



Video Arcade Games EvalBoard Example

Summary

Application Note
AP0127 (v2.0) February 27, 2008

This application note describes the Video Arcade Games demonstration project, which can be found in the \Examples\LiveDesign Evaluation Board\Reference Designs folder of the installation.

This document describes the design for an Arcade Graphics Platform. The VAG FPGA firmware was developed to run in an Altera Cyclone FPGA, initially targeted to the Altium NanoBoard, but ultimately intended for a single chip FPGA solution. This example has been updated to support the LiveDesign evaluation board and its peripherals.

References

- [1] VAG firmware, configured item BOC-SW-VAG
- [2] Sound board schematic, configured item BOC-CD-SND.

Overview

The Arcade Graphics Platform comprises the following major blocks:

- Embedded 8051 type (TSK51) microprocessors
- Tile and sprite graphics. VGA mode and resolution of 640 x 480 is used however due to memory constraints an active area of 512 x 480 pixels is used.
- The hardware provides 128 user-programmable 8x8-pixel tiles. Each pixel may take on one of sixteen (16) colors from a palette similar to that provided on legacy EGA controllers. In other words, the bottom three (3) bits define the color (one bit of R, G & B) and the fourth bit defines the brightness.
- Eight (8) 16x16-pixel sprites are supported, selected from a sprite map of 32. Each sprite pixel may be either transparent or one of three (3) from the available 16 colors from the palette. Sprites may be flipped about either or both of the horizontal and vertical axes. The eighth sprite has hardware collision-detection, which indicates when a non-transparent pixel of the sprite overlaps a non-transparent pixel of another sprite.
- PS/2 ports that may be connected to standard PS/2 keyboards and/or mice.
- Sound is implemented by a Sigma-Delta converter requiring just a passive network external to the FPGA for sound output.

Interface

The tile-map and sprite hardware is memory-mapped into the TSK51 external data space.

Address	Access	Description
0x0000	W	Sprite Control Registers
0x1000	R/W	Sprite Pixel Data
0x2000	R/W	Tile Map
0x3000	R/W	Tile Pixel Data
0x4000	W	Sprite Collision Reset
0x5000	W	Sound
0x6000	-	Not used
0x7000	R/W	Xdata memory space dual ported with MCU program space

The sprite and PS/2 hardware also utilize interrupts and general-purpose I/O ports on the TSK51.

Port	Bits	Description
0	6..4	PS/2 Port A control
2	7..0	PS/2 Port A data
0	7..6	PS/2 Port B control
3	7..0	PS/2 Port B data
1	2..0	Sprite Collision Detection

Palette

The video palette comprises of sixteen (16) pre-determined colors. These colors are similar to the legacy EGA controller palette. They are:

Index	Color	Index	Color
0	Black	8	Grey
1	Red	9	Bright Red
2	Green	10	Bright Green
3	Yellow	11	Bright Yellow
4	Blue	12	Bright Blue
5	Magenta	13	Bright Magenta
6	Cyan	14	Bright Cyan
7	Brown	15	White

Refer to the VGA Controller Core Reference document for color mapping.

Tile Map

The video display screen is divided into a map of 3840 tiles, arranged as 64 tiles across (X) by 60 tiles down (Y). Each map location occupies one (1) byte of Tile Map space and contains an index into a table of 128 tiles (or characters), as defined in Tile Pixel Data space.

Tiles are mapped starting from the upper left-hand corner of the display, then moving across the display along the x-axis, followed by the left-hand edge of the next line, and so on. The total visible display area comprises 3840 (0x0F00) bytes of Tile Map address space – the remaining 256 (0x0100) bytes are not used by the graphics hardware at all and may be used by the programmer as general-purpose RAM.

As there are only 128 tiles defined, the high-order bit of each byte in Tile Map space is ignored.

Tile Pixel Data

Each tile comprises 8x8 pixels. Each pixel may have any one of 16 colors in the palette.

The pixel data for each tile occupies 32 consecutive bytes in Tile Pixel Data space. The offset for each tile into this space is simply the tile number (index) multiplied by 32. The full set of 128 tiles occupies 4096 (4K) bytes of address space.

Each pixel requires 4 bits to encode 16 colors. Each byte in Tile Pixel Data space thence contains palette information for two (2) adjacent pixels. The **1st** pixel is encoded in the **low-order** nibble of the first byte. The **2nd** pixel is encoded in the **high-order** nibble of the same byte. Encoding continues with the next byte defining the **3rd** & **4th** pixels across the top row of the tile, and so on.

