16 Core Deterministic Multiprocessor

David Greenberg

Abstract

Multiprocessing architectures traditionally suffer from nondeterministic execution. This project seeks to implement a hardware cache system based on the work by Bergan et al. on CoreDet. The system will allow the processor to execute in parallel until interprocessor communication is detected. Then, all processors will be halted and the pending memory writes will be reordered and flushed. By deterministically reordering memory accesses, execution will appear deterministic from the perspective of the main memory without losing parallelism. The cache will be comprised of one fully associative storage buffer per processor along with an arbitration unit that will conduct the cache coherency protocol.

Table of Contents

Overview

Deterministic multiprocessing systems can provide important benefits in many applications. A deterministic multiprocessing system has the same output given the same input. In this system, the inputs to the system to are the code executed by the processors and the data supplied to the program when it starts running. The nondeterminism will be introduced by allowing processors to be stalled or halted arbitrarily by the switches on the labkit. This simulates stalls due to contention on the buses in other multiprocessing systems.

Concept

This implementation of deterministic multiprocessing is essentially a time-slicing multitasking system in which the time slices are executed in parallel, and the memory operations are subsequently reordered. In order to recover determinism, we must look at how a time-slicing multitasking system orders reads and writes to memory. Essentially, reads and writes only must ordered on the inter-time slice level, since within a single time slice the ordering of the reads and writes is given by the order of instructions executed. We will snoop inter-thread communications and enforce a deterministic ordering on them. By enforcing determinism on the time-slice level, the overall system will be deterministic, since the contents of memory will change in the same way across multiple executions of the same program.

High Level Design

In order to enforce determinism between time slices, we will implement this system as a cache that sits between the processors and the main memory. To account for the time slices, each processor will incorporate an instruction counter, as this gives us fine-grained deterministic time-slicing (a timer interrupt would not work because it would be nondeterministic if the interrupt didn't have the highest preemption priority). We will use a scheme similar to CoreDet in which we buffer all loads and stores in per-processor caches. Since we store all this information, we can construct a snooping protocol that will detect communication between threads. We can rely on each processor to remain self-consistent, and allow them to run without synchronization until the end of a time slice is reached. Determinism and synchronization is handled by a protocol that causes all pending writes to be flushed quickly and deterministically.

Core Algorithm

The system operates by moving through two phases: the parallel phase and the commit phase. The parallel phase is when the bulk of the work is done by all processors in parallel. The processors halt when reach the end of their time slice. When all processors are halted, they move to the commit phase, where pending writes are flushed to main memory in a deterministic order.

The system is broadly divided into 5 components: the Processors (CPUs), the Arbitrators, the storage buffers (content addressable memory), the read multiplexer, and the deterministic committer. The CPUs execute and issue reads and writes to the arbitrators. The arbitrators maintain coherency in the storage buffers by correctly dispatching reads and writes. The committer executes the deterministic commit algorithm, and the read multiplexer allows multiple readers to share a single port. In addition, there is a state machine that maintains which phase is currently being executed, as well as several support modules like the instruction counters, main memory controller, and memory-mapped sprite-based VGA graphics.

Modules

CAM cell

The CAM cell is the fundamental unit of the storage buffer. It supports asynchronous reads and synchronous writes, and it also can be queried as to whether the (address, data) pair it contains matches a given query address. It also has a reset signal to clear it, and an empty signal that indicates whether it contains data.

Reducer

The reducer is a tree network with branching factor 2 that propagates the highest priority input which is valid to its output. It does this by combining a 2 to 1 reduction primitive into a tree, where the primitive propagates the value of its first input if it is valid, and otherwise it propagates the value of its second input if it's valid. This produces a natural ordering that is reflected in the N to 1 reducer. The reducer has two modes: it can either propagate the highest priority input to its output, or it can propagate the index of the highest priority input to its output. For example, the former mode is used to propagate the address and data of a matched CAM cell to the output of the storage buffer, and the latter mode is used to find the index of an empty CAM cell when there is new data to be written. The reducer is implemented using generate statements, so that any power of two size can be easily be specified with parameters. The propagation mode is also specified with a parameter.

Storage Buffer

The storage buffer is used to hold writes from the CPUs to main memory until the commit phase is reached. The storage buffer is a CAM memory with asynchronous read, synchronous write, and synchronous clear. It also reports on whether there are any free CAM cells and whether the queried address is stored in the buffer. In addition to this normal CAM memory functionality, the storage buffer outputs the address and data of some written memory cell if there is at least one written cell in the buffer. This output is used to flush the contents of the buffer one cell at a time during the commit phase. Identifying an empty cell and identifying a written cell are handled by a pair of reducers. The cells themselves are generated, and their number is determined by a parameter. Each cell's write and clear inputs are gated by whether the cell matches the input address. In addition, in the case of writing a cell, if there is no matching cell, the cell whose index matches the index chosen by the empty-cell-finding reducer is written.

Arbiter

The arbiter handles reads and writes from the CPU. It decides whether to read from the storage buffer or main memory. It handles negotiating with the main memory's read port multiplexer during the parallel phase and with the commit module during the commit phase. It also stalls the CPU by gating its clock when a request to memory takes longer than one clock cycle. The arbiter contains a storage buffer for caching writes and a state machine for handling reads. If the requested address is already in the storage buffer, it is read from there. Otherwise, the CPU is halted and a request is triggered to the main memory multiplexer. Once the request is acknowledged by the multiplexer, the main memory's data is handed off to the CPU, and the CPU is allowed to proceed with its computation. The arbiter also will also halt the CPU during the commit phase or if the CPU exhausted its quantum. The arbiter also gates the write signal to the storage buffer so that the storage buffer can only be written during the parallel phase. Initially I did not do this, and the result was that under certain circumstances the clear signal and the write signal would both act on the same cell during the commit phase, causing alternating cells to be written and a deadlock to result.

Committer

The committer flushes the storage buffers' write caches to the main memory, deterministically prioritizing conflicting writes. Internally, it has two reducers that select the first written memory location from the highest priority arbiter. As long as that location exists, it is written to main memory. The write signal is not only connected to the write enable of the main memory, but also to the clear signal of the other arbiters. The result is that any other arbiter whose storage buffer contains that memory location has it deleted when it is written, and so only the highest priority arbiter gets to write that memory location.

Shuffle Network

The shuffle network uses N N-input reducers and an N-bit rotating counter to select one input of each reducer to go to its output. The reducers' inputs are all rotated by one from the previous reducer. The result is that as the counter rotates, the outputs rotate relative to the input. This allows for round robin scheduling for the read multiplexer.

Read Multiplexer

The read multiplexer resolves simultaneous reads to main memory. Each input is an address and a request line, which feeds into a reducer. The highest priority request is propagated to the main memory, and at each clock all of the requests whose address match the current request receive an acknowledge signal, which indicates that the current output of the main memory is what they should store. This allows the requester to de-assert its request signal. In order to ensure that all one requester cannot starve the others, a shuffle network sits in front of the inputs and reshuffles them during every quiescent period. This ensures that no reads receive erroneous data.

Micro8

The micro8 is a microcontroller we studied in lecture. I added two enhancements to it for my project. The first enhancement is an instruction counter. On each clock, the counter is increased until it reaches a value determined by an input. At this point, it asserts an output indicated it's reached the desired count. This output is fed into the arbiter, which uses it to gate the clock to the micro8. The second enhancement is a new opcode, CPUID. This loads a value determined by a parameter into a register. During the generation of the CPU cores, this value is set to be the index of the CPU core. This allows each core to indirectly address the memory mapped sprite that corresponds to the progress of its computation in order to give graphical updates.

Sprite

The sprite is a memory mapped bar graph based on the rectangle sprite in the pong lab. I added an additional layer of complexity that allows it to request an update from main memory every couple milliseconds. The requests from the sprites are multiplexed by the memory multiplexer.

Master State Machine

The master state machine determines the phase of the system (parallel or commit). It also allows the system to run either deterministically or nondeterministically. Determinism is handled as described in the algorithm's section. Nondeterminism is handled by allowing the system to move to commit phase when any processor is ready to commit (rather than waiting for every processor to reach the barrier). This means that if a processor stalls during the nondeterministic phase, it will not run the same as if it hadn't stalled. Transitions from deterministic to nondeterministic operation only occur when the system switches from the commit phase to the parallel phase. This prevents other race conditions I observed that can disrupt the deterministic mode after several mode transitions.

Memory

The main memory is a dual port read-after-write BRAM. The committer and one multiplexer are connected to one port of the memory, and they're multiplexed by the phase of the system. The arbiters are connected to the commiter and multiplexer. The sprites are connected to the other port of the memory via another multiplexer.

Hardware

In addition to the system described above, the first eight CPUs are connected to switches on the labkit that gate their clocks, allowing them to be arbitrarily stalled. Also, the determinism can be switched on or off by pressing a button.

Testing

In order to facilitate the possibility of completing this project, I extensively tested every component, primarily using a unit testing framework I developed. Both modes of the reducer were tested 8 and 16 value inputs and verified against several cases, including one valid input, several valid inputs, and no valid inputs. The CAM cells were tested by verifying that their status was correct after several read, write, and clear cycles. To test the storage buffer, I wrote tasks that would execute a read cycle and verify its result, a write cycle and verify its result, and a clear cycle and verify its result. I also wrote tasks to check the expected state of the storage buffer. The test itself did a variety of reads, writes, and clears, ensuring that overflow and underflow were correctly detected and that cells were reused when writing to an address already in the buffer and allocated when writing to a new address.

The shuffle network was tested by manually verifying that a sequence of inputs were shuffled in the simulator. To test the components that interacted with main memory, I wrote a behavioral simulation of the Virtex II's BRAM. I wrote a test to verify that its read and write cycles worked correctly.

To test the committer, I connected four arbiters to a committer, and wrote tasks that would emulate a write cycle from a CPU to the arbiters' storage buffers. Then, I enabled the committer and waited for enough cycles that it had time to finish committing the data. Finally, I used another task I wrote to verify the contents of the BRAM were what I expected. Additionally, I added guards to ensure that every node completed its flush and didn't become stalled.

The read multiplexer was the most difficult module to test. In order to test it, I had to instantiate a simulated BRAM, load it up with lots of memory locations, and then simulate sequential and simultaneous read requests from the four arbiters. The read tasks made use of wait statements to delay their completion until their request was acknowledged and the data was verified. To simulate parallel reads, fork/join blocks were used.

To test the Micro8, I wired up the entire system along with the test program I wrote directly in binary and then I manually checked that the output was correct. The only testing the sprite underwent was that it correctly handled an acknowledge when it requested a read.

All other testing and verification was done on the hardware itself. To facilitate testing on the actual FPGA, until I had the entire system running I only synthesized two cores with an 8 cell storage buffer, which greatly reduced synthesis times. It took about six hours to get the system working on hardware from simulation. It took another 8 hours to find and fix many bugs related to the button and switch inputs. Most of these issues had to do with changes occurring at unexpected times. The solutions usually involved registering the desired change and deferring it to align with a clock transition or state transition.

Tools

I used the Icarus verilog simulator to simulate my designs, since it is fast and command-line based. I used GTKWave to view the waveforms from the simulation. All of my code was stored in a git repository, which helped me track my progress, back up my work, and undo mistakes. I wrote a Makefile to speed up my development. Initially, it only supported the targets “build,” “simulate,” and “view waveform,” but I later added a “publish” target. In order to continue to develop in my Linux environment, I moved my git repository to my Dropbox account. I installed Dropbox on the lab computer and put my ISE project into the Dropbox as well. The "publish" target copied the verilog files from the git repository into the ISE project, allowing me to continue to do the majority of my simulation and testing on Linux, but still take advantage of the Xilinx toolchain for synthesis and analysis.

