_arg_t)))
      return -EACCES;
    iowrite32(vla.value, BG_NUM_ROOTS(dev.virtbase));
    break;

  case GC_MEMORY_BASE:
    pr_info(DRIVER_NAME ": set base");
    if (copy_from_user(&vla, (gc_memory_arg_t *) arg,
                       sizeof(gc_memory_arg_t)))
      return -EACCES;
    iowrite32(vla.value, BG_BASE(dev.virtbase));
    break;

  case GC_MEMORY_BOUND:
    pr_info(DRIVER_NAME ": set bound");
    if (copy_from_user(&vla, (gc_memory_arg_t *) arg,
                       sizeof(gc_memory_arg_t)))
      return -EACCES;
    iowrite32(vla.value, BG_BOUND(dev.virtbase));
    break;

  case GC_MEMORY_THRESHOLD:
    pr_info(DRIVER_NAME ": set threshold");
    if (copy_from_user(&vla, (gc_memory_arg_t *) arg,
                       sizeof(gc_memory_arg_t)))
      return -EACCES;
    iowrite32(vla.value, BG_THRESHOLD(dev.virtbase));
    break;

  case GC_MEMORY_READ_OP:
    pr_info(DRIVER_NAME ": read op");
    if (copy_from_user(&vla, (gc_memory_arg_t *) arg,
                     sizeof(gc_memory_arg_t))){
    return -EACCES;
  }
  pr_info(DRIVER_NAME ": read op 2");

  iowrite32(vla.value, BG_READ(dev.virtbase));
  pr_info(DRIVER_NAME ": read op 3");
  break;

case GC_MEMORY_READ_RES:
  pr_info(DRIVER_NAME ": read res");
  vla.value = ioread32(BG_READ(dev.virtbase));
  if (copy_to_user((gc_memory_arg_t *) arg, &vla,
                   sizeof(gc_memory_arg_t)))
    return -EACCES;
  break;

case GC_MEMORY_WRITE_OP:
  pr_info(DRIVER_NAME ": write op");
  if (copy_from_user(&vla, (gc_memory_arg_t *) arg,
                     sizeof(gc_memory_arg_t)))
    return -EACCES;
  iowrite32(vla.value, BG_WRITE(dev.virtbase));
  break;

case GC_MEMORY_WRITE_RES:
  pr_info(DRIVER_NAME ": write res");
  vla.value = ioread32(BG_WRITE(dev.virtbase));
  if (copy_to_user((gc_memory_arg_t *) arg, &vla,
                   sizeof(gc_memory_arg_t)))
    return -EACCES;
  break;

case GC_MEMORY_ROOT:
  pr_info(DRIVER_NAME ": root");
  if (copy_from_user(&vla, (gc_memory_arg_t *) arg,
                     sizeof(gc_memory_arg_t)))
    return -EACCES;
  iowrite32(vla.value, BG_ROOT(dev.virtbase));
  break;

case GC_MEMORY_STATUS:
  pr_info(DRIVER_NAME ": read status reg");
  vla.value = ioread32(BG_STATUS(dev.virtbase));
  if (copy_to_user((gc_memory_arg_t *) arg, &vla,
                   sizeof(gc_memory_arg_t)))
    return -EACCES;
  break;

case GC_MEMORY_READ_BRAM:
  vla.value = ioread32(BG_BRAM(dev.virtbase));
  if (copy_to_user((gc_memory_arg_t *) arg, &vla,
                   sizeof(gc_memory_arg_t)))
    return -EACCES;
    break;

  default:
    return -EINVAL;
  }

  return 0;
}

/* The operations our device knows how to do */
static const struct file_operations gc_memory_fops = {
  .owner                = THIS_MODULE,
  .unlocked_ioctl       = gc_memory_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice gc_memory_misc_device = {
  .minor                = MISC_DYNAMIC_MINOR,
  .name                 = DRIVER_NAME,
  .fops                 = &gc_memory_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init gc_memory_probe(struct platform_device *pdev)
{
  int ret;

