Nexys 3 Tetris

By Robert Fotino and Vu Le, 2015
University of California, Los Angeles
Computer Science M152A

Introduction

For our final project, we are doing a basic implementation of the classic video game, Tetris. The game board is a 10x22 grid displayed using the VGA output of the Nexys 3. Game pieces, called "tetrominoes", are the seven unique arrangements of four connected squares, shown in Figure 1 below.

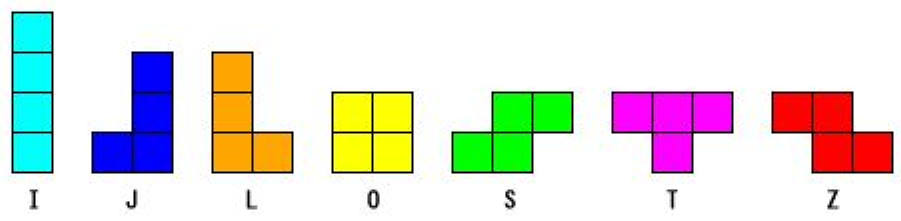

Figure 1. The seven game pieces, called tetrominoes, and the letters commonly used to describe their shapes.

When the board is in its initial state, the board is blank and it waits for the reset switch to begin. The reset switch is on the far left on the Nexys 3. The pause switch is on the far right, so that the user doesn't reset the game when they mean to pause it. When the game begins, a random tetromino is chosen as the first piece, and it falls from the top of the screen at a rate of 1 block per second. The user can use the left and right buttons on the Nexys 3 to move the piece left and right. They can also rotate the piece clockwise with the top button. Pressing the down button will move the piece down by one without waiting for the clock, while pressing the middle button will immediately drop the piece down as far as it can go. Once the current piece reaches the bottom of the board or it runs into other fallen pieces, it stays in its current position while a new piece is randomly chosen to appear at the top of the screen for the user to control. When a row of 10 blocks is filled completely filled up by pieces of fallen tetrominoes, that row disappears and the (incomplete) rows above it are shifted down by one. For each row that the user completes, their score increases by one. The score is shown on the seven segment display of the Nexys 3. The game ends when either the user hits the reset switch or the blocks get all the way to the top of the screen. When this happens, the final score and arrangement of fallen blocks are preserved until the user hits the reset switch to start a new game.

The declaration of the top level tetris module is shown in Figure 2 below:

```
module tetris (
      input wire clk_too_fast,
      input wire btn_drop,
      input wire btn_rotate,
      input wire btn_left,
      input wire btn_right,
      input wire btn_down,
      input wire sw_pause,
	input wire sw_rst,
	output wire [7:0] rgb,
	output wire hsync,
	output wire vsync,
	output wire [7:0] seg,
	output wire [3:0] an
);
```

Figure 2. The top inputs and outputs for the top level tetris module, in the file *tetris.v*.

The inputs are the master clock (called clk_too_fast) and the asynchronous signals of the 5 buttons and 2 switches. The 100 MHz master clock is reduced by 4 times to meet the timing constraints, and this 25 MHz signal is called *clk* internal to the module. The outputs are the 8-bit *rgb* value for the color of the block, the horizontal and vertical sync values *hsync* and *vsync* for the display, the 8-bit *seg* for the 7-segment display of the score, and the 4-bit *an* value that represents which digit is displayed on the board.

Design and Implementation

The top level tetris module, located in the file *tetris.v*, contains most of the game logic. It must first slow down the 100 MHz master clock to a speed of 25 MHz, in order to meet its timing constraints. When we synthesize our final design, the maximum clock frequency that will meet all of the timing constraints is ~44 MHz, and the frequency of the VGA controller's clock must be at least 25 MHz. Therefore we can slow down the master clock signal, called *clk_too_fast*, by 4 using clock division so that the *clk* signal has a frequency of 25 MHz. We then use the *clk* signal as the master clock throughout the rest of our design and the timing requirements are all met.

Next, there is a submodule called *randomizer_*, located in the file *randomizer.v*. It takes in just the *clk* signal and outputs a signal called *random* that changes once every clock cycle. This *random* signal is used as the value of the next tetromino to be chosen, when we need a random new piece at the top of the screen. In the board's initial state, it waits for the user to flip the reset switch, which starts the game by choosing a random block to start falling down the screen. This effectively "seeds" the randomizer, as the user input could come at any time, and we don't know what the value of the randomizer output will be at that time. Further randomness is introduced because the amount of time between a block being on the screen and a new one being chosen is variable based on user input. In this way we can get what appears to be a random progression of tetrominoes.

