Кейс 1. Retro Game Console – разработка устройства вывода графики, анимации и текста с интерактивными эффектами

1. Реализация видео контроллера

Задание «»

Мной был выбран вариант реализации контроллера VGA.

Причины выбора:

- простота реализации,

- отсутствие других видео интерфейсов на макетной плате

- наличие монитора со входом VGA.

Видео интерфейс VGA, благодаря своему возрасту описан очень подробно. Я использовал информацию с сайта <http://tinyvga.com>. <http://tinyvga.com/vga-timing/640x480@60Hz>

Для своей разработки выбрал режим [640х480](http://tinyvga.com/vga-timing/640x480@60Hz). Причина выбора такого режима кроется в частоте вывода пикселей она составляет 25 МГц, что в двое меньше тактовой частоты генератора расположенного на плате.

Сам интерфейс представляет 5 сигналов: два цифровых сигнала синхронизации: - H-sync, V-sync и три аналоговых сигнала цветности R,G,B. Рассмотрим временную диаграмму синхросигналов рисунок 1.

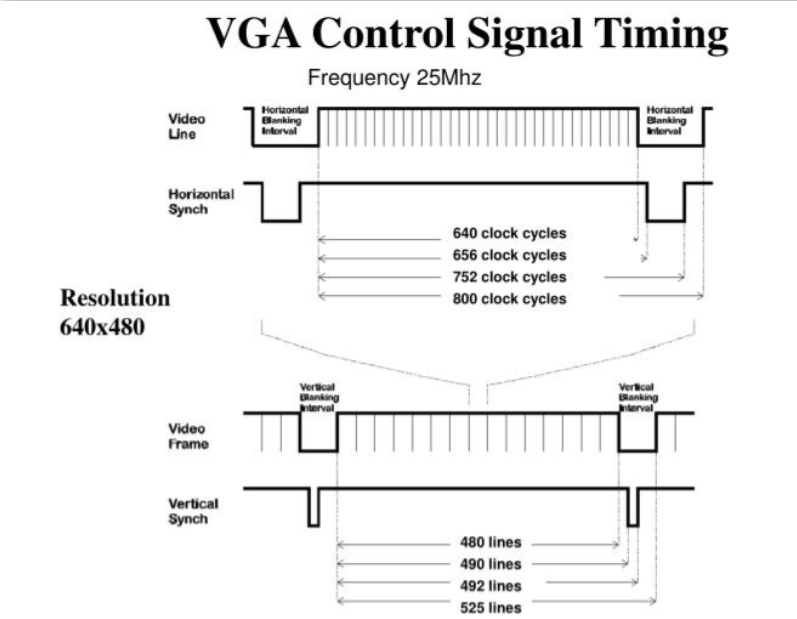


Рисунок 1. Временная диаграмма синхросигналов VGA для режима 640Х480.

Как видно из диаграммы полный кадр составляет 800 точек на 525 линий, при этом часть кадра не отображается, и начало видимой области сдвинуто относительно фронта синхросигнала. Наличие этих особенностей объясняется тем что во времена разработки интерфейса мониторы были реализованы на электронно-лучевых трубках(ЭЛТ). Для формирования изображения использовался пучок электронов (луч) который последовательно пробегал по строчкам. Для возврата луча в начало строки/экрана требовалось время, в это время изображение не выводилось.

Вернемся к диаграмме, очевидно, что нужно реализовать 2 счетчика на 800 и на 525. На основании значений счетчиков формировать синхросигналы логическими схемами. Алгоритм формирования горизонтального синхросигнала следующий:

1. Сбрасываем счетчик в «0»
2. Сигнал H-sync выставляем в «1»
3. Считаем импульсы
4. При достижении 704 сбрасываем H-sync в «0»
5. Продолжаем считать импульсы
6. При достижении 800 идем на п.1

Счетчики, используемые для формирования синхросигналов удобно использовать для идентификации координат выводимой в данный момент точки. Только нужно учитывать, что значение счетчиков сдвинуто относительно начала видимой области, что не очень удобно для формирования изображения. Для исправления этого неудобства изменим алгоритм формирования синхросигнала. Не сложно подсчитать, что от фронта H-sync до начала видимой области 48 импульсов и надо начинать считать от значения -48. Также очевидно, что для реализации счетчика до 800 требуется 10 бит. В этом случае -48 = 1024 – 48 = 976. В этом случае алгоритм формирования синхросигнала приобретает вид:

1. Загружаем в счетчик в 976
2. Сигнал H-sync выставляем в «1»
3. Считаем импульсы
4. При достижении 656 сбрасываем H-sync в «0»
5. Продолжаем считать импульсы
6. При достижении 752 идем на п.1

Используя аналогичный алгоритм будем формировать сигнал V-sync.