Conclusion

This project was very successful due to the extensive and rigorous testing all of the modules underwent prior to synthesis. Most issues were entirely avoided since the testing revealed all of the bugs in implementation of individual modules and in the ways the modules interacted. All of the issues encountered in the lab were related to the manual clock gating mechanism and the switch between deterministic and nondeterministic processing, since these were the only untested components of the system. In the end, a 16 core deterministic multiprocessor was successfully synthesized and tested on the FPGA hardware.

arbiter.v

```
‘default_nettype none

//this arbitrates all memory accesses that the cpu might do
//it buffers writes and flushes them out during the commit phase
//it also makes reads to main memory if the read isn’t in the storage buffer

//because of the construction of the arbiter, the CPU must be clocked at no fast
er
//than half of the arbiter’s clock
module arbiter(clk, rst, system_state, system_finished,
    cpu_addr, cpu_data_in, cpu_data_out, cpu_write, cpu_read, cpu_halt,
    flush_addr_out, flush_data_out, flush_out_valid, flush_addr_in, flush_in_val
id,
    mainmem_addr, mainmem_req, mainmem_ack, mainmem_data,
    out_of_memory, count_reached);
  parameter ADDR_WIDTH = 8;
  parameter DATA_WIDTH = 8;
  parameter LOG_STOBUF_SIZE = 6;

  localparam STOBUF_SIZE = 2<<LOG_STOBUF_SIZE;

  //These describe which mode the system is in
  localparam SYSTEM_PARALLEL = 0;
  localparam SYSTEM_COMMIT = 1;

  input wire clk, rst;
  //goes high if the storage buffer will overflow
  output wire out_of_memory;

  //contains the system’s mode
  input wire system_state;

  //This connects the arb to the CPU’s memory ports, and gives it a way to halt
the CPU
  input wire [ADDR_WIDTH-1:0] cpu_addr;
  input wire [DATA_WIDTH-1:0] cpu_data_in;
  output reg [DATA_WIDTH-1:0] cpu_data_out;
  input wire cpu_write, cpu_read;
  output cpu_halt;

  //used during commit mode to flush cells of memory to main memory
  output wire [ADDR_WIDTH-1:0] flush_addr_out;
  output wire [DATA_WIDTH-1:0] flush_data_out;
  output wire flush_out_valid;
  //used during commit mode to delete cells that other arbs already flushed
  input wire [ADDR_WIDTH-1:0] flush_addr_in;
  input wire flush_in_valid;

  //this represents if the attached CPU has halted due to reaching its instructi
on quota
  input wire count_reached;
  //goes high when the arbiter has finished its job (commit or running instructi
ons)
  output wire system_finished;
  assign system_finished  = system_state == SYSTEM_PARALLEL ? count_reached : ~f
lush_out_valid;

  //this is data the output from the storage buffer
  wire [DATA_WIDTH-1:0] stobuf_data;
  //storage buffer accesses CPU’s data when in parallel mode and prepares for fl
ushing otherwise
  wire [ADDR_WIDTH−1:0] stobuf_addr = system_state == SYSTEM_PARALLEL ? cpu_addr
 : flush_addr_in;
  wire stobuf_contains_addr;
  wire stobuf_not_full;

  assign out_of_memory = ~stobuf_not_full;

  //used to read from main memory
  //address requested from main memory comes from cpu
  output wire [ADDR_WIDTH−1:0] mainmem_addr;
  assign mainmem_addr = cpu_addr;
  //only need to request addr if storage buffer doesn’t contain it
  output reg mainmem_req;
  //goes high when the request is serviced
  input wire mainmem_ack;
  input wire [DATA_WIDTH−1:0] mainmem_data;

  storage_buffer #(.ADDR_WIDTH(ADDR_WIDTH), .DATA_WIDTH(DATA_WIDTH), .LOG_CELL_C
OUNT(LOG_STOBUF_SIZE))
    stobuf(.clk(clk), .rst(rst),
           .address(stobuf_addr), .data_in(cpu_data_in), .data_out(stobuf_data),
           .write(cpu_write && system_state == SYSTEM_PARALLEL), .clear(flush_in
_valid),
           .read_valid(stobuf_contains_addr), .write_valid(stobuf_not_full),
           .stored_address(flush_addr_out), .stored_data(flush_data_out),
           .stored_address_valid(flush_out_valid));

  //halt the cpu if we’re committing or waiting on main memory for a read
  reg waiting_on_mainmem;
  assign cpu_halt = waiting_on_mainmem || (system_state == SYSTEM_COMMIT) || cou
nt_reached;

  always @(posedge clk)
    if (rst) begin
      mainmem_req <= 0;
      cpu_data_out <= 0;
      waiting_on_mainmem <= 0;
    end else
    if (system_state == SYSTEM_PARALLEL) begin
    //if the storage buffer doesn’t contain an address, halt the processor until
    //we are serviced by the BRAM

    if (!count_reached && cpu_read) begin
      if (stobuf_contains_addr) begin
      //register storage buffer’s data
        cpu_data_out <= stobuf_data;
      end else if (!mainmem_ack) begin
        //halt if stobuf doesn’t contain addr
        mainmem_req <= 1;
        waiting_on_mainmem <= 1;
      end else if (!mainmem_req) begin
        //unhalt if there’s no request or ack
        waiting_on_mainmem <= 0;
      end else begin //getting here means req && ack are true
        //we’ve got the result, continue
        mainmem_req <= 0;
        cpu_data_out <= mainmem_data;
      end
    end
  end
endmodule

//This module, when enabled, flushes the address/data pairs that are valid on
//its input to a BRAM in a deterministic sequence
module committer(clk, en, addr_in, valid_in, data_in, addr_out, data_out, write)
;
  parameter ADDR_WIDTH = 8;
  parameter DATA_WIDTH = 8;
  parameter LOG_INPUT_COUNT = 2;

  localparam INPUT_COUNT = 1<<LOG_INPUT_COUNT;

  input wire clk, en;

  input wire [(ADDR_WIDTH*INPUT_COUNT)-1:0] addr_in;
  input wire [(DATA_WIDTH*INPUT_COUNT)-1:0] data_in;
  input wire [INPUT_COUNT-1:0] valid_in;

  output reg [ADDR_WIDTH-1:0] addr_out;
  output reg [DATA_WIDTH-1:0] data_out;
  output reg write;

  wire valid;
  wire [ADDR_WIDTH-1:0] addr;
  wire [DATA_WIDTH-1:0] data;

  reducer #(.DATA_WIDTH(DATA_WIDTH), .INPUT_BITS(LOG_INPUT_COUNT), .PROPAGATOR(1
))
     reduce_data(.in_data(data_in), .in_valid(valid_in), .out(data), .valid());
  reducer #(.DATA_WIDTH(ADDR_WIDTH), .INPUT_BITS(LOG_INPUT_COUNT), .PROPAGATOR(1
))
     reduce_addr(.in_data(addr_in), .in_valid(valid_in), .out(addr), .valid(vali
d));

  //write if we have a valid cell and we’re enabled
  //multiple writes registering doesn’t matter, since the result is the same
  always @(posedge clk) if (en) begin
    if (!write && valid) begin
      write <= 1;
      addr_out <= addr;
      data_out <= data;
    end else begin
      write <= 0;
    end
  end
endmodule

//This module shuffles its inputs and pushes them to the output
module shuffle_network(clk, shuffle, rst, data_in, data_out);
  parameter DATA_WIDTH = 8;
  parameter LOG_INPUT_COUNT = 2;

  localparam INPUT_COUNT = 1<<LOG_INPUT_COUNT;

  input wire clk, rst, shuffle;
  input wire [(DATA_WIDTH*INPUT_COUNT)-1:0] data_in;
  output wire [(DATA_WIDTH*INPUT_COUNT)-1:0] data_out;

  reg [INPUT_COUNT-1:0] pos;

  genvar i;
  generate for (i = 0; i < INPUT_COUNT; i = i + 1) begin: shuffle_stage
    wire [INPUT_COUNT-1:0] shuffle_pos;

    if (i == 0) begin: shuffle_pos_0
      assign shuffle_pos = pos;
    end
    else if (i == INPUT_COUNT - 1) begin: shuffle_pos_n
      assign shuffle_pos = {pos[INPUT_COUNT-2:0],pos[INPUT_COUNT-1]};
    end
    else begin: shuffle_pos_i
      assign shuffle_pos = {pos[i:0],pos[INPUT_COUNT-1:i]};
    end

    reducer #(.DATA_WIDTH(DATA_WIDTH), .INPUT_BITS(LOG_INPUT_COUNT), .PROPAGATOR
(1))
      shuffle_reducer(.in_data(data_in), .in_valid(shuffle_pos),
                      .out(data_out[DATA_WIDTH*(i+1)-1:DATA_WIDTH*i]), .valid())
;
  end
  endgenerate

  always @(posedge clk) if (rst) begin
    pos <= 1;
  end else if (shuffle) begin
    pos <= {pos[INPUT_COUNT-2:0],pos[INPUT_COUNT-1]};
  end
endmodule

//this test really just needs to be verified in a waveform viewer
module test_shuffle_network;
  reg clk, rst;
  reg [3:0] data0, data1, data2, data3;
  wire [15:0] data_out, data_in;
  wire [3:0] out0, out1, out2, out3;
  assign data_in = {data0,data1,data2,data3};
  assign out0 = data_out[3:0];
  assign out1 = data_out[7:4];
  assign out2 = data_out[11:8];
  assign out3 = data_out[15:12];

  shuffle_network #(.DATA_WIDTH(4), .LOG_INPUT_COUNT(2))
    shuffler(.clk(clk), .rst(rst), .data_in(data_in), .data_out(data_out), .shuf
fle(1));

  always #2 clk = ~clk;

  initial begin
    $display("(II) starting shuffle network test");
    clk = 0;
    rst = 1;
    data0 = 2;
    data1 = 4;
    data2 = 6;
    data3 = 8;
    #4 rst = 0;
    #100
    $display("(II) finished shuffle network test");
  end
endmodule

//this module uses a robin robin scheduler to allow multiple readers to access
//a shared BRAM though a simple syn/ack handshaking protocol
module read_multiplexer(clk, rst, addr_in, syn_in, ack_out, addr_out);
  parameter ADDR_WIDTH = 8;
  parameter LOG_INPUT_COUNT = 2;

  localparam INPUT_COUNT = 1<<LOG_INPUT_COUNT;

  input wire clk, rst;

  input wire [(ADDR_WIDTH*INPUT_COUNT)-1:0] addr_in;
  input wire [INPUT_COUNT-1:0] syn_in;

  output reg [INPUT_COUNT-1:0] ack_out;
  output wire [ADDR_WIDTH-1:0] addr_out;

  wire [((1+ADDR_WIDTH)*INPUT_COUNT)-1:0] shuffle_in;
  wire [((1+ADDR_WIDTH)*INPUT_COUNT)-1:0] shuffle_out;
  wire [(ADDR_WIDTH*INPUT_COUNT)-1:0] shuffled_addr;
  wire [INPUT_COUNT-1:0] shuffled_syn;

  //package the syn signals and addresses into a shuffable vector
  genvar j;
  generate for (j = 0; j < INPUT_COUNT; j = j + 1) begin: pack_for_shuffle
    assign shuffle_in[(1+ADDR_WIDTH)*(j+1)-1:(1+ADDR_WIDTH)*j] =
      {syn_in[j],addr_in[ADDR_WIDTH*(j+1)-1:ADDR_WIDTH*j]};
    assign shuffled_syn[j] = shuffle_out[(1+ADDR_WIDTH)*(j+1)-1];
    assign shuffled_addr[ADDR_WIDTH*(j+1)-1:ADDR_WIDTH*j] =
      shuffle_out[(1+ADDR_WIDTH)*(j+1)-2:(1+ADDR_WIDTH)*j];
  end
  endgenerate

  //shuffle the vector once per idle period
  reg shuffle, shuffle_trigged;
  always @(posedge clk) begin
    if (shuffle && shuffle_trigged) shuffle <= 0;
    if (~|ack_out) begin
      if (!shuffle_trigged) begin
        shuffle <= 1;
        shuffle_trigged <= 1;
      end
    end else shuffle_trigged <= 0;
  end
  shuffle_network #(.DATA_WIDTH(ADDR_WIDTH+1), .LOG_INPUT_COUNT(LOG_INPUT_COUNT)
)
    shuffle_addr(.clk(clk), .shuffle(shuffle), .rst(rst), .data_in(shuffle_in),
.data_out(shuffle_out));

  reducer #(.DATA_WIDTH(ADDR_WIDTH), .INPUT_BITS(LOG_INPUT_COUNT), .PROPAGATOR(1
))
    reduce_addr(.in_data(shuffled_addr), .in_valid(shuffled_syn), .out(addr_out)
, .valid());
```

```
  //if multiple readers are waiting for an address, one read will satisfy them a
ll
  genvar i;
  generate for (i = 0; i < INPUT_COUNT; i = i + 1) begin: ack
    always @(posedge clk)
      ack_out[i] = addr_in[ADDR_WIDTH*(i+1)-1:ADDR_WIDTH*i] == addr_out && syn_i
n[i];
  end
  endgenerate
endmodule

//this module simulates a BRAM on the virtex-ii (specifically the RAMB4_S8, I th
ink)
module bram(clk, do, addr, di, en, rst, we);
  parameter ADDR_WIDTH = 8;
  parameter DATA_WIDTH = 8;

  localparam CELL_COUNT = 1<<ADDR_WIDTH;

  reg [DATA_WIDTH-1:0] cells [CELL_COUNT-1:0];

  output reg [DATA_WIDTH-1:0] do;
  input wire [ADDR_WIDTH-1:0] addr;
  input wire [DATA_WIDTH-1:0] di;
  input wire clk, en, rst, we;

  always @(posedge clk)
  if (en) begin
    if (we) begin
      cells[addr] <= di;
      if (!rst) do <= di;
    end
    if (rst) begin
      do <= 0;
    end
    if (!we && !rst) do <= cells[addr];
  end
endmodule

//this test and a thorough test are very different ;)
module test_bram;
  reg clk, en, rst, we;
  reg [7:0] addr, di;
  wire [7:0] do;

  bram memory(clk, do, addr, di, en, rst, we);

  always #2 clk = ~clk;

  task do_read;
    input [7:0] addr_in;
    input [7:0] data_ex;
    begin
      #4 en = 1;
      rst = 0;
      we = 0;
      addr = addr_in;
      #4
      if (do != data_ex) begin
        $display("expected to read %d at addr %d, but saw %d instead", data_ex,addr_in,do);
        $finish;
      end
    end
  endtask

  task do_write;
    input [7:0] addr_in;
    input [7:0] data_in;
    begin
      #4 en = 1;
      rst = 0;
      we = 1;
      addr = addr_in;
      di = data_in;
      #4 we = 0;
    end
  endtask
  initial begin
    $display("(II) starting bram test");
    clk = 0;
    en = 0;
    rst = 0;
    we = 0;
    addr = 0;
    di = 0;

    do_write(32,42);
    do_write(33,43);
    do_write(34,44);
    do_write(35,45);
    do_write(36,46);

    do_read(32,42);
    do_read(33,43);
    do_read(34,44);
    do_read(35,45);
    do_read(36,46);

    do_write(33,37);
    do_read(36,46);
    do_read(33,37);

    $display("(II) finished bram test successfully");
  end
endmodule

module test_commit;
  localparam LOG_NODES = 2;
  localparam NODES = 1<<LOG_NODES;
//first, must set up a bunch of arbiters
  //general connections
  //use en to enable or disable committing of buffers
  reg clk, rst, en;
  wire system_state = en;
  //commit flush connections
  wire [8*NODES-1:0] flush_addrs;
  wire [8*NODES-1:0] flush_datas;
  wire [NODES−1:0] flush_valids;
  wire [NODES−1:0] cpu_halts;

  //into the committer
  wire [7:0] cur_flush_addr;
  wire cur_flush_valid;

  genvar i;
  generate for (i = 0; i < NODES; i = i + 1) begin: node
    reg [7:0] cpu_addr, cpu_data_in;
    reg cpu_write;
    wire [7:0] cpu_data_out;
    wire sys_fin;

    initial cpu_write = 0;

    arbiter arb(.clk(clk), .rst(rst), .system_state(system_state), .system_finis
hed(sys_fin),
      .cpu_addr(cpu_addr), .cpu_data_out(cpu_data_out), .cpu_data_in(cpu_data_in
),
      .cpu_write(cpu_write), .cpu_read(0), .count_reached(1),
      .cpu_halt(cpu_halts[i]), .flush_addr_out(flush_addrs[8*(i+1)−1:8*i]),
      .flush_out_valid(flush_valids[i]), .flush_data_out(flush_datas[8*(i+1)−1:8
*i]),
      .flush_addr_in(cur_flush_addr), .flush_in_valid(cur_flush_valid), .out_of_
memory(),
      .mainmem_addr(), .mainmem_req(), .mainmem_ack(), .mainmem_data());
  end
  endgenerate

  //memory connections
  wire we;
  assign cur_flush_valid = we;
  wire [7:0] addr;
  wire [7:0] di;
  wire [7:0] do;

  //this is used for the post−run querying
  reg [7:0] query_addr;

  assign addr = en ? cur_flush_addr : query_addr;

  bram memory(.clk(clk), .do(do), .di(di), .addr(addr), .en(1), .rst(rst), .we(w
e));

  committer commit(.clk(clk), .en(en), .addr_in(flush_addrs), .valid_in(flush_va
lids),
    .data_in(flush_datas), .addr_out(cur_flush_addr),
    .data_out(di), .write(we));

//then, fill them up with data
  task issue_write0;
  input [7:0] addr;
  input [7:0] data;
  begin
    #4 node[0].cpu_addr = addr;
    node[0].cpu_data_in = data;
    node[0].cpu_write = 0;
    #4 node[0].cpu_write = 1;
    #4 node[0].cpu_write = 0;
  end
  endtask
  task issue_write1;
  input [7:0] addr;
  input [7:0] data;
  begin
    #4 node[1].cpu_addr = addr;
    node[1].cpu_data_in = data;
    node[1].cpu_write = 0;
    #4 node[1].cpu_write = 1;
    #4 node[1].cpu_write = 0;
  end
  endtask
  task issue_write2;
  input [7:0] addr;
  input [7:0] data;
  begin
    #4 node[2].cpu_addr = addr;
    node[2].cpu_data_in = data;
    node[2].cpu_write = 0;
    #4 node[2].cpu_write = 1;
    #4 node[2].cpu_write = 0;
  end
  endtask
  task issue_write3;
  input [7:0] addr;
  input [7:0] data;
  begin
    #4 node[3].cpu_addr = addr;
    node[3].cpu_data_in = data;
    node[3].cpu_write = 0;
    #4 node[3].cpu_write = 1;
    #4 node[3].cpu_write = 0;
  end
  endtask

  task chk_addr;
  input [7:0] addr;
  input [7:0] data;
  begin
    #4 query_addr = addr;
    #4 if (do != data) begin
      $display("expected data at [%d] to be %d, but it was %d",addr,data,do);
      $finish;
    end
  end
  endtask

//next, run the committer for several clocks on a BRAM

  always #2 clk = ~clk;

  initial begin
    $display("(II) starting committer testbench");
    clk = 0;
    rst = 1;
    en = 0;
    #4 rst = 0;
    issue_write0(0,13);
    issue_write0(1,14);