  /* Register ourselves as a misc device: creates /dev/gc_memory */
  ret = misc_register(&gc_memory_misc_device);

  /* Get the address of our registers from the device tree */
  ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
  if (ret) {
    ret = -ENOENT;
    goto out_deregister;
  }

  /* Make sure we can use these registers */
  if (request_mem_region(dev.res.start, resource_size(&dev.res),
                         DRIVER_NAME) == NULL) {
    ret = -EBUSY;
    goto out_deregister;
  }

  /* Arrange access to our registers */
  dev.virtbase = of_iomap(pdev->dev.of_node, 0);
  pr_info(DRIVER_NAME ": dev.virtbase = %p", dev.virtbase);
  if (dev.virtbase == NULL) {
    ret = -ENOMEM;
    goto out_release_mem_region;
  }
  return 0;

 out_release_mem_region:
  release_mem_region(dev.res.start, resource_size(&dev.res));
 out_deregister:
  misc_deregister(&gc_memory_misc_device);
  return ret;
}

/* Clean-up code: release resources */
static int gc_memory_remove(struct platform_device *pdev)
{
  iounmap(dev.virtbase);
  release_mem_region(dev.res.start, resource_size(&dev.res));
  misc_deregister(&gc_memory_misc_device);
  return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id gc_memory_of_match[] = {
  { .compatible = "csee4840,gc_memory-1.0" },
  {},
};
MODULE_DEVICE_TABLE(of, gc_memory_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver gc_memory_driver = {
  .driver       = {
    .name       = DRIVER_NAME,
    .owner      = THIS_MODULE,
    .of_match_table = of_match_ptr(gc_memory_of_match),
  },
  .remove       = __exit_p(gc_memory_remove),
};

/* Called when the module is loaded: set things up */
static int __init gc_memory_init(void)
{
  pr_info(DRIVER_NAME ": init\n");
  return platform_driver_probe(&gc_memory_driver, gc_memory_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit gc_memory_exit(void)
{
  platform_driver_unregister(&gc_memory_driver);
  pr_info(DRIVER_NAME ": exit\n");
}

module_init(gc_memory_init);
module_exit(gc_memory_exit);
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Martha Barker (mmb2295)");
MODULE_DESCRIPTION("GC Memory driver");
```

C GC_Memory.h

```
#ifndef _GC_MEMORY_H
#define _GC_MEMORY_H

#include <linux/ioctl.h>
#include <stdbool.h>

typedef struct {
  int read_rdy, write_rdy, read_res_rdy, write_res_rdy, start_gc, gc_done,
      root_rdy;
} gc_memory_status_t;

typedef struct {
  gc_memory_status_t status;
  unsigned int value;
} gc_memory_arg_t;

#define GC_MEMORY_MAGIC 'q'

/* ioctls and their arguments */
#define GC_MEMORY_STATUS     _IOR(GC_MEMORY_MAGIC, 1, gc_memory_arg_t *)
#define GC_MEMORY_NUM_ROOTS  _IOW(GC_MEMORY_MAGIC, 2, gc_memory_arg_t *)
#define GC_MEMORY_BASE       _IOW(GC_MEMORY_MAGIC, 3, gc_memory_arg_t *)
#define GC_MEMORY_BOUND      _IOW(GC_MEMORY_MAGIC, 4, gc_memory_arg_t *)
#define GC_MEMORY_THRESHOLD  _IOW(GC_MEMORY_MAGIC, 5, gc_memory_arg_t *)
#define GC_MEMORY_READ_OP    _IOW(GC_MEMORY_MAGIC, 6, gc_memory_arg_t *)
#define GC_MEMORY_READ_RES   _IOR(GC_MEMORY_MAGIC, 7, gc_memory_arg_t *)
#define GC_MEMORY_WRITE_OP   _IOW(GC_MEMORY_MAGIC, 8, gc_memory_arg_t *)
#define GC_MEMORY_WRITE_RES  _IOR(GC_MEMORY_MAGIC, 9, gc_memory_arg_t *)
#define GC_MEMORY_ROOT       _IOW(GC_MEMORY_MAGIC, 10, gc_memory_arg_t *)
  //#define GC_MEMORY_TEST       _IOR(GC_MEMORY_MAGIC, 11, gc_memory_arg_t *)
#define GC_MEMORY_READ_BRAM  _IOR(GC_MEMORY_MAGIC, 11, gc_memory_arg_t *)