The inputs with which the user controls the board are switches and buttons, both of which can produce noisy signals that "bounce" up and down before arriving at a steady state. We want each button press or switch movement to produce exactly one "enabled" signal when it it goes to the 1 position, and then a "disabled" signal when it goes to the 0 position. For this we use a debouncer module (located in *debouncer.v*) that takes in the *clk* signal and a signal called *raw*, which is the asynchronous bouncy input, and outputs signals *enabled* and *disabled*. Internally the debouncer uses clock division to get a 200 Hz signal. When this 200 Hz signal goes high, the asynchronous input is sampled and stored in a *debounced* register. Once every *clk* cycle, the value of *debounced* is stored in a *debounced_prev* register. When *debounced* is high and *debounced_prev* is low, the *enabled* signal goes high. When *debounced* is low and *debounced_prev* is high, the *disabled* signal goes high. The *disabled* signal is used for switches, while the *enabled* signal is used for both buttons and switches.

The top level tetris module has a number of registers that are used to store the game state. The most important of these is the *mode*, a state diagram of which is shown in Figure 3 below. Other game state registers include the *cur_piece*, which indicates the type of piece the current falling tetromino is (or 0 if it is empty). There are also the *cur_pos_x* and *cur_pos_y* registers, which indicate the x and y coordinates of the top-left corner of the current tetromino. Additionally there is the 2-bit *cur_rot* register which stores the rotation of the current block (0 for 0 degrees, 1 for 90 degrees, etc). A 220-bit vector called *fallen_pieces* stores a bit for each position in the 10x22 game board, 1 if there is a fallen block in that position or 0 if it is clear. The *fallen_pieces* vector is used to test for intersection with the current piece, to test whether any lines are complete, and to add the current piece when it hits the bottom of the screen or a fallen block. The *fallen_pieces* vector along with the state of the current piece are used by the VGA display controller to output the game board. This is described in the “VGA Display Controller” section.