    issue_write0(2,15);
    issue_write0(3,16);

    issue_write1(5,199);
    issue_write1(7,201);

    issue_write2(1,42);
    issue_write2(3,43);
    issue_write2(4,44);
    issue_write2(5,45);
    issue_write2(6,46);

    issue_write3(6,107);
    issue_write3(7,108);
    issue_write3(8,192);

    #4 en = 1;
    #100
    en = 0;
    #4

//finally, verify that we got expected results
   chk_addr(0,13);
   chk_addr(1,14);
   chk_addr(2,15);
   chk_addr(3,16);
   chk_addr(4,44);
   chk_addr(5,199);
   chk_addr(6,46);
   chk_addr(7,201);
   chk_addr(8,192);
   if (!node[0].sys_fin) begin
     $display("node 0 didn’t finish flushing");
     $finish;
   end
   if (!node[1].sys_fin) begin
     $display("node 1 didn’t finish flushing");
     $finish;
   end
   if (!node[2].sys_fin) begin
     $display("node 2 didn’t finish flushing");
     $finish;
   end
   if (!node[3].sys_fin) begin
     $display("node 3 didn’t finish flushing");
     $finish;
   end

   #10 $display("(II) finished committer testbench successfully");
  end
endmodule

module test_read_multiplexer;
  localparam NODES = 4;

  //these are the wires that connect the arbiters to the multiplexer
  wire [8*NODES−1:0] req_addrs;
  wire [NODES−1:0] syn;
  wire [NODES−1:0] ack;
  wire [NODES−1:0] cpu_halts;
  wire [7:0] req_addr;

  //this is upkeep stuff (clocks, state machine)
  reg clk, rst, en;
  reg [7:0] prog_addr;

  //these wires go to the BRAM
  wire [7:0] addr;
  reg [7:0] prog_data;
  wire [7:0] do;
  reg we;

  genvar i;
  generate for (i = 0; i < NODES; i = i + 1) begin: node
    reg [7:0] cpu_addr, cpu_data_in;
    wire [7:0] cpu_data_out;
    reg cpu_read, cpu_write;
    wire sys_fin;

    initial cpu_read = 0;
    initial cpu_addr = 255;
    initial cpu_data_in = 0;

    arbiter arb(.clk(clk), .rst(rst), .system_state(0), .system_finished(sys_fin
),
      .cpu_addr(cpu_addr), .cpu_data_out(cpu_data_out), .cpu_data_in(cpu_data_in
),
      .cpu_write(cpu_write), .cpu_read(cpu_read), .count_reached(0),
      .cpu_halt(cpu_halts[i]), .flush_addr_out(),
      .flush_out_valid(), .flush_data_out(),
      .flush_addr_in(), .flush_in_valid(), .out_of_memory(),
      .mainmem_addr(req_addrs[8*(i+1)−1:8*i]), .mainmem_req(syn[i]),
      .mainmem_ack(ack[i]), .mainmem_data(do));
  end
  endgenerate

  assign addr = en ? prog_addr : req_addr;

  read_multiplexer rm(.clk(clk), .rst(rst), .addr_in(req_addrs), .syn_in(syn), .
ack_out(ack), .addr_out(req_addr));

  bram memory(.clk(clk), .do(do), .di(prog_data), .addr(addr), .en(1), .rst(rst)
, .we(we));

  always #2 clk = ~clk;

  task load_mem;
    input [7:0] addr;
    input [7:0] data;
    begin
      prog_data = data;
      prog_addr = addr;
      #4 we = 1;
      #4 we = 0;
      #4;
    end
  endtask

  task issue_write0;
  input [7:0] addr;
  input [7:0] data;
  begin
    #4 node[0].cpu_addr = addr;
    node[0].cpu_data_in = data;
    node[0].cpu_write = 0;
    #4 node[0].cpu_write = 1;
    #4 node[0].cpu_write = 0;
  end
  endtask
  task issue_write1;
  input [7:0] addr;
  input [7:0] data;
  begin
    #4 node[1].cpu_addr = addr;
    node[1].cpu_data_in = data;
    node[1].cpu_write = 0;
    #4 node[1].cpu_write = 1;
    #4 node[1].cpu_write = 0;
  end
  endtask
  task issue_write2;
  input [7:0] addr;
  input [7:0] data;
  begin
    #4 node[2].cpu_addr = addr;
    node[2].cpu_data_in = data;
    node[2].cpu_write = 0;
    #4 node[2].cpu_write = 1;
    #4 node[2].cpu_write = 0;
  end
  endtask
  task issue_write3;
  input [7:0] addr;
  input [7:0] data;
  begin
    #4 node[3].cpu_addr = addr;
    node[3].cpu_data_in = data;
    node[3].cpu_write = 0;
    #4 node[3].cpu_write = 1;
    #4 node[3].cpu_write = 0;
  end
  endtask

  task read0;
    input [7:0] taddr;
    input [7:0] tdata;
    begin
      #4 node[0].cpu_addr = taddr;
      node[0].cpu_read = 1;
      #4;
      wait (!node[0].arb.cpu_halt)
      if (node[0].cpu_data_out != tdata) begin
          $display("expected to get %d at addr %d, but got %d instead",
                   tdata, taddr, node[0].cpu_data_out);
```

```
        $finish;
    end
    node[0].cpu_read = 0; #2;
  end
endtask
task read1;
  input [7:0] taddr;
  input [7:0] tdata;
  begin
    #4 node[1].cpu_addr = taddr;
    node[1].cpu_read = 1;
    #4;
    wait (!node[1].arb.cpu_halt)
    if (node[1].cpu_data_out != tdata) begin
        $display("expected to get %d at addr %d, but got %d instead",
                 tdata, taddr, node[1].cpu_data_out);
        $finish;
    end
    node[1].cpu_read = 0; #2;
  end
endtask
task read2;
  input [7:0] taddr;
  input [7:0] tdata;
  begin
    #4 node[2].cpu_addr = taddr;
    node[2].cpu_read = 1;
    #4;
    wait (!node[2].arb.cpu_halt)
    if (node[2].cpu_data_out != tdata) begin
        $display("expected to get %d at addr %d, but got %d instead",
                 tdata, taddr, node[2].cpu_data_out);
        $finish;
    end
    node[2].cpu_read = 0; #2;
  end
endtask
task read3;
  input [7:0] taddr;
  input [7:0] tdata;
  begin
    #4 node[3].cpu_addr = taddr;
    node[3].cpu_read = 1;
    #4;
    wait (!node[3].arb.cpu_halt)
    if (node[3].cpu_data_out != tdata) begin
        $display("expected to get %d at addr %d, but got %d instead",
                 tdata, taddr, node[3].cpu_data_out);
        $finish;
    end
    node[3].cpu_read = 0; #2;
  end
endtask

initial begin
  $display("(II) starting read multiplexer test");
  clk = 0;
  en = 1;
  rst = 1;
```

```
    we = 0;
    #4 rst = 0;
    load_mem(0,14);
    load_mem(1,13);
    load_mem(2,12);
    load_mem(3,11);
    load_mem(4,10);
    load_mem(5,9);
    load_mem(6,8);
    load_mem(7,7);
    load_mem(8,6);
    load_mem(9,5);
    issue_write2(8,14);
    issue_write3(0,111);

    en = 0; #4; //stop loading memory, start testing

    read0(0,14);
    read0(1,13);
    read0(2,12);
    fork
      read0(3,11);
      read1(4,10);
      read2(5,9);
      read3(6,8);
    join
    fork
      read0(7,7);
      read1(8,6);
      read2(9,5);
      read3(0,111);
    join
    read2(4,10);
    fork
      read3(7,7);
      read2(8,14);
      read1(9,5);
      read0(0,14);
    join

    #10 $display("(II) finished read multiplexer test successfully");
  end
endmodule
```

```
‘default_nettype none

/*
The CAM module supports several operations. It must be able to
read data at a given address or report that address is missing.
It must be able to write data to an address regardless of whether
the address is already in the table. It also must be able to spit
out a written cell and clear that cell when requested.
*/
module storage_buffer(clk, rst, address, data_in, data_out, write, clear, read_v
alid, write_valid, stored_address, stored_data, stored_address_valid);
  parameter ADDR_WIDTH = 8;
  parameter DATA_WIDTH = 8;
  parameter LOG_CELL_COUNT = 5;
  localparam CELL_COUNT = 1<<LOG_CELL_COUNT;

  input wire clk, rst;
  input wire [ADDR_WIDTH−1:0] address;
  input wire [DATA_WIDTH−1:0] data_in;
  input wire write, clear;

  //this is the output for the queried data
  output wire [DATA_WIDTH−1:0] data_out;
  //these say whether a read or write issued now will succeed
  output wire read_valid, write_valid;

  //this is the address in some stored cell (used by commit)
  output wire [ADDR_WIDTH−1:0] stored_address;
  output wire [DATA_WIDTH−1:0] stored_data;
  //this is whether there is a cell that’s stored an address
  output wire stored_address_valid;

  //these are used by the reducers as inputs
  wire [CELL_COUNT*DATA_WIDTH−1:0] all_data;
  wire [CELL_COUNT*ADDR_WIDTH−1:0] all_addr;

  //This chunk handles identifying a matching cell
  wire [CELL_COUNT−1:0] match_flags;
  reducer #(.DATA_WIDTH(DATA_WIDTH), .INPUT_BITS(LOG_CELL_COUNT), .PROPAGATOR(1)
)
    match_propagator(.in_data(all_data), .in_valid(match_flags),
                     .out(data_out), .valid(read_valid));

  //This chunk handles identifying an empty cell
  wire [CELL_COUNT−1:0] empty_flags;
  wire [LOG_CELL_COUNT−1:0] empty_index;
  wire has_empty;
  reducer #(.DATA_WIDTH(0), .INPUT_BITS(LOG_CELL_COUNT), .PROPAGATOR(0))
    empty_propagator(.in_data(0), .in_valid(empty_flags),
                     .out(empty_index), .valid(has_empty));
  assign write_valid = has_empty;

  //this finds a cell that has something in it
  reducer #(.DATA_WIDTH(ADDR_WIDTH), .INPUT_BITS(LOG_CELL_COUNT), .PROPAGATOR(1)
)
    stored_addr_propagator(.in_data(all_addr), .in_valid(~empty_flags),
                           .out(stored_address), .valid(stored_address_valid));
  reducer #(.DATA_WIDTH(DATA_WIDTH), .INPUT_BITS(LOG_CELL_COUNT), .PROPAGATOR(1)
)
```

```
    stored_data_propagator(.in_data(all_data), .in_valid(~empty_flags),
                           .out(stored_data), .valid());

  genvar i;
  generate for (i = 0; i < CELL_COUNT; i = i + 1) begin: cells
    //write to this cell if there is no cell with that address (read_valid) and
    //the current targeted empty index is this one (and it’s valid), or if
    //this cell matches the current address
    wire write_gate = (!read_valid && empty_index == i && has_empty) || match_fl
ags[i];
    //clear this cell (reset) if it matches and it should be cleared
    wire clear_gate = match_flags[i] && clear;

    cam_cell #(.ADDR_WIDTH(ADDR_WIDTH), .DATA_WIDTH(DATA_WIDTH))
      slot(.clk(clk), .rst(rst || clear_gate),
           .address_query(address), .write(write_gate && write), .data_in(data_i
n),
           .data_out(all_data[DATA_WIDTH*(i+1)-1:DATA_WIDTH*i]),
           .stored_address(all_addr[ADDR_WIDTH*(i+1)-1:ADDR_WIDTH*i]),
           .empty(empty_flags[i]), .match(match_flags[i]));
  end
  endgenerate

endmodule

module storage_buffer_test;
  reg clk, rst, write, clear;
  reg [7:0] address;
  reg [7:0] data_in;
  wire [7:0] data_out;
  wire [7:0] stored_data_out;
  wire [7:0] addr_out;
  wire read_valid, write_valid, addr_valid;
  reg [31:0] counter;

  storage_buffer stobuf(.clk(clk), .rst(rst), .address(address), .data_in(data_i
n),
                        .data_out(data_out), .write(write), .read_valid(read_val
id),
                        .write_valid(write_valid), .clear(clear), .stored_addres
s(addr_out),
                        .stored_address_valid(addr_valid), .stored_data(stored_d
ata_out));

  always #1 clk = ~clk;

  task chk_read;
  input [7:0] addr;
  input [7:0] expect;
  input valid;
  begin
    address = addr;
    #2
    if (valid != read_valid) begin
      $display("expected read validity to be %d, got %d",valid,read_valid);
      $finish;
    end
    if (valid && expect != data_out) begin
      $display("expected to read %d, got %d",expect,data_out);
      $finish;
    end
  end
  endtask

  task chk_stored_addr;
  input valid;
  if (!valid && addr_valid) begin
    $display("expected nothing but found something: %d",addr_out);
    $finish;
  end else if (valid && !addr_valid) begin
    $display("expected something but found nothing");
    $finish;
  end else if (valid && addr_valid) begin
    #2 address = addr_out;
    #2 if (!read_valid) begin
      $display("expected that the predicted address %d would be valid, but it wasn't",address);
      $finish;
    end
  end
  endtask

  task do_write;
  input [7:0] addr;
  input [7:0] data;
  input valid;
  begin
    if (!write_valid && valid) begin
      $display("attempted to write to full storage buffer");
      $finish;
    end
    if (write_valid && !valid) begin
      $display("full storage buffer still accepting writes");
      $finish;
    end
    address = addr;
    data_in = data;
    write = 1;
    #2 write = 0; #2;
  end
  endtask

  task do_clear;
  input [7:0] addr;
  input valid;
  begin
    address = addr;
    #2
    if (!read_valid && valid) begin
      $display("expected to find cell, but missing");
      $finish;
    end
    if (!read_valid && valid) begin
      $display("expected missing cell, but found");
      $finish;
    end
    clear = 1;
    #2 clear = 0; #2;
  end
  endtask

  initial begin
    $dumpvars;
    $display("(II) starting storage buffer testbench");
    clk = 0;
    rst = 1;
    write = 0;
    clear = 0;
    address = 3;
    #2 rst = 0;

    chk_stored_addr(0);

    do_write(23,47,1);
    chk_read(22,0,0);
    chk_read(23,47,1);

    for (counter = 0; counter < 32; counter = counter + 1)
      do_write(counter, counter, 1);

    do_write(43, 53, 0);
    chk_stored_addr(1);

    for (counter = 0; counter < 32; counter = counter+1)
      chk_read(counter, counter, 1);

    do_clear(4,1);
    chk_stored_addr(1);
    chk_read(4,0,0);
    do_write(43,53,1);
    chk_read(43,53,1);
    do_clear(43,1);
    chk_stored_addr(1);
    do_clear(43,0);
    do_write(4,4,1);
    chk_read(53,0,0);
    chk_read(4,4,1);

    for (counter = 0; counter < 32; counter = counter+1)
      do_clear(counter, 1);

    if (stobuf.empty_flags != 32'hffffffff) begin
      $display("empty flags should be 32'hffffffff but is %d",stobuf.empty_flags);
      $finish;
    end
    chk_stored_addr(0);

    $display("(II) finished storage buffer testbench successfully");
  end
endmodule

/*
The CAM cell module stores a single cell of memory.

It's inputs are:
address: the address that's being accessed
write: writes the data to the address, otherwise it's a read/query
data_in: this contains the data to be written
```

```
it’s outputs are:
empty: 1 if it’s empty
match: 1 if the input is stored here
data_out: this contains the data to be read

*/
module cam_cell(clk, rst, address_query, stored_address, write, data_in, data_ou
t, empty, match);
  parameter ADDR_WIDTH = 8;
  parameter DATA_WIDTH = 8;

  input wire clk, rst;

  input wire [ADDR_WIDTH-1:0] address_query;
  input wire write;

  input wire [DATA_WIDTH-1:0] data_in;
  output reg [DATA_WIDTH-1:0] data_out;

  output wire match;
  output reg empty;

  output reg [ADDR_WIDTH-1:0] stored_address;

  always @(posedge clk) if (rst) begin
    empty <= 1;
  end else if (write) begin
    stored_address <= address_query;
    data_out <= data_in;
    empty <= 0;
  end

  assign match = !empty && address_query == stored_address;
endmodule

module cam_cell_test;
  reg clk, rst, write;
  reg [7:0] address;
  wire [7:0] stored_address;
  reg [7:0] data_in;
  wire [7:0] data_out;
  wire empty, match;

  cam_cell camcell(clk, rst, address, stored_address, write, data_in, data_out,
empty, match);

  always #1 clk = ~clk;

  initial begin
    $dumpvars;
    $display("(II) starting cam cell testbench");
    clk = 0;
    rst = 1;
    write = 0;
    address = 3;
    #2 rst = 0;

   if (!empty) begin
```

```
      $display("cell should be empty after reset");
      $finish;
    end

     data_in = 54;
     #2 write = 1;
     #2 write = 0; #2

    if (empty) begin
      $display("cell should not be empty after write");
      $finish;
    end

     if (!match) begin
       $display("cell doesn't match correctly (positive)");
       $finish;
     end

     address = 4; #2
     if (match) begin
       $display("cell doesn't match correctly (negative)1");
       $finish;
     end

     address = 3; #2
     if (!match) begin
       $display("cell doesn't match correctly (positive)");
       $finish;
     end

     if (data_out != data_in) begin
       $display("cell doesn't correctly store data");
       $finish;
     end

     address = 5;
     data_in = 22;
     #2 write = 1;
     #2 write = 0; #2

     if (empty) begin
       $display("cell should not be empty after write");
       $finish;
     end

     if (!match) begin
       $display("cell doesn't match correctly (positive)");
       $finish;
     end

     #2 rst = 1;
     #2 rst = 0; #2

     if (match) begin
       $display("cell doesn't match correctly (negative)2");
       $finish;
     end

     #10 $display("(II) finished cam cell testbench successfully");
```

```
  end
endmodule
```