#endif
```

D main.c

```
/*
 * Userspace program that communicates with the gc_memory device driver
 * through ioctls
 *
 * Martha Barker (mmb2295)
 * Columbia University
 */

#include <stdio.h>
#include <stdlib.h>
#include "GC_Memory.h"
#include "reference.h"
#include "hardware.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdbool.h>

int gc_memory_fd;

int base = 0;
int bound = 2048;
int threshold = 16;
int num_roots = 100;
int debug = 0;
int num_ops = 20000;
float w_pr = 0.2;
float r_pr = 0.8;

static unsigned long next_rand = 1;
/* Suggested by the rand(3) manpage */
int myrand()
{
  next_rand = next_rand * 1103515245l + 12345l;
  return next_rand >> 16 & 0x7fffffff;
}

void printHelp(){
  printf(
    "-b <n>          bram base\n"
    "-n <n>          bram bound\n"
    "-t <n>          garbage collection threshold\n"
    "-o \"n n\"      percentage of ops \"write, read, kill pointer\" \n"
    "-r <n>          number of roots\n"
    "-c <n>          number of inputs\n"
    "-d              debug mode\n"
    "-h              print help \n");
}
```

```
void processArgs(int argc, char** argv){
  const char* const short_opts = "b:n:o:dhr:c:t:";

  while(true){
    const int opt = getopt(argc, argv, short_opts);

    if(opt == -1){
      break;
    }

    switch(opt){
    case 'b':
      {
        base = atoi(optarg);
        break;
      }
    case 'n':
      {
        bound = atoi(optarg);
        break;
      }
    case 't':
      {
        threshold = atoi(optarg);
        break;
      }
    case 'd':
      {
        debug = 1;
        break;
      }
    case 'h':
      {
        printHelp();
        break;
      }
    case 'r':
      {
        num_roots = atoi(optarg);
        break;
      }
    case 'c':
      {
        num_ops = atoi(optarg);
        break;
      }
    case 'o':
      {
        char *token = strtok(optarg, " ");

        w_pr = atoi(token)/100.0;
        token = strtok(NULL, " ");
        r_pr = atoi(token)/100.0;
        break;
      }
    default:
      {
        printHelp();
        break;
      }
    }
  }
}

void print_status(gc_memory_status_t *s) {
  printf("----STATUS----\nread rdy: %d \nwrite rdy: %d \nread_res valid: %d \
     nwrite_res valid : %d \nstart gc: %d \ngc done: %d \nroot rdy: %d\n
     ----------------\n", s->read_rdy, s->write_rdy, s->read_res_rdy, s->
     write_res_rdy, s->start_gc, s->gc_done, s->root_rdy);
}

int main(int argc, char *argv[]) {

  static const char filename[] = "/dev/gc_memory";
  gc_memory_status_t status = {0, 0, 0, 0, 0, 0, 0};
  reference_arg_t ref;
  unsigned int *roots;
  unsigned int *hw_bram;
  int writes_in = 0;
  int reads_in = 0;
  int roots_in = 0;
  int num_writes = 0;
  int num_reads = 0;
  int current_roots = 0;
  unsigned int read_op_array[(int)(num_ops*r_pr)+1], write_op_array[(int)(
     num_ops*w_pr)+1], read_res_array[(int)(num_ops*r_pr)+1], write_res_array
     [(int)(num_ops*w_pr)+1];
  int read_op_i = 0, read_res_i = 0, write_op_i = 0, write_res_i = 0;
  int gc_mode = 0;

  printf("GC Memory userspace program started\n");

  if ( (gc_memory_fd = open(filename, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", filename);
    return -1;
  }

  processArgs(argc, argv);
  roots = malloc(num_roots * sizeof(unsigned int));
  hw_bram = malloc((bound-base) * sizeof(unsigned int));


  make_reference(&ref, base, bound);

  init_memory(gc_memory_fd, base, bound, threshold, num_roots);
  status.read_rdy = 1;
  status.write_rdy = 1;
```