Sprite Pixel Data

Each sprite comprises 16x16 pixels. Each pixel may be either transparent, or have any one of three (3) colors selected for the sprite from the system palette. Each sprite has its own palette lookup table (LUT).

The pixel data for each sprite occupies 64 consecutive bytes in Sprite Pixel Data space. The offset for each sprite into this space is simply the sprite number (index) multiplied by 64. The full bank of 32 sprites occupies 2048 bytes of address space.

Each pixel requires 2 bits to encode 3 colors (plus transparent). Transparency is always value 0. Each byte in Sprite Pixel Data space thence contains palette information for four (4) adjacent pixels. The **1st** pixel is encoded in the highest-order 2 bits of the **1st** byte. The **2nd** pixel is encoded in the next-lower 2 bits, the **3rd** in the next-lower again, and finally the **4th** pixel is encoded in the lowest-order 2 bits of the byte.

Note that this bit-pixel ordering is **opposite** to that of the Tile Pixel Data.

Encoding continues along the top row of the sprite, and so on.

Sprite Control Registers

There are 8 active sprites, each controlled via a bank of sprite control registers. Each sprite has byte-wide control registers. The offset for each sprite into this space is simply the sprite number (index) multiplied by 16.

The map of registers for each sprite is as follows:

Byte Offset (hex) where n = sprite number	Bits	Description WRITE ONLY
\$n0	7..0	Sprite X register
\$n1	7..0	Sprite Y register low-order byte
\$n2	0	Sprite Y register high-order byte
\$n3	0	Sprite X flip
	1	Sprite Y flip
\$n4	7..4	Color 2 palette index
	3..0	Color 1 palette index
\$n5	3..0	Color 3 palette index
\$n6	3..0	Sprite bank index

Video Arcade Games Example

Each of the eight active sprites contains an index into the sprite bank which determines the actual pixel data for the sprite. This enables the programmer to animate a sprite simply by cycling through a number of different indexes into the sprite bank.

Although the pixel resolution of the display is in fact 512 pixels across, each sprite may only be located every 2nd pixel across the display – i.e. on even-number pixels. Thus the Sprite X register (a single byte) is in effect half the true X display coordinate of the sprite.

The high-order byte of the Sprite Y register is latched but not updated until the low-order byte is written. In other words, the programmer must generally write to the high order byte of the Y register first. This ensures that a sprite will not appear at an interim position while the two Y register bytes are being updated.

As each active sprite has its own palette look-up table (LUT), the programmer may re-use the same sprite pixel data in the bank for different sprites, which may then be differentiated by color as each sprite has a separate LUT. Alternatively, individual sprites may be animated simply by color-cycling the LUT.

A sprite can be hidden by assigning its Y beyond the visible area.

Sprite Priority Encoding

Active sprites have an implicit priority encoding, determined by their index.

Sprite 0, the 1st sprite, has highest priority and will always appear in front of other sprites. The higher the sprite index, the lower the priority. Lower-priority sprites and tiles will always appear through transparent pixels in the sprites.

Sprite Collision-Detection

The eighth sprite (sprite number 7) has hardware collision-detection capability.

Whenever a non-transparent pixel from sprite 7 overlaps (collides with) a non-transparent pixel from another sprite, the number of the colliding sprite is latched by the sprite hardware. This value is readable by the programmer on the lowest-order 3 bits of port P1 of the TSK51.

The value latched will be the last collision detected until reset by the programmer. The latch is cleared by writing any value to Sprite Collision Reset memory space.

A value of 7 read from P1.0-2 indicates no collision.

PS/2 Ports A & B

Two (2) standard PS/2 ports are provided to which may be connected either or both PS/2 keyboards and mice.

The PS/2 ports are interfaced to the TSK51 as follows:

Control	PS/2 A	PS/2 B
Interrupt	0	1
Strobe (OUT)	P0.5	P0.7
Reset (OUT)	P0.4	P0.6
Busy (IN)	P0.6	P0.7
Data (IN/OUT)	P2	P3

Port Initialization

The PS/2 ports may be initialized by setting the Strobe line HIGH, and then transitioning the Reset line from HIGH to LOW. The control lines may then be left in this state for normal operation.

Port Communications

Data may be sent to the PS/2 device by writing to the data port and then transitioning the STROBE line from LOW to HIGH. The strobe line may then be left in this state for normal operation.

Hardware Design Description

The design goals of the implementation was to fit a full graphics system, its memory and a processor, input and sound hardware inside an FPGA with no off-chip active peripherals.