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Hex display driver
//
// File:   display_16hex.v
// Date:   24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 28-Nov-06 CJT: fixed race condition between CE and RS (thanks Javier!)
//
// This verilog module drives the labkit hex dot matrix displays, and puts
// up 16 hexadecimal digits (8 bytes).  These are passed to the module
// through a 64 bit wire ("data"), asynchronously.
//
///////////////////////////////////////////////////////////////////////////////

module display_16hex (reset, clock_27mhz, data,
                disp_blank, disp_clock, disp_rs, disp_ce_b,
                disp_reset_b, disp_data_out);

   input reset, clock_27mhz;    // clock and reset (active high reset)
   input [63:0] data;           // 16 hex nibbles to display

   output disp_blank, disp_clock, disp_data_out, disp_rs, disp_ce_b,
          disp_reset_b;

   reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

   ////////////////////////////////////////////////////////////////////////////
   //
   // Display Clock
   //
   // Generate a 500kHz clock for driving the displays.
   //
   ////////////////////////////////////////////////////////////////////////////

   reg [4:0] count;
   reg [7:0] reset_count;
   reg clock;
   wire dreset;

   always @(posedge clock_27mhz)
     begin
        if (reset)
          begin
             count = 0;
             clock = 0;
          end
        else if (count == 26)
          begin
             clock = ~clock;
             count = 5'h00;
          end
        else
          count = count+1;
```

```
    end

  always @(posedge clock_27mhz)
    if (reset)
      reset_count <= 100;
    else
      reset_count <= (reset_count==0) ? 0 : reset_count-1;

  assign dreset = (reset_count != 0);

  assign disp_clock = ~clock;