```
while ((num_writes + num_reads) < num_ops) {
  update_status(gc_memory_fd, &status);

  if (status.start_gc) {
    gc_mode = 1;
    status.start_gc = 0;
  }

  if ((gc_mode == 1) && (writes_in == 0) && (reads_in == 0)) {
    get_bram(gc_memory_fd, base, bound, hw_bram);
    for (int i = base; i < bound; i++) {
      unsigned int hw = hw_bram[i];
      unsigned int t = (hw & 0x1);
      unsigned int d = ((hw >> 1) & 0xfff);
      unsigned int p = ((hw >> 13) & 0xfff);
      if (t != ref.bram[i].type)
        printf("types don't match: %x %d %d\n", i, t, ref.bram[i].type);
      if (d != ref.bram[i].data)
        printf("data doesn't match: %x %x %x\n", i, d, ref.bram[i].data);
      if (p != ref.bram[i].pointer)
        printf("pointers don't match: %x %x %x\n", i, p, ref.bram[i].pointer
            );
    }
    status.root_rdy = 1;
    printf("do gc\n");
    gc_reference(&ref, base, bound);
    list_roots(roots, &ref, num_roots);

    while(true) {
      update_status(gc_memory_fd, &status);

      if (status.gc_done){
        printf("done gc\n");
        status.gc_done = 0;
        gc_mode = 0;
        roots_in = 0;
        get_bram(gc_memory_fd, base, bound, hw_bram);
        for (int i = base; i < bound; i++) {
          unsigned int hw = hw_bram[i];
          unsigned int t = (hw & 0x1);
          unsigned int d = ((hw >> 1) & 0xfff);
          unsigned int p = ((hw >> 13) & 0xfff);
          if (t != ref.bram[i].type)
            printf("types don't match: %x %d %d\n", i, t, ref.bram[i].type);
          if (d != ref.bram[i].data)
            printf("data doesn't match: %x %x %x\n", i, d, ref.bram[i].data)
                ;
          if (p != ref.bram[i].pointer)
            printf("pointers don't match: %x %x %x\n", i, p, ref.bram[i].
                pointer);
        }
        break;
      }
```

```
        if (status.root_rdy && (roots_in < num_roots)) {
          unsigned int r = roots[roots_in];
          if (r != -1)
            r = ((r & 0xfff) << 1) | (0x0);
          else
            r = ((0 & 0xfff) << 1) | (0x1);
          root_op(gc_memory_fd, &r);
          roots_in++;
          status.root_rdy = 0;
        }
      }
    }

    if (status.read_res_rdy) {
      unsigned int node, ref_node;
      read_res(gc_memory_fd, &node);
      ref_node = read_reference(&ref, read_op_array[read_res_i]);
      read_res_array[read_res_i] = node;
      read_res_i++;
      reads_in--;
      if (node != ref_node){
        printf("ERROR: read results don't match: %x %x\n", node, ref_node);
        return 0;
      }
      status.read_res_rdy = 0;
    }

    if (status.write_res_rdy) {
      unsigned int pointer, ref_pointer;
      write_res(gc_memory_fd, &pointer);
      ref_pointer = write_reference(&ref, write_op_array[write_res_i], myrand
         ()%100+2);
      current_roots++;
      write_res_array[write_res_i] = pointer;
      write_res_i++;
      writes_in--;
      if (pointer != ref_pointer){
        printf("Error: write results don't match: %x %x\n", pointer,
           ref_pointer);
        return 0;
      }
      status.write_res_rdy = 0;
    }