Such a system could then be integrated into a larger FPGA design that required one of the following resources:

- Tile mapped graphics. Tiles can be used to display a highly complex display, fonts and animations.
- Sprite graphics. Hardware control of the movement, display, overlaying of sprites frees a processor from any memory and time intensive operation to display and move graphics. Simple register updates is all that is required. To move and color a sprite.
- Sigma-delta sound generation permits full analogue sound output with only a simple passive filter on-chip.

The implemented design is intended to show proof of concept and was scoped to use the on-chip resources as efficiently as possible while still permitting all the features to be integrated.

Simple design changes can be made to increase the performance of various aspects such as:

- Larger active pixel display area
- Larger and a greater number of tiles
- Larger and a greater number of sprites
- Higher bit depth of tiles and sprites
- Color lookup table on tile data and/or other pixel value
- Multiple collision detection of sprites
- Higher bit and frequency response of sound generator
- Larger number of sound channels or voices.

Microcontroller and Memory (mcu)

Tile Map (tileMap)

The tile mapper comprises a memory store for the tile map, a memory store for the tile data and a controller.

The size of the memory was chosen to map to little wastage to Cycle embedded memory blocks.

Dual port memory allows the processor to write to the tile map (and tile data) without any arbitration issues.

Although the tile data is writeable this could be disabled and minimized out of a design as in most cases the tile data will be set from a file at build time.

Map Controller (MapCtl)

The tilemap controller takes the current raster X and Y value as generated by the VGA controller. This raster coordinate allows the controller to calculate in which tilemap the raster is in. The tilemap data and the lower significance raster coordinates are then used to retrieve the appropriate tile data pixel values for display.

The tile map controller works on a 2-pixel basis. To simplify the design the basic unit of graphic element rather than being a pixel is two pixels. A two pixel stream connects directly to the VGA controller configured in such a mode.

As the memory is synchronous there is a clock delay to access each memory location. The double access per raster location requires a two clock delay. Due to the two bit per pixel/clock one of these clock delays is hidden. However another clock delay in the pixel path must be eliminated to allow the VGA controller to received the correct pixel data corresponding to the pixel (2-pixel) address.

To achieve a negative delay of one pixel address an early count a separate horizontal pixel counter is maintained by the controller. It is started at pixel 1 when the VGA controller comes out of blanking (thus it is one pixel early). The first pixel (pixel number 0) is retrieved during blanking and so is ready for the VGA controller on the first clock edge.

Sprite Array (sptArray)

Each of the sprites consists a register array control bits a controller and shared sprite data.

As the sprite data is shared amongst all the sprite controllers its access but be arbitrated.

Sprite Registers (sptReg)

Each sprite has a set of sprite registers to control the position of the sprite, its mirroring, colour and which bitmap to show.

Video Arcade Games Example

The registers are implemented in flip-flops to reduce embedded memory requirements and allow contact access to all the register bits.

Sprite Controller (sptCtlSch/sptCtlVHDL)

The sprite controller has been implemented initially in VHDL and then an identical implementation in schematic.

The controller operation is as follows:

- During blanking if the raster Y coordinate matches the sprite Y location then a shift registers is loaded with the first line of sprite data from the sprite bitmap memory. The offset into the sprite memory uses the sprite number to select the correct bitmap.
- The arbiter sends a strobe to the sprite controller to ensure that only the valid data from that sprite is latched into the shift register
- If the X flip bit is set then the shift register data is loaded with flipped data
- Once the raster is in view (not blanking) the raster X is checked against sprite X value. Once they match the shift register holding the sprite line data is shifted on each pixel address change (2 pixels)
- The data output from the shift register (2 bits per pixel) is converted to 4 bit per pixel data via the three color lookup sprite registers
- As the shift register is shifted out, transparent pixel code (00) are shift in, thus when the sprite line data is all shifted out the shift register continues to shift out transparent data to the end of line thus eliminating the need for a horizontal sprite pixel counter.
- On the raster X match, a row counter is also incremented to track how many lines of the sprite have been displayed.
- In the next blanking interval, if the row counter has not expired then the next row of sprite data is retrieved from the memory indexed by the row counter and sprite register number and the process repeats
- If the Y flip bit is set to 1-ones complement, then the Y row counter address is used to index in to the sprite memory to flip the Y.

Row Flipper (flipRow)

Based on an input signal the sprite row data is flipped from right to left to allow for horizontal mirroring. The bit order of each pixel is maintained.

Arbiter (sptArbiter)

The sprite bitmap memory is shared by all the sprite controllers.

The arbiter time slices (16 times) access to the memory and multiplexes the addressing from each sprite controller.