Текст модуля на языке systemverilog реализующий этот алгоритм в листинге 1.

Листинг 1

// модуль генерации сигналов для монитора vga 640 x 480 x 60

// модуль рассчитан на входную частоту сигнала CLK - 50 МГц

//

//`define debug

module vga

( input clk,

input rst, // сигнал нужен для моделирования при имплементации можно установить в "0"

output h\_sync,

output v\_sync,

output [9:0] x,

output [8:0] y

);

reg pix\_clk = 1'b0;

logic vl, vf;

logic [9:0] pos\_x, pos\_y;

листинг 1 продолжение

always\_ff @(posedge clk)

pix\_clk = ~pix\_clk; // формируем частоту пикселей pix\_clk = clk /2 т.е. 25 МГц

always\_ff @(posedge pix\_clk)

if (rst) begin

pos\_x <= 10'd976; // загружаем "волшебные" цифры. Это необходимо для формирования

pos\_y <= 10'd990; // паузы между фронтом синхросигнала и началом видимого поля

end

else if (~(pos\_x == 10'd752))

pos\_x <= pos\_x + 1'b1;

else begin

pos\_x <= pos\_x + 223;

if (pos\_y == 10'd492)

pos\_y <= pos\_y + 10'd499;

else pos\_y <= pos\_y + 1'b1;

end

assign h\_sync = ((pos\_x < 10'd975) & (pos\_x > 10'd656))? 1'b0 : 1'b1;

assign v\_sync = ((pos\_y < 991) & (pos\_y > 490))? 1'b0 : 1'b1;

assign x = pos\_x;

assign y = pos\_y;

endmodule

После симуляции модуля были получены диаграммы похожие на требуемые (рис. 2).

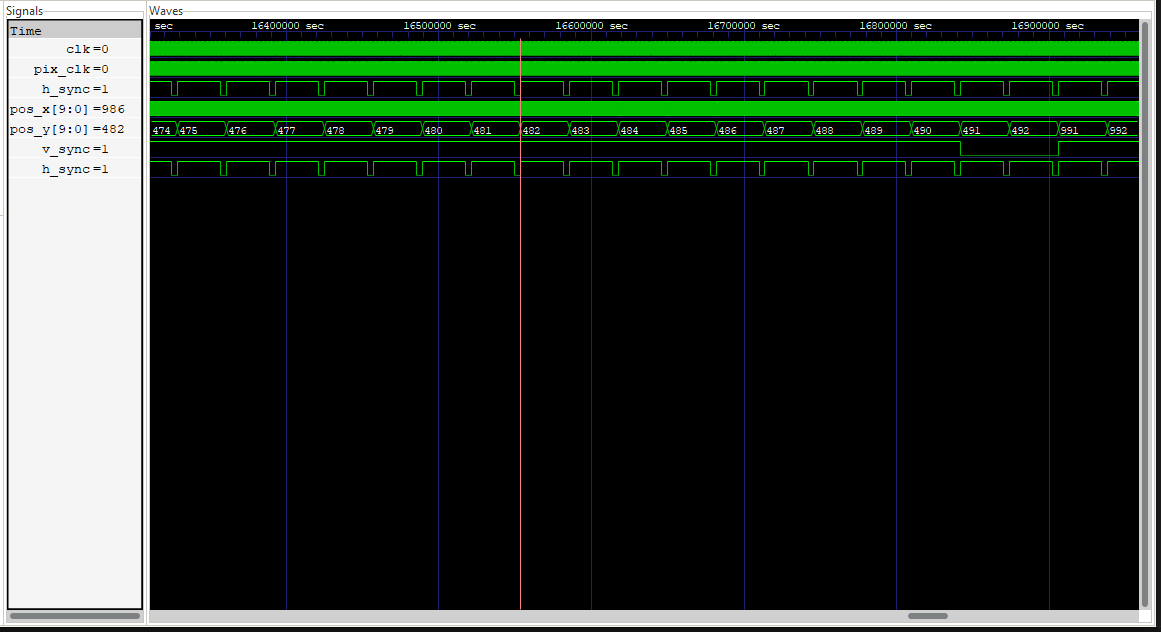


Рисунок 2 диаграммы симуляции модуля VGA.