    if (status.read_rdy && (num_reads < num_ops * r_pr) && (current_roots > 0)
        && (gc_mode == 0)) {
      int r = myrand() % current_roots;
      unsigned int pointer = get_root(&ref, r);
      read_op_array[read_op_i] = pointer;
      read_op_i++;
      read_op(gc_memory_fd, &pointer);
      reads_in++;
      num_reads++;
      status.read_rdy = 0;
    }

    if (status.write_rdy && (num_writes < num_ops * w_pr) && (current_roots <
        num_roots) && (gc_mode == 0)) {
      unsigned int pointer;
      unsigned int node;
      if (current_roots > 0){
        pointer = get_root(&ref, (myrand()%current_roots));
        node = ((pointer & 0xfff) << 13) | (((myrand()%12) & 0xfff) << 1) | (0
            x1);
      }else {
        node = ((0 & 0xfff) << 13) | ((0 & 0xfff) << 1) | (0x0);
      }
      write_op_array[write_op_i] = node;
      write_op_i++;
      write_op(gc_memory_fd, &node);
      writes_in++;
      num_writes++;
      status.write_rdy = 0;
      current_roots -= decrement_roots(&ref);

    }
  }

  free(hw_bram);
  free(roots);
  free_reference(&ref);
  printf("GC Memory userspace program terminating\n");
  return 0;
}
```

E hardware.c

```
#include "reference.h"
#include "GC_Memory.h"
#include <stdlib.h>
#include <stdio.h>
#include <sys/ioctl.h>

void init_memory(int gc_memory_fd, int base, int bound, int threshold, int
    num_roots) {

  gc_memory_arg_t vla;

  vla.value = base;
  if (ioctl(gc_memory_fd, GC_MEMORY_BASE, &vla)) {
    perror("ioctl(GC_MEMORY_BASE) failed");
    return;
  }

  vla.value = bound;
  if (ioctl(gc_memory_fd, GC_MEMORY_BOUND, &vla)) {
    perror("ioctl(GC_MEMORY_BOUND) failed");
    return;
  }

  vla.value = threshold;
  if (ioctl(gc_memory_fd, GC_MEMORY_THRESHOLD, &vla)) {
    perror("ioctl(GC_MEMORY_THRESHOLD) failed");
    return;
  }

  vla.value = num_roots;
  if (ioctl(gc_memory_fd, GC_MEMORY_NUM_ROOTS, &vla)) {
    perror("ioctl(GC_MEMORY_NUM_ROOTS) failed");
    return;
  }

}

void write_op(int gc_memory_fd, unsigned int *v) {
  gc_memory_arg_t vla;
  vla.value = *v;
  if (ioctl(gc_memory_fd, GC_MEMORY_WRITE_OP, &vla)) {
    perror("ioctl(GC_MEMORY_WRITE_OP) failed");
    return;
  }
}

void read_op(int gc_memory_fd, unsigned int *v) {
  gc_memory_arg_t vla;
  vla.value = *v;
  if (ioctl(gc_memory_fd, GC_MEMORY_READ_OP, &vla)) {
    perror("ioctl(GC_MEMORY_READ_OP) failed");
    return;
  }
}

void write_res(int gc_memory_fd, unsigned int *v) {
  gc_memory_arg_t vla;

  if (ioctl(gc_memory_fd, GC_MEMORY_WRITE_RES, &vla)) {
    perror("ioctl(GC_MEMORY_WRITE_RES) failed");
    return;
  }
  *v = vla.value;
}

void read_res(int gc_memory_fd, unsigned int *v) {
  gc_memory_arg_t vla;

  if (ioctl(gc_memory_fd, GC_MEMORY_READ_RES, &vla)) {
    perror("ioctl(GC_MEMORY_READ_RES) failed");
    return;
  }
  *v = vla.value;
}

void get_status(int gc_memory_fd, unsigned int *v) {
  gc_memory_arg_t vla;