To ensure that each sprite can load its row data during blanking, the blanking interval must be at least 16 clocks wide.

Sprite Memory (sptMem2K)

The sprite memory has a 32 bit wide data out port for the sprite controller. This is a full row (16 x 2 bits per pixel) of sprite image data that can be loaded directly in one clock into the sprite controller's shift register.

An 8 bit port is available for microprocessor read/writing however in most cases a microprocessor will not be required as the bitmap is loaded at build time and thus the microprocessor port could be disabled and minimised out of an implementation.

As there are 4 memory arrays for the sprite ram (only to allow microprocessor access) the bitmap data is split into four stripes. Each memory array stores one quarter vertical stripe of sprite data.

Sprite Priority and Collision (sprPriHit)

The sprites have a priority of layering on the screen. This module implements this priority.

The background (tiles) are at the back then sprite number 7 on top, sprite 6 above and so on with sprite number 0 on top of all others.

This module also detects when sprite 7 collides with any other sprite. Comparison is done on a pixel by pixel basis. Again the priority is sprite 0 down.

Sound

The sound is a Delta Sigma DAC implementation based on Xilinx Application note XAPP154.

Only a simple RC filter is required off-chip.

A set of registers set the start and stop addresses for the sound generator to grab its samples.

EvalBoard Peripherals

VGA Controller

This peripheral is a wrapper in another symbol to expose X and Y coordinates.

As the resolution used is 512 the coordinates can be derived directly from ADDR_PIXEL. For other resolutions the X and Y would have to be generated by separate counters or better still the VGA controller expose the internal rasters counters.

PS2 Controller

Two PS2 controllers are used for various keyboard and mouse input options.

Software Design Description

The design goal of the implementation was to showcase the graphics/sound hardware and illustrate what can be achieved on the TSK51 utilizing only resources within the FPGA.

Two distinct software modules were devised to meet these ends:

- BOCMAN – a Pacman-like arcade game.
- BOCANOID – an Arkanoid (Breakout) style game.

Each of these programs were designed to run on the embedded TSK51 processor. However, in an effort to show the power and versatility of the platform, BOCMAN and BOCANOID were designed to each run on a TSK51 MCU 'side-by-side' within a single design. As a result, the tile, sprite and sound data resources were shared between the two programs.

Resources

Single MCU Project (spt8MapMCU)

BOCMAN and BOCANOID are incorporated into a single embedded project. This negates the need to re-build the FPGA image when switching target software modules and typically requires only a simple change to the (embedded) project options (C pre-processor defines) and a build of the software.

As a result, the two programs must share all tile, sprite and sound resources.

This project is built with an 8KB internal 'ROM' of which the top 4K is additionally mapped into TSK51 XDATA space. This provides some degree of flexibility for differing software requirements. Hence each project is limited to a maximum of 8KB combined ROM & XDATA space, with the constraint that XDATA is limited to 4KB.

The sound ROM is limited to 4KB – as a result the sample clock was downgraded to 4kHz, still providing about 1 second of sample data. Three (3) short sequences are contained within the ROM, used by both BOCMAN and BOCANOID – the latter playing snippets of samples to generate a number of distinct 'bleeps & bloops' during game-play.

The TSK51 has sole access to the graphics, sound and input (keyboard & mouse) resources of the hardware implementation.

Double MCU Project (spt8MapMCUX2)

This project runs BOCMAN and BOCANOID 'side-by-side' on the same display, sharing all tile resources (map & data) and sprite pixel data. Each MCU has its own set of sprites (registers). There is no sound in this project.

MCU-A has the same ROM/XDATA architecture as the single-MCU project. MCU-B has only a 4K ROM and (separate) 2K RAM.

Each MCU has a single PS/2 port. The 2nd (mouse) port has been transplanted to MCU-B with the same interface from the single-MCU version (rather than mirroring the PS/2 port interface on MCU-A) to allow software to run unchanged on either the single or X2 MCU projects.

Common Software Architecture

Keyboard

The PS/2 keyboard is connected to external interrupt 1. The PS/2 controller interrupts the TSK51 whenever a scan-code has been received from the keyboard. The scan-code is read from P3.

Due to memory constraints and modest requirements the keyboard interrupt routine (ISR) is quite simple. Break-codes and extended-key-codes are flagged until a key scan code is received.

In the event of a make-code, the ISR latches the last key-press detected in a global unsigned char. The main program may read this latch and is responsible for clearing it if required. Potential race conditions and missed key-strokes are not considered in this simple implementation.