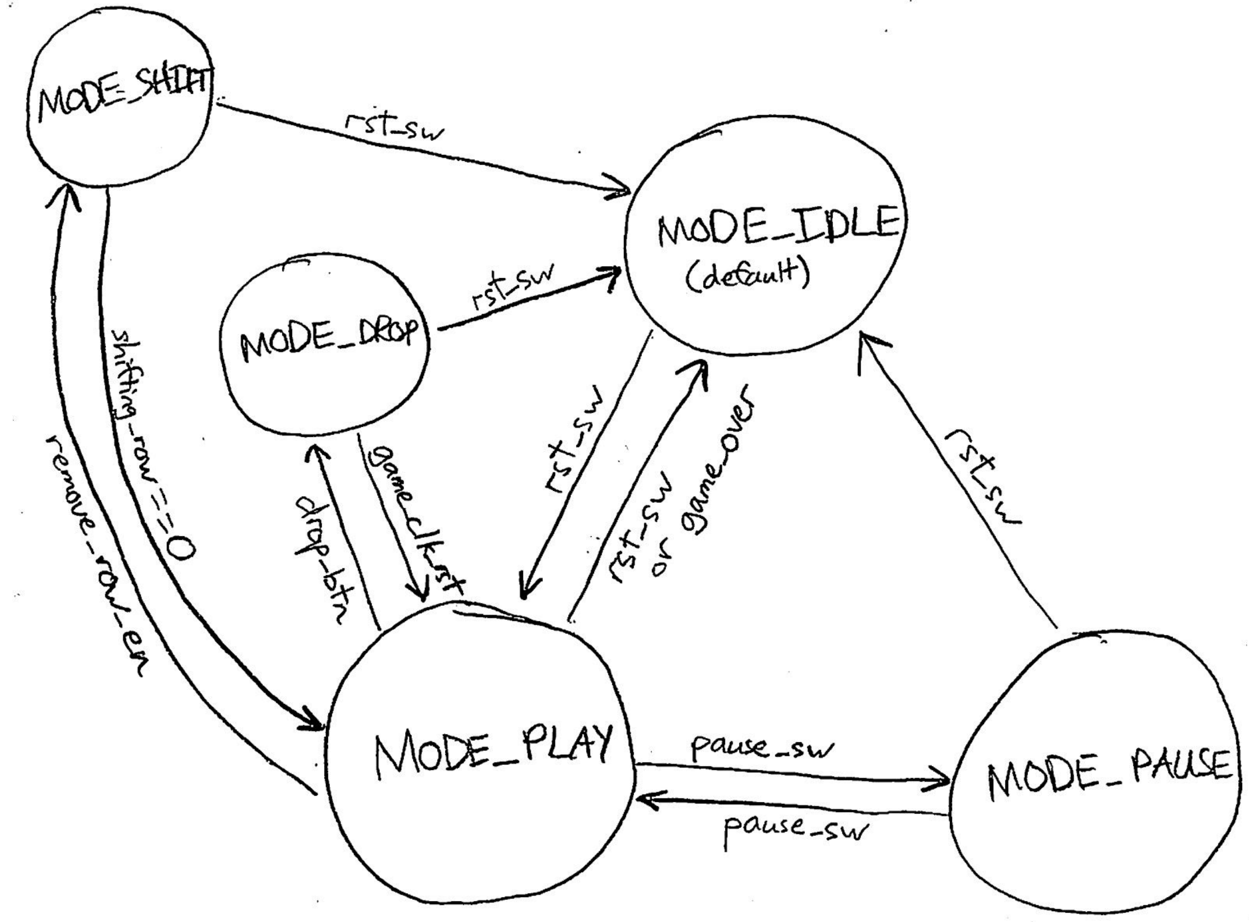

Figure 3. State diagram for the finite state machine described by the *mode* register. When blocks are actively falling down the screen, the game is in the MODE_PLAY state. When the user has hit the pause switch, the game is in the MODE_PAUSE state. When the game is waiting for the user to hit the reset switch so that it can clear the board and begin the game, it is in the MODE_IDLE state. The MODE_DROP and MODE_SHIFT states will be explained shortly.

When the game is in MODE_PLAY, the current tetromino falls at a rate of 1 Hz. This is accomplished with clock division in a *game_clock* module, found in the file *game_clock.v*. A counter counts to 25,000,000 (because the *game_clock*'s input clock is 25 MHz) and then the game clock output goes high. When a new piece needs to be chosen, the *game_clk_rst* signal is set high for one clock cycle, and the game clock counter goes back to zero, so that there is a full second in between the new block appearing at the top and falling to the bottom. Additionally, when the game is not in MODE_PLAY, the counter does not count at all. This allows for pausing in between seconds and not counting time spent in MODE_DROP or MODE_SHIFT.

Since Verilog does not allow passing two-dimensional bit vectors between modules, the *fallen_pieces* vector must be one-dimensional, and we have to do some math to find the right index based on x and y coordinates (the index is y*width + x). We use this to our advantage by having a single bit vector for the block position in one dimension so that we can index it from a flat vector like *fallen_pieces* without additional calculation. We implemented a module *calc_cur_blk* in the file *calc_cur_blk.v* that takes in a type of piece, an x and y position, and a rotation, and it gives four one-dimensional block position vectors as well as the width and height that the piece with the given parameters occupies. It does this with a series of case statements, and specific calculations depending on the geometry of each block type. We use four block positions of the current piece to add to *fallen_pieces* when the piece hits the bottom of the screen or another block. We reuse the *calc_cur_blk* module in testing for collision, described below.

When the user wants to move the current piece left, right, or down, or they want to rotate it, there is a chance that the piece will either go offscreen or it will intersect with a block in *fallen_pieces*. This is known as a “collision”, and we'd like to be able to test for it before actually moving or rotating the piece. To test for this, we send the button signals through a module called *calc_test_pos_rot*, located in the file *calc_test_pos_rot.v*. This module takes in the current x and y position, the rotation, and which button is pressed, and it uses these to determine a hypothetical next x position, y position, and rotation. For example, if the user was pressing the left button, *calc_test_pos_rot* would output *cur_pos_x* - 1, *cur_pos_y*, and *cur_rot*. The top level tetris module uses these outputs to feed into another instance of *calc_cur_blk* called *calc_test_blk*. We can then test whether the new position and rotation would intersect with any of *fallen_pieces* or whether it would go offscreen, and if so we don't move it to the new position/rotation. If we were attempting to move the block down, and there is no more space below it, we add the current piece's 4 block positions to *fallen_pieces* and choose a new piece at the top of the screen.