  ////////////////////////////////////////////////////////////////////////////
  //
  // Display State Machine
  //
  ////////////////////////////////////////////////////////////////////////////

  reg [7:0] state;             // FSM state
  reg [9:0] dot_index;         // index to current dot being clocked out
  reg [31:0] control;          // control register
  reg [3:0] char_index;        // index of current character
  reg [39:0] dots;             // dots for a single digit
  reg [3:0] nibble;            // hex nibble of current character

  assign disp_blank = 1'b0; // low <= not blanked

  always @(posedge clock)
    if (dreset)
      begin
         state <= 0;
         dot_index <= 0;
         control <= 32'h7F7F7F7F;
      end
    else
      casex (state)
        8'h00:
          begin
             // Reset displays
             disp_data_out <= 1'b0;
             disp_rs <= 1'b0; // dot register
             disp_ce_b <= 1'b1;
             disp_reset_b <= 1'b0;
             dot_index <= 0;
             state <= state+1;
          end

        8'h01:
          begin
             // End reset
             disp_reset_b <= 1'b1;
             state <= state+1;
          end

        8'h02:
          begin
             // Initialize dot register (set all dots to zero)
             disp_ce_b <= 1'b0;
             disp_data_out <= 1'b0; // dot_index[0];
```

```
            if (dot_index == 639)
              state <= state+1;
            else
              dot_index <= dot_index+1;
         end

       8'h03:
         begin
            // Latch dot data
            disp_ce_b <= 1'b1;
            dot_index <= 31;          // re-purpose to init ctrl reg
            disp_rs <= 1'b1; // Select the control register
            state <= state+1;
         end

       8'h04:
         begin
            // Setup the control register
            disp_ce_b <= 1'b0;
            disp_data_out <= control[31];
            control <= {control[30:0], 1'b0}; // shift left
            if (dot_index == 0)
              state <= state+1;
            else
              dot_index <= dot_index-1;
         end

       8'h05:
         begin
            // Latch the control register data / dot data
            disp_ce_b <= 1'b1;
            dot_index <= 39;          // init for single char
            char_index <= 15;         // start with MS char
            state <= state+1;
            disp_rs <= 1'b0;          // Select the dot register
         end

       8'h06:
         begin
            // Load the user's dot data into the dot reg, char by char
            disp_ce_b <= 1'b0;
            disp_data_out <= dots[dot_index]; // dot data from msb
            if (dot_index == 0)
              if (char_index == 0)
                state <= 5;                   // all done, latch data
              else
              begin
                char_index <= char_index - 1; // goto next char
                dot_index <= 39;
              end
            else
              dot_index <= dot_index-1;       // else loop thru all dots
         end

     endcase

   always @ (data or char_index)
     case (char_index)
       4'h0:            nibble <= data[3:0];
```

display_16hex.v

```
      4'h1:            nibble <= data[7:4];
      4'h2:            nibble <= data[11:8];
      4'h3:            nibble <= data[15:12];
      4'h4:            nibble <= data[19:16];
      4'h5:            nibble <= data[23:20];
      4'h6:            nibble <= data[27:24];
      4'h7:            nibble <= data[31:28];
      4'h8:            nibble <= data[35:32];
      4'h9:            nibble <= data[39:36];
      4'hA:            nibble <= data[43:40];
      4'hB:            nibble <= data[47:44];
      4'hC:            nibble <= data[51:48];
      4'hD:            nibble <= data[55:52];
      4'hE:            nibble <= data[59:56];
      4'hF:            nibble <= data[63:60];
    endcase

  always @(nibble)
    case (nibble)
      4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
      4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
      4'h2: dots <= 40'b01100010_01010001_01001001_01001001_01000110;
      4'h3: dots <= 40'b00100010_01000001_01001001_01001001_00110110;
      4'h4: dots <= 40'b00011000_00010100_00010010_01111111_00010000;
      4'h5: dots <= 40'b00100111_01000101_01000101_01000101_00111001;
      4'h6: dots <= 40'b00111100_01001010_01001001_01001001_00110000;
      4'h7: dots <= 40'b00000001_01110001_00001001_00000101_00000011;
      4'h8: dots <= 40'b00110110_01001001_01001001_01001001_00110110;
      4'h9: dots <= 40'b00000110_01001001_01001001_00101001_00011110;
      4'hA: dots <= 40'b01111110_00001001_00001001_00001001_01111110;
      4'hB: dots <= 40'b01111111_01001001_01001001_01001001_00110110;
      4'hC: dots <= 40'b00111110_01000001_01000001_01000001_00100010;
      4'hD: dots <= 40'b01111111_01000001_01000001_01000001_00111110;
      4'hE: dots <= 40'b01111111_01001001_01001001_01001001_01000001;
      4'hF: dots <= 40'b01111111_00001001_00001001_00001001_00000001;
    endcase

endmodule
```

edgedetect.v

```
‘timescale 1ns / 1ps
‘default_nettype none

//this detects edges on the "in" signal
//and outputs the posedge and negedge
//on pos and neg, respectively
module edgedetect(clk,rst,in,pos,neg);
  input clk,in,rst;
  output reg pos,neg;

  reg old;

  always @(posedge clk) begin
         if (old == 0 && in == 1) pos <= 1;
         if (old == 1 && in == 0) neg <= 1;
         if (pos) pos <= 0;
         if (neg) neg <= 0;
         old <= in;
  end
endmodule
```

```
‘default_nettype none
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2006-Mar-08: Corrected default assignments to "vga_out_red", "vga_out_green"
//              and "vga_out_blue". (Was 10’h0, now 8’h0.)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////

module labkit (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
               ac97_bit_clock,

               vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
               vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
               vga_out_vsync,

               tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
               tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
               tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

               tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
               tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
               tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
               tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

               ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
               ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

               ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
               ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

               clock_feedback_out, clock_feedback_in,

               flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
               flash_reset_b, flash_sts, flash_byte_b,

               rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

               mouse_clock, mouse_data, keyboard_clock, keyboard_data,

               clock_27mhz, clock1, clock2,

               disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
               disp_reset_b, disp_data_in,

               button0, button1, button2, button3, button_enter, button_right,
               button_left, button_down, button_up,

               switch,

               led,

               user1, user2, user3, user4,

               daughtercard,

               systemace_data, systemace_address, systemace_ce_b,
               systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

               analyzer1_data, analyzer1_clock,
               analyzer2_data, analyzer2_clock,
               analyzer3_data, analyzer3_clock,
                analyzer4_data, analyzer4_clock);

   output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
   input  ac97_bit_clock, ac97_sdata_in;

   output [7:0] vga_out_red, vga_out_green, vga_out_blue;
   output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
          vga_out_hsync, vga_out_vsync;

   output [9:0] tv_out_ycrcb;
   output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
          tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
          tv_out_subcar_reset;

   input  [19:0] tv_in_ycrcb;
   input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
          tv_in_hff, tv_in_aff;
   output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
          tv_in_reset_b, tv_in_clock;
   inout  tv_in_i2c_data;

   inout  [35:0] ram0_data;
   output [18:0] ram0_address;
   output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
   output [3:0] ram0_bwe_b;

   inout  [35:0] ram1_data;
   output [18:0] ram1_address;
   output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
   output [3:0] ram1_bwe_b;

   input  clock_feedback_in;
   output clock_feedback_out;

   inout  [15:0] flash_data;
   output [23:0] flash_address;
   output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
   input  flash_sts;

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

   output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
   input  disp_data_in;
   output  disp_data_out;

   input  button0, button1, button2, button3, button_enter, button_right,
          button_left, button_down, button_up;
   input  [7:0] switch;
   output [7:0] led;

   inout [31:0] user1, user2, user3, user4;

   inout [43:0] daughtercard;
   inout  [15:0] systemace_data;
   output [6:0]  systemace_address;
   output systemace_ce_b, systemace_we_b, systemace_oe_b;
   input  systemace_irq, systemace_mpbrdy;

   output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
                 analyzer4_data;
   output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

   ////////////////////////////////////////////////////////////////////////////
   //
   // I/O Assignments
   //
   ////////////////////////////////////////////////////////////////////////////

   // Audio Input and Output
   assign beep= 1'b0;
   assign audio_reset_b = 1'b0;
   assign ac97_synch = 1'b0;
   assign ac97_sdata_out = 1'b0;
   // ac97_sdata_in is an input

   // Video Output
   assign tv_out_ycrcb = 10'h0;
   assign tv_out_reset_b = 1'b0;
   assign tv_out_clock = 1'b0;
   assign tv_out_i2c_clock = 1'b0;
   assign tv_out_i2c_data = 1'b0;
   assign tv_out_pal_ntsc = 1'b0;
   assign tv_out_hsync_b = 1'b1;
   assign tv_out_vsync_b = 1'b1;
   assign tv_out_blank_b = 1'b1;
   assign tv_out_subcar_reset = 1'b0;

   // Video Input
   assign tv_in_i2c_clock = 1'b0;
   assign tv_in_fifo_read = 1'b0;
   assign tv_in_fifo_clock = 1'b0;
   assign tv_in_iso = 1'b0;
   assign tv_in_reset_b = 1'b0;
   assign tv_in_clock = 1'b0;
   assign tv_in_i2c_data = 1'bZ;
   // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
   // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

   // SRAMs
   assign ram0_data = 36'hZ;
   assign ram0_address = 19'h0;
   assign ram0_adv_ld = 1'b0;
   assign ram0_clk = 1'b0;
   assign ram0_cen_b = 1'b1;
   assign ram0_ce_b = 1'b1;
   assign ram0_oe_b = 1'b1;
   assign ram0_we_b = 1'b1;
   assign ram0_bwe_b = 4'hF;
   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_clk = 1'b0;
   assign ram1_cen_b = 1’b1;
   assign ram1_ce_b = 1’b1;
   assign ram1_oe_b = 1’b1;
   assign ram1_we_b = 1’b1;
   assign ram1_bwe_b = 4’hF;
   assign clock_feedback_out = 1’b0;
   // clock_feedback_in is an input

   // Flash ROM
   assign flash_data = 16’hZ;
   assign flash_address = 24’h0;
   assign flash_ce_b = 1’b1;
   assign flash_oe_b = 1’b1;
   assign flash_we_b = 1’b1;
   assign flash_reset_b = 1’b0;
   assign flash_byte_b = 1’b1;
   // flash_sts is an input

   // RS−232 Interface
   assign rs232_txd = 1’b1;
   assign rs232_rts = 1’b1;
   // rs232_rxd and rs232_cts are inputs

   // PS/2 Ports
   // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

   // Buttons, Switches, and Individual LEDs
   // assign led = 8’hFF;
   // button0, button1, button2, button3, button_enter, button_right,
   // button_left, button_down, button_up, and switches are inputs

   // User I/Os
   assign user1 = 32’hZ;
   assign user2 = 32’hZ;
   assign user3 = 32’hZ;
   assign user4 = 32’hZ;

   // Daughtercard Connectors
   assign daughtercard = 44’hZ;

   // SystemACE Microprocessor Port
   assign systemace_data = 16’hZ;
   assign systemace_address = 7’h0;
   assign systemace_ce_b = 1’b1;
   assign systemace_we_b = 1’b1;
   assign systemace_oe_b = 1’b1;
   // systemace_irq and systemace_mpbrdy are inputs

   // Logic Analyzer
   assign analyzer1_data = 16’h0;
   assign analyzer1_clock = 1’b1;
   assign analyzer2_data = 16’h0;
   assign analyzer2_clock = 1’b1;
   assign analyzer3_data = 16’h0;
   assign analyzer3_clock = 1’b1;
   assign analyzer4_data = 16’h0;
   assign analyzer4_clock = 1’b1;

localparam LOG_NCPU = 4;
localparam NCPU = 1<<LOG_NCPU;
localparam DATA_WIDTH = 8;
localparam ADDR_WIDTH = 8;

   // use FPGA's digital clock manager to produce a
   // 65MHz clock (actually 64.8MHz)
   wire clock_65mhz_unbuf,clock_65mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

   // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
                   .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;

   // ENTER button is user reset
   wire rst,user_reset;
   debounce db1(.reset(power_on_reset),.clock(clock_27mhz),.noisy(~button_enter)
,.clean(user_reset));

wire [63:0] hexdata;
wire sys_clk;
wire [NCPU-1:0] cpu_clk;
wire system_state;

   //ensure reset is asserted for CPU clocks
   reg [1:0] rst_delay;
   always @(posedge sys_clk) if (user_reset | power_on_reset) begin
     rst_delay <= 2'b11;
   end else begin
     rst_delay <= {rst_delay[0], 1'b0};
   end

   assign rst = rst_delay[1];

   // generate basic XVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;