Video Arcade Games Example

The ISR also maintains an unsigned char array of (128) scan-codes that contain the state (pressed or not-pressed) of each key. This may be used by programs that require key state information or information about multiple keys simultaneously.

Mouse

The PS/2 mouse is connected to external interrupt 0. The PS/2 controller interrupts the TSK51 whenever data is received from the mouse. The mouse data is read from P2.

Due to memory constraints and modest requirements the mouse interrupt routine (ISR) is quite simple. The ISR assumes the mouse has been initialized by the program and is in 'report' mode.

The mouse sends data to the controller in packets of 3 bytes. The ISR keeps a track of which packet it is expecting to receive next (Button, X or Y) and updates a global (3-byte) unsigned char array. New mouse data is flagged in a bit variable, *mouseAck*. Programs which access the mouse poll this flag and then read from the mouse data array.

Main Routine

Each of the modules within the shared embedded project (BOCMAN, BOCANOID) also share a common *main()* routine. This eliminates the need for duplication of platform initialization code and promotes consistent software architecture and practices.

The main routine generally configures the TSK51 processor hardware and peripherals before jumping to program-specific initialization and main-loop routines. Heavy use of macros and pre-processor directives provides clean, easy-to-follow code.

Tick Handler

Each program comprises two threads of execution – the main-line code and an interrupt-driven routine called periodically by a hardware timer. This allows programs to, for example, calibrate execution speed of certain components independently of the FPGA clock speed.

Timer0 is utilised as the generic 'game ticker'. Each program-specific header defines a base timer interval used to initialise the timer. The main routine provides the ISR which calls a program-specific *tick_handler* routine.

NANO_HW.H

A VAG-specific header file was created to be included in all VAG software modules. The header simplifies the MCU & hardware initialisation and abstracts the graphics and sound hardware resources as far as possible via the use of C pre-processor macros.

Abstracting the resources facilitates cleaner and more readable code and ensures a more consistent and easily updated access to the hardware resources continually under development and refinement – a crucial factor in the beta-testing of the BOC project.

This header also facilitates the porting of C source code to/from other platforms, for example, the benchmark PC-based platform devised within the scope of the BOC project (see below).

BOCMAN

BOCMAN showcases the tile maps, sprites and sound. The five (5) sprites are all animated, and the program also illustrates both X- and Y-hardware sprite flipping and different sprites using the same sprite data but distinct color lookup tables. The game utilizes two (2) sound sample sequences from the ROM.

BOCMAN is controlled via a keyboard attached to the 'KEYBOARD' connector on the EvalBoard.

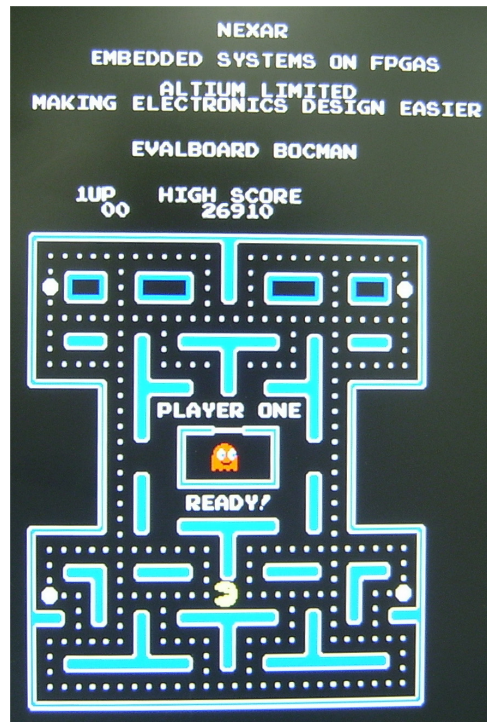
Game-Play and Implementation Notes

BOCMAN comprises a maze-like playing field through which the player must guide the 'BOCMAN' (using the arrow-keys) whilst eating the 'pellets' and avoiding the enemies. Throughout the maze are four 'pills' which when eaten by the 'BOCMAN', render the enemies susceptible themselves to being eaten by the 'BOCMAN' for a short time. The game ends when either the 'BOCMAN' is tagged by an enemy, or the player has eaten all 'pills' and 'pellets' in the maze.

The maze, pellets and pills are implemented via tiles. These are quite static in nature (i.e. they don't move at all and seldom change) and thus lend themselves to a tile-map.

The 'BOCMAN' and enemies are implemented via sprites as they, for the most part, pass unaffected over the background and/or each other. The 'BOCMAN' is the only sprite that requires collision-detection and as such utilizes the 8th sprite.