Для проверки работоспособности модуля был создан проект в Quartus, после программирования платы полученным битстримом монитор вышел из режима энергосбережения. Что послужило первым признаком работоспособности.

Для дополнительной проверки в главный модуль проекта была добавлена строчка кода которая формирует красный квадрат.

assign r\_pin = ((x > 0) & (x < 50) & (y > 0) & (y < 50)) ? 1'b1 : 1'b0 ;

Результат работы проекта приведен на рисунке 3.

Рисунок 3 «Красный квадрат»

Дале задание предполагает разработку шрифта, вывод текста, разработку спрайта вывод спрайта на экран.

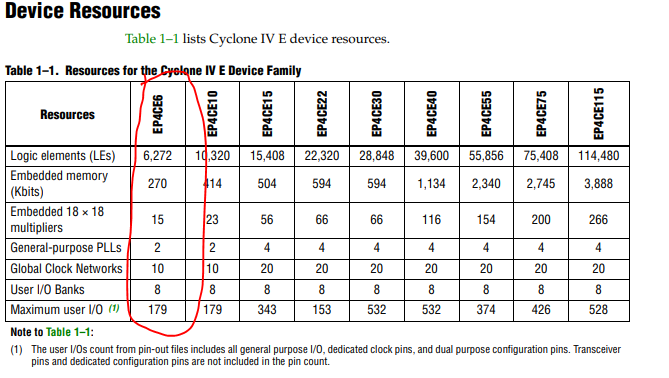
С позволения организаторов изменим порядок выполнения задания.

**4-5. Разработка пиксельной картинки, спрайта организация вывода на экран**

В настоящее время для вывода изображения на экран компьютера используется метод отображения памяти. То есть в оперативной памяти формируется изображение, а контроллер последовательно читая данные из памяти выводит изображение на экран. Определим объем памяти необходимой для реализации такого подхода.

640 х 480 = 307200 бит

Для формирования монохромного изображения необходимо 300 Kbit. Соответственно для формирования изображения из трех цветов нужно в три раза больше памяти. То есть нам нужен объем в 900 Kbit. Обратимся к справочным данным на нашу FPGA рисунок 4.



Как видно объем памяти FPGA не достаточен для формирования изображения в памяти и дальнейшего отображения на экран. По этому изображение для вывода будем формировать «на лету».

Как было показано выше закрасить прямоугольный участок экрана каким либо цветом довольно просто. Но сформировать сложное изображение из набора таких прямоугольников весьма сложно. Для формирования изображений используются небольшие картинки - спрайты. Спрайт по сути является массивом данных каждый элемент которого содержит информацию о цвете и яркости точки экрана, и задача отображения сводится к тому чтобы читать данные из памяти и своевременно выводить их на экран. Читать биты из массива задача довольно тривиальная. Реализуем вывод спрайта в модуле sprite.

Некоторые соображения по поводу реализации, для того чтобы вывести картинку нужно:

- иметь картинку,

- знать координаты вывода картинки,

- иметь представление какая часть экрана сейчас отображается.

Так как спрайтов обычно нужно больше одного и картинка у спрайта как правило не меняется в процессе игры (ну или количество изменений ограниченно), выполним передачу картинки в модуль через параметр. Через параметры также сделаем передачу размеров картинки. Для простоты реализации спрайты сделаем монохромными.

Текст модуля sprite приведен в листинге 2.

Листинг 2

module sprite #(parameter SIZE = 8,

bit [SIZE-1:0] [SIZE-1:0] MASK = {8'h18, 8'h18, 8'h18, 8'hFF, 8'hFF, 8'h18, 8'h18, 8'h18}

)

( input [9 : 0] x,

input [9 : 0] y,

input [9 : 0] pos\_x,

input [9 : 0] pos\_y,

input [2 : 0] collor,

output r,

output g,

output b

);

wire on;

wire bit\_im;

assign on = ((x > pos\_x) & (x < pos\_x + SIZE) & (y > pos\_y) & (y < pos\_y + SIZE));

assign bit\_im = (on)? MASK [pos\_y - y] [x - pos\_x]: 1'b0;

assign r = bit\_im & collor[0];

assign g = bit\_im & collor[1];

assign b = bit\_im & collor[2];

endmodule

Модуль оказался очень простым, через параметры передаем размер спрайта и его битовую маску.