//this will contain read,incr,write,jmp
wire [17:0] instrs [9:0];
//load CPUID into reg 4
assign instrs[0] = 18'b11111_0_0100_00000000;
//load 0 into reg 0
assign instrs[1] = 18'b00000_0_0000_00000000;
//write to memaddr [reg4] from reg 0
assign instrs[2] = 18'b10110_1_0000_0100_0000;
//read from memaddr [reg4] to reg 0
assign instrs[3] = 18'b00010_1_0000_0100_0000;
//add one to reg 0
assign instrs[4] = 18'b01100_0_0000_00000001;
//write to memaddr [reg4] from reg 0
assign instrs[5] = 18'b10110_1_0000_0100_0000;
```

```
//jump, unconditional, address 3
assign instrs[6] = 18'b11010_000_0000000011;

//these are the micro's ports
wire rst_instr_count;

//these connect the arbs and sprites to the read multiplexer
  wire [(NCPU*ADDR_WIDTH)−1:0] req_addrs, sprite_req_addrs;
  wire [NCPU−1:0] syn, sprite_syn;
  wire [NCPU−1:0] ack, sprite_ack;
  wire [ADDR_WIDTH−1:0] req_addr, sprite_req_addr;
  //commit flush connections
  wire [(NCPU*ADDR_WIDTH)−1:0] flush_addrs;
  wire [(NCPU*ADDR_WIDTH)−1:0] flush_datas;
  wire [NCPU−1:0] flush_valids;
  wire [ADDR_WIDTH−1:0] cur_flush_addr;
  wire cur_flush_valid;

  wire [NCPU:0] arb_finish;

//this is all the sprites' pixels
  wire [NCPU−1:0] pixel_r, pixel_g, pixel_b;

//these wires go to the BRAM
  wire [ADDR_WIDTH−1:0] addr, addr_sprite;
  wire [DATA_WIDTH−1:0] do, di, do_sprite;
  wire we;
  assign cur_flush_valid = we;

  assign addr = system_state ? cur_flush_addr : req_addr;

ticker #(.COUNT(500_000))
  sys_clk_gen(.clk(clock_27mhz), .rst(power_on_reset), .out(arb_finish[NCPU]), .
toggle(sys_clk));

wire [7:0] sprite_pos [NCPU−1:0];

genvar i;
generate for (i = 0; i < NCPU; i = i + 1) begin: node
  wire [9:0] cpu_instr_addr;
  wire [17:0] cpu_instr = instrs[cpu_instr_addr];
  wire cpu_halt;
  reg cpu_manual_stall;
  wire cpu_clk = sys_clk & ~cpu_halt & cpu_manual_stall;
  wire [DATA_WIDTH−1:0] mem_in, mem_out;
  wire [ADDR_WIDTH−1:0] mem_addr;
  wire read, write, count_reached;

  wire switch_debounced;
  if (i < 8) begin: can_debounce
    debounce switch_db(rst, clock_27mhz, switch[7−i], switch_debounced);
  end else begin: cannot_debounce
    assign switch_debounced = 1;
  end
  always @(posedge sys_clk) cpu_manual_stall <= switch_debounced;


micro8 #(.CPUID(i)) micro(.clk(cpu_clk), .reset(rst), .interrupt(0),
  .instruction(cpu_instr),
  .in_port(mem_in), .out_port(mem_out), .port_id(mem_addr),
  .read_strobe(read), .write_strobe(write),
  .interrupt_ack(), .address(cpu_instr_addr),
  .instruction_max(16), .reset_instruction_count(rst_instr_count),
  .instruction_max_hit(count_reached));

   assign hexdata[63-(4*i):60-(4*i)] = cpu_instr_addr[4:0];

arbiter #(.LOG_STOBUF_SIZE(1))
  arb(.clk(clock_27mhz), .rst(rst), .system_state(system_state),
  .system_finished(arb_finish[i]),
  .cpu_addr(mem_addr), .cpu_data_out(mem_in), .cpu_data_in(mem_out),
  .cpu_write(write), .cpu_read(read), .cpu_halt(cpu_halt),
  .flush_addr_out(flush_addrs[(ADDR_WIDTH*(1+i))-1:ADDR_WIDTH*i]),
  .flush_addr_in(cur_flush_addr),
  .flush_data_out(flush_datas[(DATA_WIDTH*(1+i))-1:DATA_WIDTH*i]),
  .flush_out_valid(flush_valids[i]),
  .flush_in_valid(cur_flush_valid), .out_of_memory(),
  .mainmem_addr(req_addrs[(DATA_WIDTH*(1+i))-1:DATA_WIDTH*i]),
  .mainmem_req(syn[i]), .mainmem_ack(ack[i]), .mainmem_data(do),
  .count_reached(count_reached));

sprite #(.XPOS((i)*64), .MEM_ADDR(i))
  sp(.clk(clock_27mhz), .rst(rst), .syn(sprite_syn[i]), .ack(sprite_ack[i]),
  .addr_out(sprite_req_addrs[(DATA_WIDTH*(1+i))-1:DATA_WIDTH*i]),
  .data_in(do_sprite), .hcount(hcount), .vcount(vcount),
  .pixel({pixel_r[i], pixel_g[i], pixel_b[i]}),
  .sprite_pos(sprite_pos[i]));

end
endgenerate

read_multiplexer #(.LOG_INPUT_COUNT(LOG_NCPU)) cpu_multi(.clk(clock_27mhz), .rst
(rst), .addr_in(req_addrs),
  .syn_in(syn), .ack_out(ack), .addr_out(req_addr));

committer #(.LOG_INPUT_COUNT(LOG_NCPU)) commit(.clk(clock_27mhz), .en(system_sta
te), .addr_in(flush_addrs), .valid_in(flush_valids),
  .data_in(flush_datas), .addr_out(cur_flush_addr), .data_out(di), .write(we));

  read_multiplexer #(.LOG_INPUT_COUNT(LOG_NCPU)) sprite_multi(.clk(clock_27mhz),
 .rst(rst), .addr_in(sprite_req_addrs),
    .syn_in(sprite_syn), .ack_out(sprite_ack), .addr_out(sprite_req_addr));

dualport_bram memory(
        addr,
        sprite_req_addr,
        clock_27mhz,
        clock_27mhz,
        di,
        do,
        do_sprite,
        we);

  wire deterministic;
  debounce button3_db(.reset(rst), .clock(clock_27mhz), .noisy(button3),
    .clean(deterministic));

  master_state #(.NODE_COUNT(NCPU+1))
```

```
    master(.clk(clock_27mhz), .rst(rst), .state(system_state),
   .finish_in(arb_finish), .started_parallel(), .started_commit(rst_instr_count)
,
  .deterministic(deterministic));

   xvga xvga1(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
              .hsync(hsync),.vsync(vsync),.blank(blank));

  assign led[7] = ~flush_valids[0];
  assign led[6] = ~flush_valids[1];
//  assign led[5] = ~system_state;
  assign led[5] = ~flush_valids[2];
  assign led[4] = ~flush_valids[3];
  assign led[3] = ~flush_valids[4];
  assign led[2] = ~flush_valids[5];
  assign led[1] = ~flush_valids[6];
  assign led[0] = ~flush_valids[7];
//  assign led[4:0] = 5'b11_111;
/*  assign hexdata[39:32] = node[0].mem_out;
  assign hexdata[47:40] = node[1].mem_out;
  assign hexdata[55:48] = node[2].mem_out;
  assign hexdata[63:56] = node[3].mem_out;
*/
/*  assign hexdata[39:32] = sprite_pos[0];
  assign hexdata[47:40] = sprite_pos[1];
  assign hexdata[55:48] = sprite_pos[2];
  assign hexdata[63:56] = sprite_pos[3];
*/

  wire pixel[2:0];
  assign pixel[2] = |pixel_r;
  assign pixel[1] = |pixel_g;
  assign pixel[0] = |pixel_b;

  reg [2:0] rgb;
  reg b,hs,vs;
  always @(posedge clock_65mhz) begin
    hs <= hsync;
    vs <= vsync;
    b <= blank;
//    rgb <= pixel;
  end

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.
/*   assign vga_out_red = {8{rgb[2]}};
   assign vga_out_green = {8{rgb[1]}};
   assign vga_out_blue = {8{rgb[0]}};
*/
   assign vga_out_red = {8{pixel[2]}};
   assign vga_out_green = {8{pixel[1]}};
   assign vga_out_blue = {8{pixel[0]}};
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_blank_b = ~b;
   assign vga_out_pixel_clock = ~clock_65mhz;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;

  display_16hex hexdisp(rst, clock_27mhz, hexdata,
      disp_blank, disp_clock, disp_rs, disp_ce_b,
                disp_reset_b, disp_data_out);


endmodule

//video version
module debounce (input reset, clock, noisy,
                 output reg clean);

   reg [19:0] count;
   reg new;

   always @(posedge clock)
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 270_000) clean <= new;
     else count <= count+1;

endmodule

////////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
//
////////////////////////////////////////////////////////////////////////////////

module xvga(input vclock,
            output reg [10:0] hcount,    // pixel number on current line
            output reg [9:0] vcount,     // line number
            output reg vsync,hsync,blank);

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   reg hblank,vblank;
   wire hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount == 1023);
   assign hsyncon = (hcount == 1047);
   assign hsyncoff = (hcount == 1183);
   assign hreset = (hcount == 1343);

   // vertical: 806 lines total
   // display 768 lines
   wire vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount == 767);
   assign vsyncon = hreset & (vcount == 776);
   assign vsyncoff = hreset & (vcount == 782);
   assign vreset = hreset & (vcount == 805);

   // sync and blanking
   wire next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always @(posedge vclock) begin
      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

      vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
      vblank <= next_vblank;
      vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

      blank <= next_vblank | (next_hblank & ~hreset);
   end
endmodule
```

micro8test.v

```
module test_micro8;
//we’ll connect a pair of micro8s to the memory, and have them read and write
//to address 0

localparam LOG_NCPU = 2;
localparam NCPU = 1<<LOG_NCPU;
localparam DATA_WIDTH = 8;
localparam ADDR_WIDTH = 8;

reg sys_clk;
reg [6:0] sys_clk_counter;
reg rst;
wire [NCPU−1:0] cpu_clk;
wire system_state;

initial begin
  sys_clk = 0;
  rst = 1;
  sys_clk_counter = 0;
  #80 rst = 0;
  #5000; $finish;
end

always #5 sys_clk = ~sys_clk;

always @(posedge sys_clk) begin
  sys_clk_counter = sys_clk_counter + 1;
end

//this will contain read,incr,write,jmp
wire [17:0] instrs [9:0];
//load CPUID into reg 4
assign instrs[0] = 18’b11111_0_0100_00000000;
//load 0 into reg 0
assign instrs[1] = 18’b00000_0_0000_00000000;
//write to memaddr [reg4] from reg 0
assign instrs[2] = 18’b10110_1_0000_0100_0000;
//read from memaddr [reg4] to reg 0
assign instrs[3] = 18’b00010_1_0000_0100_0000;
//add one to reg 0
assign instrs[4] = 18’b01100_0_0000_00000001;
//write to memaddr [reg4] from reg 0
assign instrs[5] = 18’b10110_1_0000_0100_0000;
//jump, unconditional, address 3
assign instrs[6] = 18’b11010_000_0000000011;

genvar i;
generate for (i = 0; i < NCPU; i = i + 1) begin: node

  wire [9:0] cpu_instr_addr;
  wire [17:0] cpu_instr = instrs[cpu_instr_addr];
  wire cpu_clk = sys_clk_counter[i+1] & ~cpu_halt;
  wire cpu_halt;
  wire [DATA_WIDTH−1:0] mem_in, mem_out;
  wire [ADDR_WIDTH−1:0] mem_addr;
  wire read, write, count_reached;

micro8 #(.CPUID(i)) micro(.clk(cpu_clk), .reset(rst), .interrupt(0),
  .instruction(cpu_instr),
  .in_port(mem_in), .out_port(mem_out), .port_id(mem_addr),
  .read_strobe(read), .write_strobe(write),
  .interrupt_ack(), .address(cpu_instr_addr),
  .instruction_max(16), .reset_instruction_count(rst_instr_count),
  .instruction_max_hit(count_reached));

arbiter arb(.clk(sys_clk), .rst(rst), .system_state(system_state),
  .system_finished(arb_finish[i]),
  .cpu_addr(mem_addr), .cpu_data_out(mem_in), .cpu_data_in(mem_out),
  .cpu_write(write), .cpu_read(read), .cpu_halt(cpu_halt),
  .flush_addr_out(flush_addrs[(ADDR_WIDTH*(1+i))-1:ADDR_WIDTH*i]),
  .flush_addr_in(cur_flush_addr),
  .flush_data_out(flush_datas[(DATA_WIDTH*(1+i))-1:DATA_WIDTH*i]),
  .flush_out_valid(flush_valids[i]),
  .flush_in_valid(cur_flush_valid), .out_of_memory(),
  .mainmem_addr(req_addrs[(DATA_WIDTH*(1+i))-1:DATA_WIDTH*i]),
  .mainmem_req(syn[i]), .mainmem_ack(ack[i]), .mainmem_data(do),
  .count_reached(count_reached));

end
endgenerate
//these are the micro's ports
wire rst_instr_count;