Pressing the <ESC> key on the keyboard at any time will reset the game. It literally executes a jump to address 0000 on the processor.



Data Structures

Due to the limited amount of internal data and (to a lesser extent) external RAM, as much data as possible is stored in ROM. This includes the data for the maze, movement logic and sprite animation.

The maze is displayed as derived from the maze data in ROM. A 'board' array is constructed in RAM which contains the current state of the maze (including pellets & pills) as well as providing a structure on which to calculate object movement.

Each object (player and enemies) has an associated *state* structure which comprises object coordinates (in 'board' space), tile offset (in pixels), current direction, next (potential) direction, and sprite animation cell number. This allows generic object movement logic to operate on any object in the game.

Object movement logic is table-driven from object state data and movement logic tables in ROM. This simplifies program code and reduces code space requirements.

Objects and Sprites

Each object (player & enemies) is represented by a sprite.

Each sprite in turn can be animated. This is implemented by storing a look-up table in ROM for each sprite that contains indexes into the sprite bank for each object. During each game 'tick' the sprite cell associated with each object is incremented, thus 'animating' the sprite.

Each of the enemies uses the same sequence of sprites from the sprite bank. However they each have a distinct palette which provides for 4 distinct 'characters'. When the player eats a pill, the sprite sequence for each enemy is switched, giving them an alternate appearance for a short time.

The player sprite is implemented in the 8th sprite to enable hardware collision-detection with each of the other sprites in use. Again, a (different) sequence of sprites is used for animation.

Program Flow

The main-line code does nothing but initialize the program data structures, draw the maze, and then loop endlessly polling the keyboard and inserting player movement codes into the player object data structure.

The bulk of the work is done in the game 'tick' routine.

This routine updates (moves) all the objects in the game and takes care of 'eating' objects (modifying the tile map), updating the score, playing sounds, and maintaining the state of the game.

BOCANOID

BOCANOID showcases the tile maps, sprites and sound. Whilst the three (3) sprites are not animated, it incorporates a more active tile map and offers a more colorful and 'busy' display than BOCMAN. The game utilizes a number of short extracts from the sample sequences in ROM.

Video Arcade Games Example

BOCANOID is controlled via a mouse attached to the 'MOUSE' connector on the EvalBoard.

BOCANOID also has more modest resource requirements (in all aspects) and hence was a candidate for accompanying BOCMAN in the double-MCU project.

Note that due to time & resource constraints, the game never finishes. It is also possible to get to a state whereby one or a few bricks remains but cannot be destroyed as the very simplified physics of the ball movement do not allow the ball to ever hit the brick. Pressing the <ESC> key on the keyboard at any time will reset the game. It literally executes a jump to address 0000 on the processor.

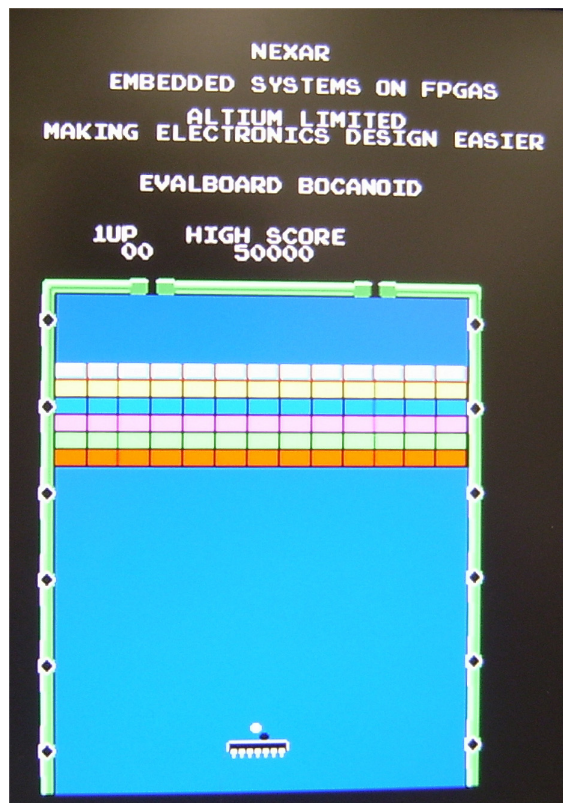
Game-Play and Implementation Notes

BOCANOID comprises an array of 'bricks' which the player strives to eliminate by hitting with a 'ball'. The ball is launched (via the left-hand mouse button) and there-after controlled via a 'bat' at the base of the playfield which may be moved left or right (with the mouse). The ball bounces off the bat, walls and any bricks it encounters. When the ball hits a brick, it will ricochet after destroying the brick. Gray (shaded) bricks require 2 hits of the ball to destroy.