  if (ioctl(gc_memory_fd, GC_MEMORY_STATUS, &vla)) {
    perror("ioctl(GC_MEMORY_STATUS) failed");
    return;
  }
  *v = vla.value;
}

void update_status(int gc_memory_fd, gc_memory_status_t *status) {
  unsigned int s;
  get_status(gc_memory_fd, &s);

  if ((s >> 0) & 0x1)
    status->read_rdy = 1;
  if ((s >> 1) & 0x1)
    status->write_rdy = 1;
  if ((s >> 2) & 0x1)
    status->read_res_rdy = 1;
  if ((s >> 3) & 0x1)
    status->write_res_rdy = 1;
  if ((s >> 4) & 0x1)
    status->start_gc = 1;
  if ((s >> 5) & 0x1)
    status->gc_done = 1;
  if ((s >> 6) & 0x1)
    status->root_rdy = 1;
}

void get_bram(int gc_memory_fd, int base, int bound, unsigned int *bram) {
  gc_memory_arg_t vla;
  unsigned int v;

  for (int i = base; i < bound; i++) {
    if (ioctl(gc_memory_fd, GC_MEMORY_READ_BRAM, &vla)) {
      perror("ioctl(GC_MEMORY_READ_BRAM) failed");
      return;
    }
    v = vla.value;
    bram[i] = v;
  }
}

void root_op(int gc_memory_fd, unsigned int *v) {
  gc_memory_arg_t vla;
  vla.value = *v;
  if (ioctl(gc_memory_fd, GC_MEMORY_ROOT, &vla)) {
    perror("ioctl(GC_MEMORY_ROOT) failed");
    return;
  }
}
```

F hardware.h

```
#ifndef _HARDWARE_H
#define _HARDWARE_H

void init_memory(int gc_memory_fd, int base, int bound, int threshold, int
    num_roots);

void write_op(int gc_memory_fd, unsigned int *v);

void read_op(int gc_memory_fd, unsigned int *v);

void write_res(int gc_memory_fd, unsigned int *v);

void read_res(int gc_memory_fd, unsigned int *v);

void get_status(int gc_memory_fd, unsigned int *v);

void update_status(int gc_memory_fd, gc_memory_status_t *status);

void get_bram(int gc_memory_fd, int base, int bound, unsigned int *bram);

void root_op(int gc_memory_fd, unsigned int *v);

//void garbage_collection(int gc_memory_fd, unsigned int *roots, int *
    writes_in, int *reads_in);

#endif
```

G reference.c

```
#include "reference.h"
#include <stdlib.h>
#include <stdio.h>

void make_reference(reference_arg_t *ref, int base, int bound) {

  ref->head = base;
  ref->tail = bound-1;
  ref->bram = malloc((bound-base) * sizeof(bram_entry_t));
  ref->roots = NULL;

  for(int i = base; i < bound-1; i++) {
    ref->bram[i].type = 1;
    ref->bram[i].data = 0;
    ref->bram[i].pointer = i+1;
    ref->bram[i].marked = 2;
  }
  ref->bram[bound-1].type = 1;
  ref->bram[bound-1].marked = 2;
  ref->bram[bound-1].pointer = bound;
}

void free_reference(reference_arg_t *ref) {
  free(ref->bram);
  root_t *traverse = ref->roots;
  root_t *to_free;

  while (traverse != NULL) {
    to_free = traverse;
    traverse = traverse->next;
    free(to_free);
  }
}

unsigned int write_reference(reference_arg_t *ref, unsigned int node, int
    cycles){
  unsigned int t, d, p;
  unsigned int next, current;
  root_t *new_root;

  current = ref->head;
  next = ref->bram[ref->head].pointer;

  t = (node & 0x1);
  d = ((node >> 1) & 0xfff);
  p = ((node >> 13) & 0xfff);

  ref->bram[current].type = t;
  ref->bram[current].data = d;
  ref->bram[current].pointer = p;
  ref->bram[current].marked = 0;
  ref->head = next;