When the user hits the middle button, the current piece should appear to drop instantly to the bottom. To accomplish this, when the middle button is pressed the game goes into MODE_DROP. When the game is in MODE_DROP, it always moves the piece down one, once per 25 MHz clock cycle. When the piece cannot be moved down anymore because it would intersect with a *fallen_pieces* or go offscreen, a new block is chosen and the game goes back to MODE_PLAY.

When the user fills up a row in *fallen_pieces*, the game should increment the score by one and the filled row should be removed by shifting the fallen blocks above it down by one row each. In order to accomplish this, we have a *complete_row* module, located in the file *complete_row.v*. This module takes in *clk* and *fallen_pieces*, and it outputs *remove_row_y*, the y-position of the row it is examining, and *remove_row_en*, which is 1 when the row is filled and 0 otherwise. The module cycles through the rows of *fallen_pieces* with the 25 MHz clock, and when it finds a row that is filled and signals *remove_row_en*, the top level tetris module sets a register *shifting_row* equal to *remove_row_y*, increments the score, and goes into MODE_SHIFT. In MODE_SHIFT, on each 25 MHz clock cycle the row above *remove_row_y* is shifted down one row in *fallen_pieces* and *remove_row_y* is decremented. When *remove_row_y* is 0, zeroes are set to the top row and the game goes back to MODE_PLAY. With this system, multiple filled rows will take several scan and shift cycles to remove, but to the user it will seem instantaneous.

The game ends when the current block positions intersect with any of *fallen_pieces*. This means that the piece has spawned at the top of the board, but there isn't enough room. When this happens the *game_over* signal goes high, the current piece is added to *fallen_pieces*, and the game goes into MODE_IDLE to wait for the reset switch to start again. The score and the game aftermath in *fallen_pieces* are preserved visually until the user starts a new game.

Seven Segment Display Controller

The seven segment display on the Nexys 3 board shows the user what their current score is. The score starts out at zero, and increases by one for every completed row. Getting more than one row at a time does not give any sort of score multiplier. The top level tetris module handles the calculation of the score and feeds it to a module called *seg_display*, located in the file *seg_display.v*. The four digits of the score are passed separately, to ease the calculation burden of the *seg_display* module. The *seg_display* also takes in the 25 MHz *clk* as input, and outputs an 8-bit *seg* signal that contains the segment pattern to be displayed and a 4-bit *an* signal that contains which digits on which to display the segment pattern.

Since we only have one 8-bit *seg* signal, we must multiplex the digits and output one at a time, masking the others with the *an* signal. We use clock division to get a 500 Hz *seg_clk*, and every time *seg_clk* goes high we cycle forward the 2-bit *digit* register, which determines which digit we are outputting. Since we cycle through the four digits on the seven segment display at a speed of 500 Hz, much faster than the human eye can detect, to the user it looks like all four digits are being displayed at once. Since there are only four digits, the maximum score that a user can get is 9999. If they complete more rows after 9999, their score will not change.

VGA Display Controller

The VGA output on the Nexys 3 board is used to display the tetris game on the screen. The module for the VGA controller is in the file *vga_display.v*. The output resolution we use is 640x480 pixels. There is a black background, with the game board centered on the screen. The game board has a 1 pixel white border and a gray background. Fallen blocks are displayed as white, while the current movable piece is displayed in color.

The VGA signal is analog, and it does not have any way to exchange resolution information with the monitor. Furthermore, the monitor has no persistent storage, so it can only display the pixels one by one as they are received from the board. The monitor detects the resolution through the timing of the *hsync* and *vsync* signals, which stand for horizontal sync and vertical sync, respectively. There is also an 8-bit *rgb* signal, which is for the current color being output to the screen; two bits for blue, three for red, and three for green. The timing of these signals to produce an image on the VGA display is discussed below.