//these connect the arbs to the read multiplexer
  wire [(NCPU*ADDR_WIDTH)-1:0] req_addrs;
  wire [NCPU-1:0] syn;
  wire [NCPU-1:0] ack;
  wire [NCPU-11:0] cpu_halts;
  wire [ADDR_WIDTH-1:0] req_addr;
  //commit flush connections
  wire [(NCPU*ADDR_WIDTH)-1:0] flush_addrs;
  wire [(NCPU*ADDR_WIDTH)-1:0] flush_datas;
  wire [NCPU-1:0] flush_valids;
  wire [ADDR_WIDTH-1:0] cur_flush_addr;
  wire cur_flush_valid;

  wire [NCPU:0] arb_finish;
  assign arb_finish[NCPU] = sys_clk_counter[3:0] == 3'b0;

//these wires go to the BRAM
  wire [ADDR_WIDTH-1:0] addr;
  wire [DATA_WIDTH-1:0] do, di;
  wire we;
  assign cur_flush_valid = we;

  assign addr = system_state ? cur_flush_addr : req_addr;

read_multiplexer #(.LOG_INPUT_COUNT(LOG_NCPU)) rd_multi(.clk(sys_clk), .rst(rst)
, .addr_in(req_addrs),
  .syn_in(syn), .ack_out(ack), .addr_out(req_addr));

committer #(.LOG_INPUT_COUNT(LOG_NCPU)) commit(.clk(sys_clk), .en(system_state),
 .addr_in(flush_addrs), .valid_in(flush_valids),
  .data_in(flush_datas), .addr_out(cur_flush_addr), .data_out(di), .write(we));

bram memory(.clk(sys_clk), .do(do), .di(di), .addr(addr), .en(1), .rst(rst), .we
(we));
```

micro8test.v

```
master_state #(.NODE_COUNT(NCPU+1)) master(.clk(sys_clk), .rst(rst), .state(syst
em_state),
  .finish_in(arb_finish), .started_parallel(), .started_commit(rst_instr_count),
  .deterministic(0));
endmodule
```

micro8.v

```
‘default_nettype none

// clone of Xilinx Picoblaze(R)...
module micro8(
  input wire clk,
  input wire reset,
  input wire interrupt,
  input wire [17:0] instruction,
  input wire [7:0] in_port,
  input wire [7:0] instruction_max,
  input wire reset_instruction_count,
  output wire instruction_max_hit,
  output wire [7:0] out_port,
  output wire [7:0] port_id,
  output reg read_strobe,
  output reg write_strobe,
  output reg interrupt_ack,
  output reg [9:0] address
);

  parameter CPUID = 0;

  ////////////////////////////////////////////////////////////
  // CONTROL
  ////////////////////////////////////////////////////////////

  reg [9:0] pc;   // address of current instruction
  wire [9:0] pc_inc = pc + 1;

  // internal reset, always asserted for two cycles!
  reg ireset,xreset;
  always @(posedge clk) {ireset,xreset} <= reset ? 2’b11 : {xreset,1’b0};

  // interrupt enable
  reg interrupt_enabled,next_interrupt_enabled;
  always @(posedge clk) interrupt_enabled <= next_interrupt_enabled;
  wire irq = interrupt_enabled & interrupt;

  // current instruction
  reg [17:0] inst;
  always @(posedge clk) inst <= instruction;

  // 31 location PC stack
  (* ram_style = "distributed" *)
  reg [9:0] stack[31:0];
  reg [4:0] sp,sp_inc;   // sp points to top of stack
  reg [1:0] stsel;
  localparam st_nop = 2’b00;
  localparam st_push = 2’b01;
  localparam st_pop = 2’b10;
  always @*
    case (stsel)
      st_nop:  sp_inc = 5’h00;
      st_push: sp_inc = 5’h01;
      st_pop:  sp_inc = 5’h1F;
      default: sp_inc = 5’hxx;
    endcase
  wire [4:0] sp_next = sp + sp_inc;
  always @(posedge clk) begin
```

```
    sp <= ireset ? 5'h00 : sp_next;
    if (stsel == st_push) stack[sp_next] <= irq ? pc : pc_inc;
  end
  wire [9:0] tos = stack[sp];   // async read of top of PC stack

  // program counter
  reg [1:0] pcsel;
  localparam pc_next = 2'b00;
  localparam pc_aaa = 2'b01;
  localparam pc_tos = 2'b10;
  localparam pc_trap = 2'b11;
  always @*
    case (pcsel)
      pc_next: address = pc_inc;
      pc_aaa:  address = inst[9:0];
      pc_tos:  address = tos;
      pc_trap: address = ireset ? 10'h000 : 10'h3FF;   // reset or interrupt
      default: address = 10'hxxx;
    endcase
  always @(posedge clk) pc <= address;

  // dual-port 16-location register file
  (* ram_style = "distributed" *)
  reg [7:0] regfile[15:0];
  wire [7:0] wdata;
  reg werf;
  always @(posedge clk) if (werf) regfile[inst[11:8]] <= wdata;
  wire [7:0] sx = regfile[inst[11:8]];
  wire [7:0] sy = regfile[inst[7:4]];

  ////////////////////////////////////////////////////////////
  // INSTRUCTION COUNTER
  ////////////////////////////////////////////////////////////
  reg [7:0] instruction_count;
  assign instruction_max_hit = instruction_count == instruction_max;
  wire reset_instruction_count_internal = reset_instruction_count || ireset;

  always @(posedge clk or posedge reset_instruction_count_internal)
  if (reset_instruction_count_internal) begin
    instruction_count = 0;
  end else begin
    instruction_count = instruction_count + 1;
  end

  ////////////////////////////////////////////////////////////
  // DATAPATH/ALU
  ////////////////////////////////////////////////////////////

  reg z,c;  // condition codes

  reg cpuid_flag; // flag for whether we read from CPUID or normal

  wire [7:0] b = inst[12] ? sy : inst[7:0];  // second ALU operand

  // 64 location scratch pad
  (* ram_style = "distributed" *)
  reg [7:0] scratchpad[63:0];
  reg wesp;
  always @(posedge clk) if (wesp) scratchpad[b[5:0]] <= sx;
  wire [7:0] cpuid_data = CPUID;
  wire [7:0] spdata = !cpuid_flag ? scratchpad[b[5:0]] : cpuid_data;

  // external i/o
  assign out_port = sx;
  assign port_id = b;

  // adder -- we only want one, so make it explicitly here!
  // subtracts of B implemented as adds of ~B+1
  wire [7:0] addsub_b = inst[14] ? ~b : b;
  // cin should 0 for adds, 1 for subtracts, C condition code for ADDCY,SUBCY
  wire addsub_cin = inst[13] ? c : inst[14];
  wire [7:0] addsub;
  wire addsub_cout;
  assign {addsub_cout,addsub} = sx + addsub_b + addsub_cin;
  wire zaddsub = ~|addsub;

  // bool_op_regean ops
  reg [7:0] bool_op_reg;
  always @*
    case (inst[14:13])
      2'b00: bool_op_reg = b;           // LOAD
      2'b01: bool_op_reg = sx & b;      // AND, TEST
      2'b10: bool_op_reg = sx | b;      // OR
      2'b11: bool_op_reg = sx ^ b;      // XOR
      default: bool_op_reg = 8'hxx;
    endcase
  wire zbool_op_reg = ~|bool_op_reg;

  // rotate
  reg [7:0] rotate;
  always @* begin
    if (inst[3]) begin  // SR
      rotate[6:0] = sx[7:1];
      case (inst[2:1])
        2'b00: rotate[7] = c;       // SRA
        2'b01: rotate[7] = sx[7];   // SRX
        2'b10: rotate[7] = sx[0];   // RR
        2'b11: rotate[7] = inst[0]; // SR0,SR1
        default: rotate[7] = 1'bx;
      endcase
    end
    else begin  // SL
      rotate[7:1] = sx[6:0];
      case (inst[2:1])
        2'b00: rotate[0] = c;       // SLA
        2'b01: rotate[0] = sx[0];   // SLX
        2'b10: rotate[0] = sx[7];   // RL
        2'b11: rotate[0] = inst[0]; // SL0,SL1
        default: rotate[0] = 1'bx;
      endcase
    end
  end
  wire zrot = ~|rotate;

  // regfile write data mux
  reg [7:0] wdatax;
  reg [2:0] wsel;
  localparam w_bool = 3'b000;
  localparam w_rot = 3’b001;
  localparam w_sp = 3’b010;
  localparam w_in = 3’b011;
  localparam w_addsub = 3’b100;
  always @*
    case (wsel[1:0])
      2’b00: wdatax = bool_op_reg;
      2’b01: wdatax = rotate;
      2’b10: wdatax = spdata;
      2’b11: wdatax = in_port;
      default: wdatax = 8’hxx;
    endcase
  assign wdata = wsel[2] ? addsub : wdatax;

  // Z condition code
  reg preserved_z,next_z;
  reg [2:0] zsel;
  localparam z_nop = 3’b000;
  localparam z_addsub = 3’b001;
  localparam z_bool = 3’b010;
  localparam z_rot = 3’b011;
  localparam z_pop = 3’b100;
  localparam z_zero = 3’b101;
  always @*
    case (zsel)
      z_nop:   next_z = z;
      z_addsub: next_z = zaddsub;
      z_bool:  next_z = zbool_op_reg;
      z_rot:   next_z = zrot;
      z_pop:   next_z = preserved_z;
      z_zero:  next_z = 0;
      default: next_z = 1’bx;
    endcase
  always @(posedge clk) begin
    z <= next_z;
    if (irq) preserved_z <= z;
  end

  // C condition code
  reg preserved_c,next_c;
  reg [2:0] csel;
  localparam c_nop = 3’b000;
  localparam c_alu = 3’b001;
  localparam c_cmp = 3’b010;
  localparam c_zero = 3’b011;
  localparam c_parity = 3’b100;
  localparam c_sx0 = 3’b101;
  localparam c_sx7 = 3’b110;
  localparam c_pop = 3’b111;
  always @*
    case (csel)
      c_nop:   next_c = c;
      c_alu:   next_c = addsub_cout;   // arithmetic operations
      c_cmp:   next_c = ~addsub_cout;  // COMPARE
      c_zero:  next_c = 0;         // boolean operations
      c_parity: next_c = ^bool_op_reg;    // TEST
      c_sx0:   next_c = sx[0];     // shift/rotate right
      c_sx7:   next_c = sx[7];     // shift/rotate left
      c_pop:   next_c = preserved_c;   // interrupt
      default: next_c = 1’bx;
    endcase
  always @(posedge clk) begin
    c <= next_c;
    if (irq) preserved_c <= c;
  end

  ////////////////////////////////////////////////////////////
  // INSTRUCTION DECODE
  ////////////////////////////////////////////////////////////

  // condition test: inst[12:10]
  //   0--: unconditional
  //   100: if Z
  //   101: if ~Z
  //   110: if C
  //   111: if ~C
  wire cond = inst[12] ? inst[10]^(inst[11] ? c : z) : 1;

  always @* begin
    // default values for signals = no operation
    next_interrupt_enabled = interrupt_enabled;
    zsel = z_nop;  // assume no changes to Z,C
    csel = c_nop;
    pcsel = pc_next; // usually will execute next instruction
    stsel = st_nop;  // no changes in PC stack
    wsel = 3’bxxx;
    werf = 1’b0;     // no write to register file
    read_strobe = 1’b0;  // no read_strobe
    write_strobe = 1’b0; // no write_strobe
    interrupt_ack = 1’b0; // no write_strobe
    wesp = 1’b0;     // no write to scratch pad
    cpuid_flag = 1’b0; // no cpuid

    if (ireset) begin
      pcsel = pc_trap;
      zsel = z_zero;
      csel = c_zero;
      next_interrupt_enabled = 1’b0;
    end
    else if (irq) begin
      pcsel = pc_trap;
      next_interrupt_enabled = 1’b0;
      interrupt_ack = 1’b1;
      stsel = st_push;   // push
    end
    else if (instruction_count < instruction_max) begin
      case (inst[17:13])
      5’b00000: // LOAD
                begin
                  werf = 1’b1;
                  wsel = w_bool;
                end
      5’b00010: // INPUT
                begin
                  read_strobe = 1’b1;
                  werf = 1’b1;
                  wsel = w_in;
                end
```

```
      5'b00011: // FETCH
                begin
                  werf = 1'b1;
                  wsel = w_sp;
                end
      5'b00101: // AND
                begin
                  werf = 1'b1;
                  wsel = w_bool;
                  csel = c_zero;
                  zsel = z_bool;
                end
      5'b00110: // OR
                begin
                  werf = 1'b1;
                  wsel = w_bool;
                  csel = c_zero;
                  zsel = z_bool;
                end
      5'b00111: // XOR
                begin
                  werf = 1'b1;
                  wsel = w_bool;
                  csel = c_zero;
                  zsel = z_bool;
                end
      5'b01001: // TEST
                begin
                  csel = c_parity;
                  zsel = z_bool;
                end
      5'b01010: // COMPARE
                begin
                  csel = c_cmp;  // unsigned comparison...
                  zsel = z_addsub;
                end
      5'b01100: // ADD
                begin
                  werf = 1'b1;
                  wsel = w_addsub;
                  csel = c_alu;
                  zsel = z_addsub;
                end
      5'b01101: // ADDCY
                begin
                  werf = 1'b1;
                  wsel = w_addsub;
                  csel = c_alu;
                  zsel = z_addsub;
                end
      5'b01110: // SUB
                begin
                  werf = 1'b1;
                  wsel = w_addsub;
                  csel = c_alu;
                  zsel = z_addsub;
                end
      5'b01111: // SUBCY
                begin
```

```
                  werf = 1'b1;
                  wsel = w_addsub;
                  csel = c_alu;
                  zsel = z_addsub;
                end
      5'b10000: // SR?, SL?
                begin
                  werf = 1'b1;
                  wsel = w_rot;
                  csel = inst[3] ? c_sx0 : c_sx7;
                  zsel = z_rot;
                end
      5'b10101: // RETURN
                if (cond) begin
                  pcsel = pc_tos;
                  stsel = st_pop;
                end
      5'b10110: // OUTPUT
                write_strobe = ~clk;
      5'b10111: // STORE
                wesp = 1;
      5'b11000: // CALL
                if (cond) begin
                  pcsel = pc_aaa;
                  stsel = st_push;
                end
      5'b11010: // JUMP
                if (cond) pcsel = pc_aaa;
      5'b11100: // RETURNI
                begin
                  pcsel = pc_tos;
                  stsel = st_pop;   // pop
                  zsel = z_pop;
                  csel = c_pop;
                  next_interrupt_enabled = inst[0];
                end
      5'b11110: // enable/disable interrupt
                begin
                  next_interrupt_enabled = inst[0];
                end
      5'b11111: // CPUID
                begin
                  werf = 1'b1;
                  wsel = w_sp;
                  cpuid_flag = 1'b1;
                end
    endcase
    end
  end

endmodule
```

```
module rect
   #(parameter WIDTH = 48,      // default width: 64 pixels
               HEIGHT = 48,     // default height: 64 pixels
               COLOR = 3'b111)  // default color: white
   (input [10:0] x,hcount,
    input [9:0] y,vcount,
    output reg [2:0] pixel);

   always @ * begin
      if ((hcount >= x && hcount < (x+WIDTH)) &&
         (vcount < y))
//       (vcount >= y && vcount < (y+HEIGHT)))
        pixel = COLOR;
      else pixel = 0;
   end
endmodule

module sprite(clk, rst, syn, ack, addr_out, data_in, hcount, vcount, pixel, spri
te_pos);
  //where this one is placed on the screen
  parameter XPOS = 0;
  //the address of bram this reads from
  parameter MEM_ADDR = 0;
  parameter DATA_WIDTH = 8;
  parameter ADDR_WIDTH = 8;
  parameter DIVIDER_LOG = 16;

  input wire clk, ack, rst;
  output reg syn;
  input wire [10:0] hcount;
  input wire [9:0] vcount;
  input wire [DATA_WIDTH−1:0] data_in;
  output wire [ADDR_WIDTH−1:0] addr_out;
  assign addr_out = MEM_ADDR;
  output wire [2:0] pixel;
  output wire [7:0] sprite_pos;

  reg [9:0] ypos;
  wire [10:0] xpos_const = XPOS;
  reg [DIVIDER_LOG−1:0] divider;

  assign sprite_pos = ypos[7:0];

  //pad out data_in when assigning to ypos
  wire [9−DATA_WIDTH:0] padding;
  assign padding = 0;

  rect block(.x(xpos_const), .y(ypos), .hcount(hcount), .vcount(vcount), .pixel(
pixel));

  always @(posedge clk) if (rst) begin
    divider <= 0;
    syn <= 0;
  end else begin
    divider <= divider + 1;
    if (divider == 0) begin
      syn <= 1;
    end
    if (ack) begin
```

```
      ypos <= {padding,data_in};
      syn <= 0;
    end
  end
endmodule

module test_sprite;
  reg clk, rst, ack;
  wire syn;

  sprite #(.DIVIDER_LOG(4)) blah(.clk(clk), .rst(rst), .syn(syn), .ack(ack),
    .data_in(44), .hcount(), .vcount(), .pixel());

  always #2 clk = ~clk;

  initial begin
    $display("(II) begin sprite test");

    clk = 0;
    rst = 1;
    ack = 0;
    #4 rst = 0;

    wait (syn);
    #4 ack = 1;
    #4 ack = 0;

    $display("(II) end sprite test");
  end
endmodule
```

system.v

```
module master_state(clk, rst, deterministic, state, finish_in, started_parallel,
 started_commit);
  parameter NODE_COUNT = 4;

  //These describe which mode the system is in
  localparam SYSTEM_PARALLEL = 0;
  localparam SYSTEM_COMMIT = 1;

  input wire clk, rst, deterministic;
  output reg state;
  output reg started_parallel, started_commit;
  input wire [NODE_COUNT-1:0] finish_in;

  reg det_actual;

  always @(posedge clk) if (rst) begin
    state <= 0;
    started_parallel = 0;
    started_commit = 0;
    det_actual <= 0;
  end
  else begin
    if (started_parallel) det_actual <= deterministic;
    started_parallel = 0;
    started_commit = 0;
    if ((state == SYSTEM_PARALLEL) &&
         ((det_actual && (&finish_in)) || (!det_actual && (|finish_in[NODE_COUNT
-2:0]) && finish_in[NODE_COUNT-1]))) begin
        state <= SYSTEM_COMMIT;
        started_commit = 1;
    end
    else if (state == SYSTEM_COMMIT && (&finish_in)) begin
      state <= SYSTEM_PARALLEL;
      started_parallel = 1;
    end
  end
endmodule
```

ticker.v

```
‘timescale 1ns / 1ps

//this module generates a signal with period 2*COUNT/27_000_000
//toggle outputs a square wave with that period
//out stays high for one clock cycle with that period
module ticker(input clk, input rst, output out, output toggle);
  parameter COUNT=13_500_000;
  reg [24:0] count;
  reg toggler;

  edgedetect hzer(.clk(clk), .rst(rst), .in(toggler), .pos(), .neg(out));

  assign toggle = toggler;

  always @(posedge clk) begin
    if (rst) toggler <= 0;
         else if (count == COUNT) begin toggler <= ~toggler; count <= 0; end
         else count <= count + 1;
  end
endmodule
```

```
'default_nettype none

//This propagates a value if either input is valid
//input 1 has a higher priority than input 2
module reducer2to1(in1, in2, in1_valid, in2_valid, out, valid);
  //width of input in bits
  parameter DATA_WIDTH = 8;

  input wire [DATA_WIDTH-1:0] in1, in2;
  input wire in1_valid, in2_valid;
  output wire [DATA_WIDTH-1:0] out;
  output wire valid;

  assign valid = in1_valid || in2_valid;
  assign out = in1_valid ? in1 : in2;
endmodule

module reducer(in_data, in_valid, out, valid);
  //width of input in bits
  parameter DATA_WIDTH = 8;
  //number of bits needed to represent the input count
  parameter INPUT_BITS = 3;
  //this controls whether this is a reduction propagator or a priority decoder
  parameter PROPAGATOR = 1;

  localparam INPUT_COUNT = 1<<INPUT_BITS;
  localparam DATA_WIDTH_PRIVATE = PROPAGATOR == 1 ? DATA_WIDTH : INPUT_BITS;

  //input data to propagate
  input wire [INPUT_COUNT*DATA_WIDTH_PRIVATE-1:0] in_data;
  //input should be propagated or not
  input wire [INPUT_COUNT-1:0] in_valid;

  //propagation destination
  output wire [DATA_WIDTH_PRIVATE-1:0] out;
  output wire valid;

  genvar i;
  genvar j;
  generate for (i = 0; i < INPUT_BITS; i = i + 1) begin: levels
    for (j = 0; j < INPUT_COUNT/(2<<i); j = j + 1) begin: nodes
      wire valid;
      wire [DATA_WIDTH_PRIVATE-1:0] data;
      if (i == 0) begin: base_case
        wire [DATA_WIDTH_PRIVATE-1:0] in1, in2;
        //control whether we propagate values or indices
        if (PROPAGATOR) begin: propagator_case
            assign in1 = in_data[DATA_WIDTH_PRIVATE*(2*j+1)-1:DATA_WIDTH_PRIVATE
*(2*j)];
            assign in2 = in_data[DATA_WIDTH_PRIVATE*(2*j+2)-1:DATA_WIDTH_PRIVATE
*(2*j+1)];
        end else begin: priority_dec_case
            assign in1 = 2*j;
            assign in2 = 2*j+1;
        end
        reducer2to1 #(.DATA_WIDTH(DATA_WIDTH_PRIVATE))
          base_reducer(.in1_valid(in_valid[2*j]), .in2_valid(in_valid[2*j+1]),
            .in1(in1), .in2(in2), .out(data), .valid(valid));
      end else begin: reduction_case
```

```
        reducer2to1 #(.DATA_WIDTH(DATA_WIDTH_PRIVATE))
          combine_reducer(.in1_valid(levels[i-1].nodes[2*j].valid),
                          .in2_valid(levels[i-1].nodes[2*j+1].valid),
                          .in1(levels[i-1].nodes[2*j].data),
                          .in2(levels[i-1].nodes[2*j+1].data),
                          .out(data), .valid(valid));

      end
    end
  end
  assign out = levels[INPUT_BITS-1].nodes[0].data;
  assign valid = levels[INPUT_BITS-1].nodes[0].valid;
  endgenerate
endmodule

module test_reducer;
  reg [3:0] in1, in2, in3, in4, in5, in6, in7, in8;
  wire [31:0] inputs = {in8,in7,in6,in5,in4,in3,in2,in1};
  reg [7:0] in_valids;
  wire out_valid;
  wire[3:0] out_data;
  reg [31:0] i;

  reducer #(.DATA_WIDTH(4), .INPUT_BITS(3)) reducer(
            .in_data(inputs), .in_valid(in_valids),
            .out(out_data), .valid(out_valid));

  task chk;
  input [3:0] expect;
  input expect_valid;
  if (out_valid != expect_valid) begin
    $display("valid was %d but was expecting %d", out_valid, expect_valid);
    $finish;
  end else if (expect_valid && out_data != expect) begin
    $display("out was %d but was expecting %d", out_data, expect);
    $finish;
  end
  endtask

  initial begin
    $display("(II) starting reducer testbench");
    in1 = 1;
    in2 = 2;
    in3 = 3;
    in4 = 4;
    in5 = 5;
    in6 = 6;
    in7 = 7;
    in8 = 8;
    in_valids = 0;

    #5 chk(0,0);
    in_valids[3] = 1;
    #5 chk(4,1);
    in_valids[3] = 0;
    in_valids[6] = 1;
    #5 chk(7,1);
    in_valids = 8'hff;
    #5 chk(1,1);
    in_valids = 8’hf0;
    #5 chk(5,1);
    #10 $display("(II) finished reducer testbench successfully");
  end
endmodule

module priority_decoder_test;
  reg [15:0] in;
  wire [3:0] out;
  wire valid;
  reducer #(.DATA_WIDTH(0), .INPUT_BITS(4), .PROPAGATOR(0))
    test_dec2(.in_data(0), .in_valid(in), .out(out), .valid(valid));


  task chk;
  input [3:0] expect;
  input expect_valid;
  if (valid != expect_valid) begin
    $display("valid was %d but was expecting %d", valid, expect_valid);
    $finish;
  end else if (expect_valid && out != expect) begin
    $display("out was %d but was expecting %d", out, expect);
    $finish;
  end
  endtask

  initial begin
    $dumpvars;
    $display("(II) starting priority decoder testbench");
    #10 in = 16’haaf0;
    #10 chk(4,1);
    #10 in = 16’h0f00;
    #10 chk(8,1);
    #10 in = 16’h0002;
    #10 chk(1,1);
    #10 in = 16’h0000;
    #10 chk(4,0);
    #10 in = 16’h8000;
    #10 chk(15,1);
    #5 $display("(II) finished priority decoder testbench successfully");
  end
endmodule
```