  new_root = malloc(sizeof(root_t));
  new_root->pointer = current;
  new_root->cycles = cycles;
  new_root->next = ref->roots;
  ref->roots = new_root;

  return current;
}

unsigned int read_reference(reference_arg_t *ref, unsigned int pointer){
  unsigned int t, d, p;
  unsigned int node;

  t = ref->bram[pointer].type;
  d = ref->bram[pointer].data;
  p = ref->bram[pointer].pointer;

  node = ((p & 0xfff) << 13) | ((d & 0xfff) << 1) | (t & 0x1);
  return node;
}

int decrement_roots(reference_arg_t *ref) {
  root_t *traverse = ref->roots;
  root_t *previous = NULL;
  int removed = 0;
  root_t *to_free;

  while (traverse != NULL) {
    traverse->cycles-=1;
    if (traverse->cycles == 0) {
      removed++;
      if(traverse == ref->roots) {
        to_free = ref->roots;
        ref->roots = ref->roots->next;
        traverse = traverse->next;
        free(to_free);
      }else {
        to_free = traverse;
        previous->next = traverse->next;
        traverse = traverse->next;
        free(to_free);
      }
    }else {
      previous = traverse;
      traverse = traverse->next;
    }
  }

  return removed;
}

void gc_reference(reference_arg_t *ref, int base, int bound) {
  root_t *traverse = ref->roots;
  unsigned int prev = -1;

  while (traverse != NULL) {
    int addr = traverse->pointer;
    bram_entry_t entry = ref->bram[addr];

    while (1) {
      if (entry.marked == 1)
        break;

      ref->bram[addr].marked = 1;

      if (entry.type == 0)
        break;

      addr = entry.pointer;
      entry = ref->bram[addr];

    }
    traverse = traverse->next;
  }

  for (int i = base; i < bound; i++) {
    if (ref->bram[i].marked == 0) {
      ref->bram[i].marked = 2;
      ref->bram[i].type = 0;
      ref->bram[i].data = 0;
      ref->bram[i].pointer = i;

      ref->bram[ref->tail].type = 1;
      ref->bram[ref->tail].pointer = i;

      ref->tail = i;
    } else if (ref->bram[i].marked == 1) {
      ref->bram[i].marked = 0;
    }
  }
}

void list_roots(unsigned int *roots, reference_arg_t *ref, int num_roots) {
  root_t *traverse;
  int i;

  traverse = ref->roots;
  i = 0;

  while (traverse != NULL) {
    roots[i] = traverse->pointer;
    traverse = traverse->next;
    i++;
  }

  while (i < num_roots) {
    roots[i] = -1;
    i++;
  }
}

unsigned int get_root(reference_arg_t *ref, int num){
  root_t *traverse = ref->roots;
  int i = 0;

  while (i < num) {
    i++;
    if (traverse->next == NULL)
      return -1;
    else
      traverse = traverse->next;
  }
  return traverse->pointer;
}
```

H reference.h

```
#ifndef _REFERENCE_H
#define _REFERENCE_H

typedef struct {
  int type;
  unsigned int data;
  unsigned int pointer;
  int marked;
} bram_entry_t;

typedef struct root_t {
  unsigned int pointer;
  int cycles;
  struct root_t *next;
} root_t;

typedef struct {
  unsigned int head;
  unsigned int tail;
  bram_entry_t *bram;
  root_t *roots;
} reference_arg_t;

void make_reference(reference_arg_t *ref, int base, int bound);

unsigned int write_reference(reference_arg_t *ref, unsigned int node, int
    cycles);

unsigned int read_reference(reference_arg_t *ref, unsigned int pointer);

void free_reference(reference_arg_t *ref);

int decrement_roots(reference_arg_t *ref);

void gc_reference(reference_arg_t *ref, int base, int bound);

void list_roots(unsigned int *roots, reference_arg_t *ref, int num_roots);

unsigned int get_root(reference_arg_t *ref, int key);

#endif
```