For a 640x480 resolution, the monitor expects the frequency of the clock to be ~25.17 MHz. Our 25 MHz clock signal is close enough for the job. We have *counter_x* and *counter_y* signals that keep track of the current position we are showing on the screen. For each horizontal line, there is a 640-cycle period of displaying actual pixel data on the screen by changing the value of *rgb* appropriately based on the size of the board, the value of *fallen_pieces*, and the current piece's state. After each line's pixel data there is a 16-cycle period waiting known as the "front porch" that precludes the *hsync* signal. Then the *hsync* signal goes low for a 96-cycle period, which tells the monitor that it needs to start displaying the new line soon. After the *hsync* signal goes back to high and there is a waiting period of 48 cycles known as the "back porch" before the next horizontal line's *rgb* data starts being output. The vertical sync operates in a similar way. For 480 lines, the *vsync* stays high and pixel data gets output. Then there is a front porch of 10 lines, followed by *vsync* going low for 2 lines. This tells the monitor that it needs to start displaying at the top of the screen soon. After a back porch of 33 lines, the next video frame starts getting output at the top left corner of the screen.

Simulation

Since the tetris game makes use of the VGA controller, it is difficult to just use the simulator and understand what is happening with the circuit. Most of our testing efforts involved synthesizing the design and putting it on the Nexys 3 board directly, so that we could make use of what we see on the VGA output for debugging. However, we encountered some difficulties in initially getting the VGA controller to work, and we were able to use the simulator to good effect in resolving our issues.

In order to test our VGA controller in the simulator, we turn the resolution way down to 8 pixels wide by 6 pixels high. The front porch for the horizontal sync is lowered to one cycle, the pulse width of the *hsync* signal is lowered to two cycles, and the back porch is lowered to one cycle. The front porch for the vertical sync is lowered to 2 lines, the pulse width of *vsync* is lowered to one line, and the back porch is lowered to 3 lines. This allows us to more easily see what is going on in the simulator, because there aren't such large gaps between *hsync* and *vsync* signal changes.

The code for the test bench is shown in Figure 4 below, and the simulation output is shown in Figure 5. The test bench instantiates a *vga_display* module and gives it just the *clk* as input, and gets just the *vsync* and *hsync* pulses as output. After 311 cycles (the time to complete the first video frame), the test bench finishes.

```
module tb();

	reg clk;
	wire hsync;
	wire vsync;

	initial begin
		clk = 0;
	end

	vga_display vga_display_(
		.clk(clk),
		.hsync(hsync),
		.vsync(vsync)
	);

	always #1 clk <= ~clk;
	always #311 $finish;

endmodule
```

Figure 4. The code for the VGA controller test bench. The clock is artificially created by making a behavioral *clk* signal that switches on and off after a waiting period of one cycle. We call $finish after 311 cycles so that that we get exactly as much output as we want.