Сигналы x, y содержат координаты текущей позиции отображаемой точки на экране.

Сигналы pos\_x, pos\_y задают положение верхней левой точки спрайта.

Color – задает цвет спрайта, у моей платы возможности отображения цвета не очень большие по 1 биту на цвет, по этому ширина цвета 3 бита.

В первой строке вычисляем находится ли отображаемая сейчас на экране точка в границах спрайта.

Во второй строке если сигнал on == 1 то копируем данные из битовой маски на выход модуля в соответствии со значением параметра color.

Добавим в главный модуль проекта экземпляр модуля sprite. В качестве изображения нарисуем одинокий самолетик, примерно так:

Листинг 3 инициализация спрайта.

sprite #(.SIZE(32),

.MASK({

32'b00000000\_00000001\_00000000\_00000000,

32'b00000000\_00000001\_00000000\_00000000,

32'b00000000\_00000011\_10000000\_00000000,

32'b00000000\_00000011\_10000000\_00000000,

32'b00000000\_00000110\_11000000\_00000000,

32'b00000000\_00001101\_11100000\_00000000,

32'b00000000\_00011001\_10110000\_00000000,

32'b00000000\_00110001\_10001100\_00000000,

32'b00000000\_01100001\_10000110\_00000000,

32'b00000000\_11000001\_10000011\_00000000,

32'b00000001\_10000001\_10000001\_10000000,

32'b00000011\_00000001\_10000000\_11000000,

32'b00000110\_00000001\_10000000\_01100000,

32'b00001100\_00000001\_10000000\_00110000,

32'b00011100\_00000001\_10000000\_00111000,

32'b00111000\_00000001\_10000000\_00011100,

32'b00111000\_00000001\_10000000\_00011100,

32'b00111000\_00000001\_10000000\_00011100,

32'b00111111\_11111111\_11111111\_11111100,

32'b00111111\_11111111\_11111111\_01111100,

32'b00000000\_00000001\_10000000\_00000000,

32'b00000000\_00000001\_10000000\_00000000,

32'b00000000\_00000001\_10000000\_00000000,

32'b00000000\_00000001\_10000000\_00000000,

32'b00000000\_00000001\_10000000\_00000000,

32'b00000000\_00000011\_11000000\_00000000,

32'b00000000\_00000111\_11100000\_00000000,

32'b00000000\_00001111\_11110000\_00000000,

32'b00000000\_00011111\_11111000\_00000000,

32'b00000000\_00111111\_11111100\_00000000,

32'b00000000\_00000000\_00000000\_00000000,

32'b00000000\_00000000\_00000000\_00000000

}))

plane(

.x(x),

.y(y),

.pos\_x(310),

.pos\_y(440),

.collor(3'b100),

.r(plane\_r),

.g(plane\_g),

.b(plane\_b)

);

Запустив проект видим изображение близкое к желаемому.

Рисунок 5 самолетик.

Нарисуем в добавок к нашему самолетику еще какой ни будь корабль пришельцев.

Грустно смотреть на статические картинки, тем более, что для изменения положения спрайта достаточно изменить переменные pos\_x, pos\_y. Добавим простенький цикл.

// задаем частоту просчета сцены

logic tik;

low\_clock #(.F\_CLK\_SLOW(20))

low\_clock\_i

(

.clk(clk),

.rst(1'b0),

.clk\_slow(),

.one\_pulse(tik)

);

// главный цикл геймплея

always\_ff @(posedge clk)

if(tik)

begin

// движение коробля пришельцнв

if (f\_dir\_mov) f\_x\_pos <= f\_x\_pos - 2;

else f\_x\_pos <= f\_x\_pos + 2;

if ( f\_x\_pos < 10'd1) f\_dir\_mov = 0;

if ( f\_x\_pos > 10'd608) f\_dir\_mov = 1;

// движение самолета

if (( key\_left ) & ( p\_x\_pos > 10'd1)) p\_x\_pos <= p\_x\_pos - 2;

if (( key\_right) & ( p\_x\_pos < 10'd608)) p\_x\_pos <= p\_x\_pos + 2;

end

В каждом цикле координата х корабля пришельцев меняется на 2 точки в зависимости от направления движения определяемого сигналом f\_dir\_mov. При достижении края экрана направление движения меняется.