The walls and bricks are implemented via tiles.

The bat and ball are implemented via sprites. The bat is itself is composed of 2 sprites displayed side-by-side, in effect creating a 32x16 pixel sprite.

There is no collision-detection utilized in BOCANOID.



Data Structures

The only significant data structure is the 'board' which is an internal representation of the playfield. Because of the dynamic nature of the data, this structure is maintained in RAM.

Objects and Sprites

The ball object has attributes including coordinates (in 'board' space) as well as X- and Y- increments (i.e. a velocity vector) and (pixel) offset within the tile map.

The bat object simply requires coordinates.

Program Flow

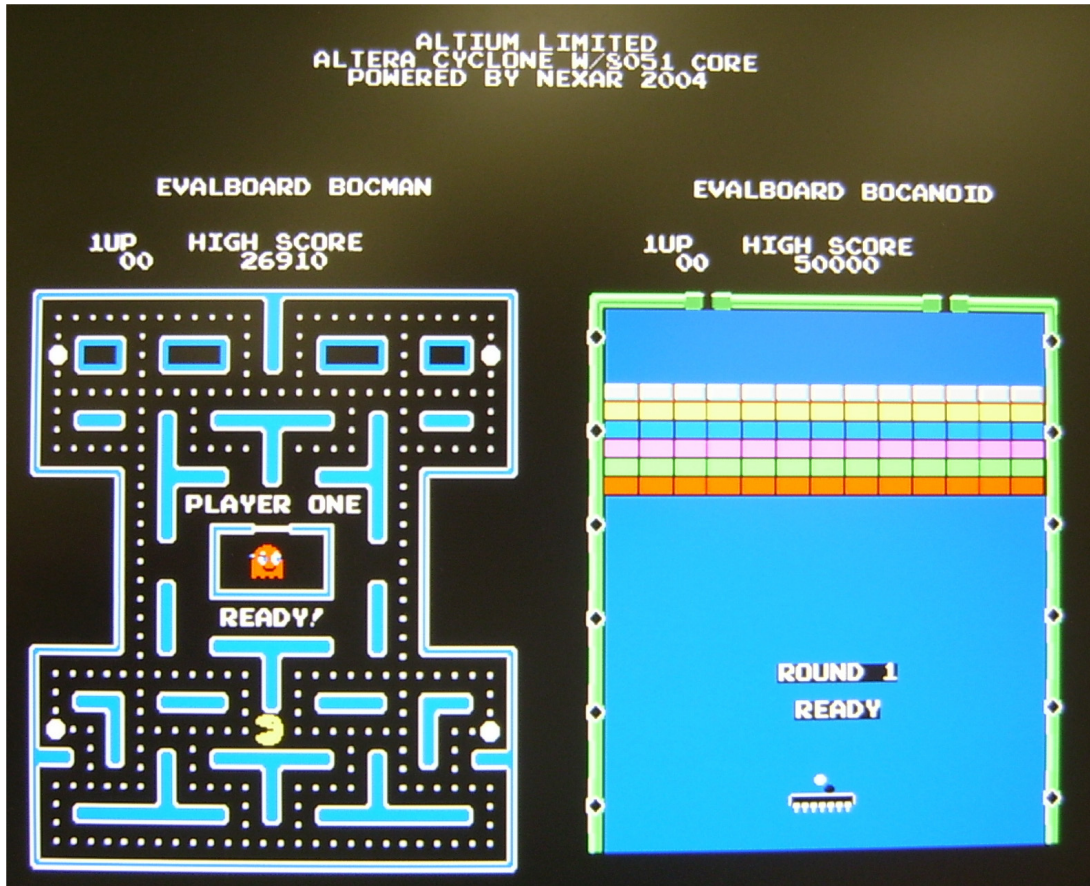
The main-line code does nothing but initialize the program data structures, draw the board, and then loop endlessly polling the mouse and inserting player movement codes into the global player *inputs* variable.

The bulk of the work is done in the game 'tick' routine.

This routine updates (moves) the bat and the ball and takes care of ball deflection off objects, destroying bricks, updating the score, playing sounds, and maintaining the state of the game.

Double MCU – BOCMAN & BOCANOID

The projects are effectively unchanged on the double-MCU platform, except for a few cosmetic changes to accommodate the split-screen and the removal of the debugging information display.



BOCMAN can be reset at any time by pressing the <ESC> key on the keyboard. BOCANOID can be reset at any time by pressing both left and right mouse buttons together.