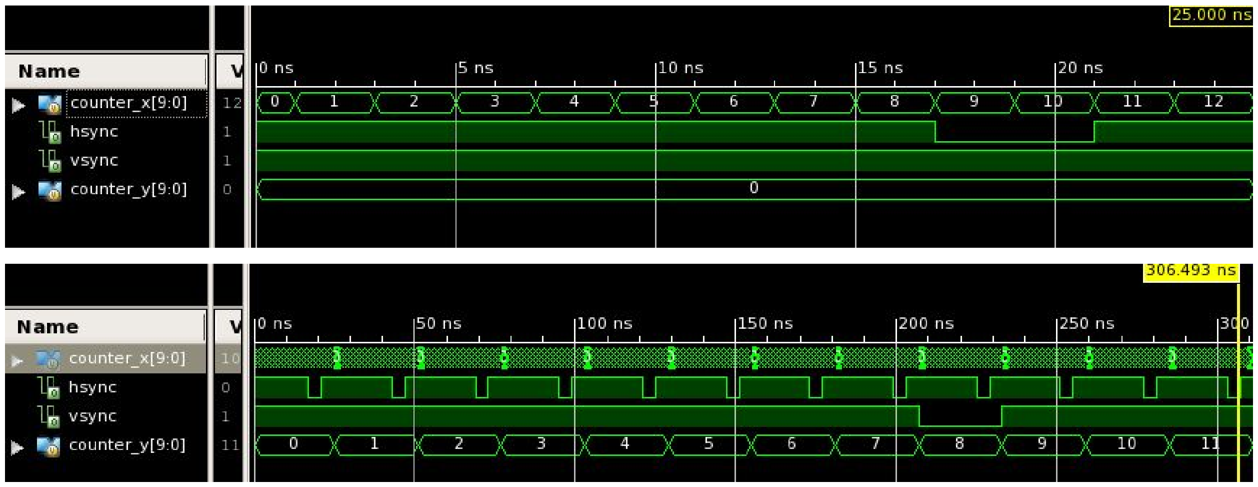

Figure 5. The waveform output for the simulation of the VGA controller. The top image shows the first 25 cycles, which is up until the end of the first horizontal line. The bottom image shows all 311 cycles, which is the entire first video frame. In the top image, *counter_x* goes from 0 to 7 and the *rgb* signal should be busy outputting pixels. Then when *counter_x* is 8, this is the front porch for *hsync*. When *counter_x* is 9 and 10, this is the *hsync* pulse. After, there is another lull for the back porch before the next line starts. A similar scenario occurs in the bottom image with the relationship between the *vsync* and *counter_y* signals.

We were able to use these waveforms to determine that our code was outputting the *hsync* and *vsync* timing signals correctly. After this simulation, we changed the resolution and pulse widths back to what they were before, in “definitions.vh”. Using a header file for all of our defined values made it easier to change the code, without searching for a bunch of magic numbers.

Conclusion

In conclusion, we were able to implement a simple tetris game in hardware on the Spartan 6 FPGA used by the Nexys 3 board. The 100 MHz master clock was too fast to meet timing constraints, so we used a clock signal of 25 MHz as main clock. The *randomizer* module cycles through types of pieces so that we can sample an effectively random piece for the new tetromino on user input. The *debouncer* module is used to reduce the noise in input signals from the buttons and switches. In the top level tetris module, we use registers to store the game modes at each given time and the game does a different function on each clock cycle depending on the current mode.

The entire game board is represented by a 220-bit vector called *fallen_pieces.* Each bit is set to 1 if there is a block at the bit’s position, and set to 0 otherwise. Each row in the *fallen_pieces* is checked on each 25MHz clock cycle. If a complete row is found, it will be removed from the game, and the above rows will be shifted down by one on each clock cycle. The tetris module uses registers to enable the MODE_SHIFT as well as to store the number of rows to be shifted down. When a new piece is created and intersects with any positive bits of the *fallen_pieces,* it signals that the game is over because there is not enough room.

The score is calculated in the top level tetris module and passed to *seg_display* module that handles the display of numbers on the 7-segment display. The *vga_display* module is used to display the game on the screen.

Difficulties Encountered

We encountered several difficulties while implementing the tetris game. At first, we tried to do too many layers of combinational logic in each clock cycle, so the amount of resources needed to synthesize the code grew to be too big for the board. We made the decision to sacrifice color in the fallen pieces, so that fallen tetrominos are just white and use one bit as opposed to three. We were using three bits per block to represent the 7 block colors plus 1 empty color. This resulted in a 10 blocks wide * 22 blocks high * 3 bits per block = 660 bit vector for *fallen_pieces*. Since there were many operations that indexed this huge bit vector in parallel, the board couldn't support the vast array of logic components required. What we could have done instead is changed the design to use a block RAM on the Nexys 3 board. It is possible to have an up to 9 Kb true dual port RAM using specialized RAM slices on the board, for no additional cost. However, this would have required changing the design quite a bit as a dual port RAM only allows a read/write operation on each port once per clock cycle, and our design used massively parallel access.

Another way we reduced the amount of slices needed on the board was by using sequential logic for MODE_DROP and MODE_SHIFT. We first tried to calculate the exact spot to drop the current tetromino all in one clock cycle, so that we could implement the middle button's functionality more easily. However, this required a lot of arbitrary parallel access to the *fallen_pieces* array; more than the board could support, with logic still needed to implement other components. After discovering this we introduced the *mode* register so that we could operate more like a finite state machine and do things sequentially instead of combinationally.

The final difficulty we encountered was that we didn't meet the timing requirements of the board. The maximum path required a maximum clock speed of ~44 MHz to meet timing requirements, and we were operating with the Nexys 3 master clock of 100 MHz. The result was that the code would synthesize, and BITGEN would even run successfully so that we could program the board. However, while running the game, we ran into certain "artifacts", where random errors would occur that the logic should not have allowed. We believe that this was the result of occasional hold time violations. To fix this, we reduce the master clock down to 25 MHz, because that's all we need. Our fastest clock requirement is for the VGA controller, which must conveniently be 25 MHz.