Движение самолетика осуществлено аналогично движению коробля пришельцев за тем исключением что направление движения определяется кнопками. Частота обработки событий зависит от сигнала tik формируемого в модуле делителе low\_clock, сигнал частота формирования сигнала определяется параметром F\_CLK\_SLOW в данном случае 20 раз в секунду.

Результаты анимации можно посмотреть в файле

**2 Разработка пиксельного шрифта**

После того как мы научились формировать и выводить спрайты можно переходить к выводу букв и цифр, то есть к выводу текста. В принципе текст набрать из букв спрайтов, но это трудоемко, и что еще хуже ресурсоемко. Под каждую букву нужно резервировать память, а буквы часто повторяются и при таком подходе придется резервировать несколько (много) блоков с одинаковым содержимым. Для вывода текста используется немного другой подход, формируется один блок содержащий изображения всех необходимых символов и затем эти изображения выводятся в нужном месте. Наиболее трудоемкая часть этого процесса — это создание изображения символов. Можно сформировать битовую маску как мы формировали изображения самолетов. Но этот процесс меня не вдохновил, а особенно учитывая ограниченное время на создание проекта я рисковал остаться на уровне –«пишу строчки 0 и 1». Есть специализированный софт для создания растровых шрифтов, но там тоже нужно задать значение каждого пикселя в символе. На просторах интернета я нашел битовый [шрифт](https://github.com/YuRaNnNzZZ/PixellariCyrillic/blob/master/Pixellari_20210312.txt) и использовал его в своем проекте. <https://github.com/YuRaNnNzZZ/PixellariCyrillic/blob/master/Pixellari_20210312.txt>

Для вывода текста я немного модифицировал модуль sprait,

Листинг 3 код модуля text

module text #(parameter L = 10, // длинна строки

logic [0 : L-1] [7:0] data = {8'd35, 8'd36, 8'd37, 8'd38, 8'd39, 8'd40, 8'd41, 8'd42, 8'd43, 8'd44 } // выводимый текст "ABCDEFGHIJ"

)

( input [9 : 0] x,

input [9 : 0] y,

input [9 : 0] pos\_x,

input [9 : 0] pos\_y,

input [2 : 0] collor,

output r,

output g,

output b

);

localparam SIZE = 16; //размер шрифта

localparam logic [0:60][15:0][15:0] bitmap = { `LETTER\_30,`LETTER\_31,`LETTER\_32,`LETTER\_33,`LETTER\_34,`LETTER\_35,`LETTER\_36,`LETTER\_37,`LETTER\_38,`LETTER\_39,`LETTER\_40,`LETTER\_41,`LETTER\_42,`LETTER\_43,`LETTER\_44,`LETTER\_45,`LETTER\_46,`LETTER\_47,`LETTER\_48,`LETTER\_49,`LETTER\_50,`LETTER\_51,`LETTER\_52,`LETTER\_53,`LETTER\_54,`LETTER\_55,`LETTER\_56,`LETTER\_57,`LETTER\_58,`LETTER\_59,`LETTER\_60,`LETTER\_61,`LETTER\_62,`LETTER\_63,`LETTER\_64,`LETTER\_65,`LETTER\_66,`LETTER\_67,`LETTER\_68,`LETTER\_69,`LETTER\_70,`LETTER\_71,`LETTER\_72,`LETTER\_73,`LETTER\_74,`LETTER\_75,`LETTER\_76,`LETTER\_77,`LETTER\_78,`LETTER\_79,`LETTER\_80,`LETTER\_81,`LETTER\_82,`LETTER\_83,`LETTER\_84,`LETTER\_85,`LETTER\_86,`LETTER\_87,`LETTER\_88,`LETTER\_89,`LETTER\_90};

wire on; //

wire [6:0] pos\_char; // номер символа в строке который выводится в данный момент

wire [7:0] char\_code;

wire bit\_im;

assign on = ((x > pos\_x) & (x < pos\_x + SIZE \* L) & (y > pos\_y) & (y < pos\_y + SIZE));

assign pos\_char = (on)? ((x - pos\_x ) / SIZE) : '0; // вычисляем позицию символа которого выводим в текущий момент

assign char\_code = data[pos\_char]; // вычисляем позицию символа в кодовой странице

assign bit\_im = (on) ? bitmap[char\_code][pos\_y - y][x - pos\_x] : 1'b0;

assign r = bit\_im & collor[0];

assign g = bit\_im & collor[1];

assign b = bit\_im & collor[2];

endmodule

Из примечательного в этом модуле следует отметить то что символы задаются через директиву `define потом инициализируется массив констант bit\_map. При выводе текста определяется к какому знакоместу относится выводимая в данный момент точка потом из массива data вычитываем код символа и используя полученный код выводим изображение символа. Приведенная таблица является кусочком таблицы ASCII начиная с 30 кода. Для корректного вывода букв и цифр необходимо из кода символа ASCII вычесть 3010 т.е. код ASCII для вывода буквы A - 65 вычитаем 30 получаем 35.

Проверим работу модуля, потешим свое самолюбие выведем имя и фамилию автора. Имплементируем модуль

text #(.L(14),

.data({8'd35, 8'd46, 8'd39, 8'd58, 8'd39, 8'd59, 8'd2, 8'd40, 8'd39, 8'd38, 8'd49, 8'd52, 8'd49, 8'd56})

)

text\_1 (

.x(x),

.y(y),

.pos\_x(100),

.pos\_y(10),

.collor(3'b110),

.r(text\_r),

.g(text\_g),

.b(text\_b)

);

**6 Разработка каркасной картинки-чертежа.**

Задание: «Разработать векторную каркасную картинку-чертёж (можно в псевдо 3Д), задаваемую координатами точек в картинной плоскости дисплея».

Для визуализации каркасной картинки, задаваемой точками необходимо реализовать алгоритм растеризации отрезка. Наиболее известный алгоритм растеризации это алгоритм Брезенхема.

Суть алгоритма проста.

На первом шаге переменным x и y присваиваются значения x1 и y1. А также вычисляется коэффициент наклона . Далее на каждом шаге значение x увеличивается на 1, значение

При движении вдоль оси Х на каждом шаге коэффициент ***К*** добавляется к переменной ***error***. Если ***error*** переменная превышает 1 то значение то значение y увеличиваем на 1, а значение error уменьшаем на 1. Выполняем пока x < x2. Для того чтобы избавится от дробного коэффициента К умножим его на (x2-x1) и сравнение переменной error будем проводить со значением (x2-x1).

У алгоритма Брезенхейма есть известное ограничение – он не может отобразить линии с коэффициентом k > 1. Это ограничение обходится заменой осей. К сожалению, мне не хватило времени для реализации универсального алгоритма растеризации. Текст приведен в листинге

module line(

input clk,

input [9:0] x,

input [9:0] y,

input [9:0] x1,

input [9:0] y1,

input [9:0] x2,

input [9:0] y2,

output white

);

logic [9:0] dx, dy, new\_x, new\_y, dot\_x, dot\_y, error, new\_error;

logic start;

logic on, work;

logic clk\_2 = 0;

assign start = (x == 10'd1000 ) & (y == 10'd480); // сигнал инициализации, сбрасываем настройки в "0" после отрисовки кадра

assign dx = x2 - x1;

assign dy = y2 - y1;

assign work = ((dot\_x == x) & (dot\_y == y)) ; // сигнал разрешающий вычисления

assign on = (x < x2) & (x > x1); // сигнал разрешающий вывод изображения

always\_comb begin

new\_error = ((error + dy) < dx)? error + dy: error + dy - dx;

new\_y = ((error + dy) < dx)? dot\_y + 1: dot\_y;

end

always\_ff @(posedge clk) // костыль вызван тем что тактовая частот в два раза частоты вывода пикселей

clk\_2 <= ~clk\_2;

always\_ff @(posedge clk\_2) begin

if (start) begin

dot\_x <= x1;

dot\_y <= y1;

error <= dy;

end

if( work)

begin

error <= new\_error;

dot\_y <= new\_y;

dot\_x <= dot\_x + 1;

end

end

assign white = on & (x == dot\_x) & (y == dot\_y);

endmodule