Arcade Graphics Platform – Build Instructions

The Arcade Graphics Platform comprises two distinct FPGA projects:

- Single MCU (`VideoArcade.PRJFPG`) – runs BOCMAN and BOCANOID with sound.
- Double MCU (`VideoArcadeMCUX2.PRJFPG`) – runs BOCMAN and BOCANOID without sound.

The following sections describe the process for building and running the Arcade Graphics Platform and associated embedded software projects.

Single MCU Projects

Installation

Design files have been targeted to be built from the `\Examples\LiveDesign Evaluation Board\Reference Designs\VideoArcadeGames - TSK51\HardwarePlatform` folder of the installation.

BOCMAN

1. Open the project `VideoArcade.PRJFPG`.
2. Go to **Project Options** for the embedded software project `Games.PRJEMB`.
3. Under C Compiler Options (Preprocessor), ensure that **BUILD_BOCMAN** is defined and that all other defines are prefixed with **"NO_"**.
4. Under Linker Options (Memory), ensure that XRAM is 1536 bytes based at 0x7A00 and XROM is 6556 bytes (based at 0x0000).
5. Compile and update the embedded code in the MCU (from the Devices view)¹.
6. Plug a keyboard into the PS/2 connector labeled "KB" on the EvalBoard.
7. Connect the sound low-pass filter to the connector labeled "USER I/O 2" on the EvalBoard.

After downloading and running the processor, the game should be waiting for a key press to start. Use the arrow keys to control the player. When the game ends, the processor must be reset to restart the game (e.g. use the reset button on the EvalBoard).

BOCANOID

1. Open the project `VideoArcade.PRJFPG`.
2. Go to **Project Options** for the embedded software project `Games.PRJEMB`.
3. Under C Compiler Options (Preprocessor), ensure that **BUILD_BOCANOID** is defined and that all other defines are prefixed with **"NO_"**.
4. Under Linker Options, Memory, ensure that XRAM is 1536 bytes based at 0x7A00 and XROM is 6556 bytes (based at 0x0000).
5. Compile and update the embedded code in the MCU (from the Devices view)¹.
6. Plug a mouse into the PS/2 connector labeled "MOUSE" on the EvalBoard.
7. Connect the sound low-pass filter to the connector labeled "USER I/O 2" on the EvalBoard.
8. After downloading and running the processor, the game should be waiting a left mouse click button to start. Use the mouse to control the bat. The game may be restarted at any time by resetting the processor (e.g. use the reset button on the EvalBoard).

¹ This stage assumes that you have already completely built the FPGA design project (i.e. by running the Build stage in the Process Flow for the target physical device, in the Devices view). If the design has not been built, then this should be carried out at this stage, instead of simply compiling and updating the embedded code in the MCU.

Double MCU Project

Installation

Design files have been targeted to be built from the \Examples\LiveDesign Evaluation Board\Reference Designs\VideoArcadeGames - TSK51\HardwarePlatform folder of the installation.

BOCMAN and BOCANOID

1. Open the project VideoArcadeMCUX2.PRJFPG.
2. (Re)build the entire FPGA project.
3. Plug a keyboard into the PS/2 connector labeled "KB" on the EvalBoard.
4. Plug a mouse into the PS/2 connector labeled "MOUSE" on the EvalBoard.
5. After downloading to and running both processors, the BOCMAN game should be running on the left-hand side of the display, waiting for a key-press to start the game. BOCANOID should be running on the right-hand side of the display, waiting for a left mouse-button click to start the game.

Revision History

Date	Version No.	Revision
28-Sep-2004	1.0	New product release
27-Feb-2008	2.0	Updated for Altium Designer Summer 08

Software, hardware, documentation and related materials:

Copyright © 2008 Altium Limited.

All rights reserved. You are permitted to print this document provided that (1) the use of such is for personal use only and will not be copied or posted on any network computer or broadcast in any media, and (2) no modifications of the document is made. Unauthorized duplication, in whole or part, of this document by any means, mechanical or electronic, including translation into another language, except for brief excerpts in published reviews, is prohibited without the express written permission of Altium Limited. Unauthorized duplication of this work may also be prohibited by local statute. Violators may be subject to both criminal and civil penalties, including fines and/or imprisonment. Altium, Altium Designer, Board Insight, Design Explorer, DXP, LiveDesign, NanoBoard, NanoTalk, P-CAD, SimCode, Situs, TASKING, and Topological Autorouting and their respective logos are trademarks or registered trademarks of Altium Limited or its subsidiaries. All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed.