

OGP Video Controller

Concept: Timothy Miller
Authors: Timothy Miller, Patrick McNamara
Open Graphics Project
Open Hardware Foundation
Traversal Technology LLC

March 13, 2007

Contents

1 Introduction and Motivation

The main objective of the Open Graphics Project Video Controller (“VC”) is to provide a powerful and flexible means for specifying video modes of any sort, while still keeping the number of logic gates to a minimum. Many FPGA and ASIC architectures come pre-built with dedicated on-chip static RAM blocks (BRAM). These BRAMs are generally in greater abundance than we can take advantage of, so it seemed sensible to us that we might use one as a “program store” to hold a display list that describes the activity of the VC.

This approach has allowed us to safely ignore details of many video modes, such as interlacing, serration pulses, and even packet video standards like DPVL.

2 Basics of Video Modes

This document generally assumes an understanding of video signals and video timing. However, to minimize ambiguity and misunderstanding, this section provides a brief introduction.

Computer video is a 1-dimensional encoding of 3-dimensional information. Motion video is divided into a sequence of still images (“frames”). Each frame is divided into a sequence of rows (“scanlines”). And each row is divided into columns (“pixels”). To indicate to a monitor the boundaries between these dimensions, synchronization (“sync”) signals are transmitted along with the raw video signal.

The Vertical Sync (VSYNC) signal indicates the end of one frame and the beginning of the next. Horizontal Sync (HSYNC) indicates the end of one scanline and the beginning of the next. Surrounding the sync signals are “blanking” periods, where no valid pixels are being transmitted. The Data Enable (DE) signal is used to indicate when valid pixels are present in the data stream. (DE is not transmitted to an analog monitor.)

Sync signals are digital, indicating (to an analog monitor) their state by high and low voltages. However, different video modes require different polarities for these signals. As is common in digital electronics, we refer to the signal states as “asserted” and “deasserted”, regardless of the assertion polarity.

A scanline is divided into two major time periods. They are the active video period, and the horizontal blanking period. The horizontal blanking period itself is divided into a blank period before the sync pulse (the horizontal

3.1 FETCHing, SENDing, and WAITing

The vast majority of a video program is dedicated to the specification of *timing* information, as described in the *Basics of Video Modes* section. However, the program must also specify which pixels to send, as well as when to send them. For that, we use the **FETCH** and **SEND** instructions. In addition, there is the **WAIT** instruction, used to specify only timing. Detail on these is given below in the **OPCODEs** section.

FETCH is used to request a sequence of pixels to be read from graphics memory. Due to inherent latencies in the memory system, requests must be made sufficiently far in advance (typically the time of one scanline) of when the data must be available to be transmitted. **FETCH** makes the request, and when the memory system responds with the requested pixels, they are stored in a queue, ready to be consumed by a **SEND** instruction.

SEND causes a number of pixels to be retrieved from the queue and sends them to the monitor, along with sync signals.

WAIT is similar to **SEND**, in that it manipulates sync signals for some number of machine cycles. However, unlike **SEND**, it does not fetch any pixels from the queue. **WAIT** is used during blanking to specify timing.

While it is intuitive that **SEND** and **WAIT** should take some number of machine cycles, proportional the number of pixels in the scanline or the number of cycles in a blanking period, it is also the case that **FETCH** takes time from the perspective of the video program. **FETCH** consumes one video machine cycle and cannot be done at the same time as **SEND**; therefore, it must be take the place of one **WAIT** cycle during blanking.

3.2 Instruction Format

An instruction is divided into fixed-size fields with two main formats:

	31	30	29	28:27	26	25	24	23:21	20:9	8:0
[A]	DE	VSYNC	HSYNC	Cursor	RET	INT	0	OPCODE	COUNT	IADDR
[B]	DE	VSYNC	HSYNC	Cursor	RET	INT	0	OPCODE	Memory_Address	

3.3 Common Flags

The following fields are common to all instructions:

<i>Field</i>	<i>Description</i>
DE	Value of the Data Enable signal.
VSYNC	Value of the Vertical Sync signal.
HSYNC	Value of the Horizontal Sync signal.
Cursor	<ul style="list-style-type: none"> • During blanking, the value '00' leaves cursor coordinates unchanged. • During active video, '00' advances the cursor X coordinate. • '01' increments the cursor Y coordinate and resets X to its initial value. • '10' resets the cursor Y coordinate. • '11' resets both cursor X and Y coordinates.
RET	Instruction flow control: Return from subroutine.
INT	Indicates that the video interrupt signal should be asserted.
x	Bit 24 is not assigned to any function and should be set to zero.
OPCODE	Type of instruction to be executed (see below).
COUNT	Indicates the number of times this instruction should be repeated before execution of the following instruction (RepeatCount), or the number of words to fetch from graphics memory. (FetchCount).
IADDR	For a JUMP or CALL instruction, the BRAM address of the next instruction to be executed.
Memory_Address	For the ADDR instruction, the address in graphics memory from which pixels should be fetched.

3.4 COUNT values

CALL, **WAIT**, and **SEND** instructions expect an iteration count (**RepeatCount**), indicating the number of times that instruction should be executed before advancing to the next instruction. The encoding of the **RepeatCount** value, however is not straightforward. For these instructions, only counts 0 through 2048 are defined. Moreover, they are encoded as 2049 minus the desired count value. That is:

$$\text{COUNT} = 2049 - \text{RepeatCount}$$

For **FETCH**, however, the number of memory words to fetch (**FetchCount**) is stored as a normal binary number in the field, so for this instruction:

$$\text{COUNT} = \text{FetchCount}$$

Whenever **RepeatCount** is referred to as being assigned a value, it should be converted before being incorporated into the instruction word.

3.5 Subroutines and the RET flag

The **RET** flag is valid for all but the **JUMP** instruction and causes execution to resume from the calling routine. Only a single level of subroutine call is allowed.

At the outer level, the **RET** flag causes execution to resume at address 0 (zero) of the BRAM program store.

If an instruction has both a **RepeatCount** greater than one and the **RET** flag is set, the **RET** will take effect after the last iteration.

If the **CALL** instruction has a **RepeatCount** greater than one, a return in the subroutine will start the next iteration of the **CALL** instruction, which then calls to the same subroutine again.

If a return is executed, and the calling instruction itself also will execute a return (last loop and **RET** flag set), both returns are executed together with no delay. In other words, a double return causes an immediate jump to the top of the program file.

A return does not take any execution time. For instance, if a **CALL** has iterations left to count, it is the **CALL** instruction itself that will be executed immediately after the instruction with the **RET** flag set.

3.6 Instruction OPCODEs

The following are details of the 7 OPCODEs defined for the VC.

3.6.1 JUMP: 0

Unconditionally branch to the program store address specified in the **IADDR** field.

Both **RET** and **COUNT** are meaningless for this instruction, and they should be set to '0' and '2048' (0 encoded as **RepeatCount**), respectively.

This instruction uses format **[A]**.

3.6.2 CALL: 1

Make a subroutine call to the program store address in the **IADDR** field. The call is made **RepeatCount** number of times. On return from the last iteration, if the **RET** flag is set, execution resumes at the top of the program store (address 0).

Note that **CALL** itself takes one machine cycle, and that time should be accounted for in subroutines. **CALL** instructions should take the place of one **WAIT** cycle during blanking.

This instruction uses format **[A]**.

3.6.3 WAIT: 2

Affect only sync signals for **RepeatCount** cycles. This instruction is used during blanking. Typically, **DE**, **VSYNC**, and **HSYNC** signals would be set to appropriate values to indicate the the various phases of blanking.

IADDR is ignored for this instruction. This instruction uses format **[A]**.

3.6.4 SEND: 3

Over a period of **RepeatCount** cycles, send **RepeatCount * 4** pixels to the video output (DAC or DVI). Typically, **DE**, **VSYNC**, and **HSYNC** signals would be set to appropriate values to indicate active video.

IADDR is ignored for this instruction. This instruction uses format **[A]**.

Before this instruction can operate properly, pixels must have been fetched from graphics memory using the **FETCH** instruction.

3.6.5 FETCH: 5

Fetch **FetchCount * 256 bits** from graphics memory. The number of pixels this represents depends on the bit depth. For 32-bit, this is 8 pixels. Fetches always start on 256-bit boundaries and can be only whole numbers of 256-bit words.

From the perspective of VC timing, **FETCH** takes exactly one cycle and must be executed during blanking. It must take the place of one **WAIT** instruction cycle. After **FETCH** is executed, a separate mechanism requests data from graphics memory "in the background".

FETCH causes the current graphics memory pointer to be incremented by `FetchCount` 256-bit words. However, the increment takes up to 3 cycles to complete, so FETCH should not be closely followed by FETCH, ADDR, or INC.

FETCH is typically used as the last cycle of HBP, immediately preceeding a SEND instruction. FETCH must be executed at least one scanline in advance of its corresponding SEND.

Note: `FetchCount` for FETCH and `RepeatCount` for SEND are never the same value. FETCHes are always done in 256-bit units, while SENDs are done in units of 2 or 4 pixels of 8, 16, or 32 bits.

IADDR is ignored for this instruction. This instruction uses format [A].

3.6.6 ADDR: 6

The 21-bit `Memory_Address` field is shifted left by 7 bits to form a 28-bit memory address. (Memory addresses always refer to 256-bit words.) This address is taken as the new value for the current graphics memory pointer where pixels will be fetched from.

For VC timing, ADDR takes one machine cycle, and it must take the place of one WAIT instruction cycle during blanking.

ADDR takes only one cycle to complete, so it can be followed immediately by FETCH or INC. However, it must not come closely *after* either FETCH or INC.

ADDR is typically used only once, at the very beginning of a display program, and it can be followed immediately by an INC instruction in order to specify all bits of a 28-bit graphics memory address.

This instruction uses format [B].

3.6.7 INC: 7

The 21-bit `Memory_Address` field is sign-extended to 28 bits and added to the current graphics memory pointer.

For VC timing, INC takes one machine cycle, and it must take the place of one WAIT instruction cycle during blanking.

This sum takes up to 3 cycles to compute, so INC must not be closely followed by either FETCH or ADDR.

INC is typically found immediately following a SEND instruction, as the first cycle of HFP.

This instruction uses format [B].

4 Address Space and Global Configuration Registers

In order to be programmed and configured, the VC must occupy a portion of a graphics processor's PCI address space. The design expects to occupy an aligned space of 1024 32-bit words. From the perspective of the PCI bus, the VC's registers are found starting at some base address `ENGINE_BASE + VIDEO_BASE + 0x000`, up to `ENGINE_BASE + VIDEO_BASE + 0xFFC`, a 4096 byte area. Henceforth, register addresses are assumed to be relative to `ENGINE_BASE + VIDEO_BASE`, and to avoid confusion, we use byte addresses (all multiples of 4). The VC only accepts 32-bit word writes.

The program store occupies the first 512 words of the address space, offsets `0x000` through `0x7FC`. In the current design, the BRAM containing the program store is dual ported. With due care, program words can be modified while the VC is actively running. This can be especially useful for things like panning that would require the initial graphics memory address to be changed on the fly. More invasive changes should be done while the VC is disabled.

Besides the video program, there is a handful of configuration registers that affect VC behavior. They occupy the upper portion of the address space, starting at offset 0x800.

<i>Offset</i>	<i>Name</i>	<i>Description</i>
0x800	VID_CURSOR_X	Initial value of cursor X coordinate. counter
0x804	VID_CURSOR_Y	Initial value of cursor Y coordinate. counter.
0x808	VID_CLEAR_INT	Write any value to clear the video interrupt signal.
0x80C	VID_BIT_DEPTH	Video bit depth. Currently, only values 8, 16, and 32 are defined for analog and dual-link DVI. These values may change, and more values will be added to indicate single-link DVI modes.
0x810	VID_RESET	Write any value to reset the VC to a known state. This affects only internal registers, leaving the program store and config registers unchanged. This would typically be done while the VC is disabled.
0x818	VID_ENABLE	Write '0' to halt and disable the VC. Write '1' to enable the VC and activate video. It is recommended to reset the VC before enabling it.
0x81C	VID_CLOCK	Bit 1 is used to select between two different video clock sources. Bit 0 is the (active high) reset signal to the clock generators.

Note: Setting the video clock speed (dot rate) is documented separately.

5 Example

Many progressive video modes can be specified by template. We have written a “typical” video program that can be used to specify most common progressive video modes by tweaking operands to a fixed sequence of instructions. The following pseudocode describes the structure of the template:

```

; vp_addr = base 256-bit-word address of framebuffer
; vp_width = width of viewport in pixels
; fp_width = width of framebuffer in pixels (>= vp_width)
; depth = depth in bits per pixel
; hfp = duration of HFP in pixels
; hsync = duration of HSYNC in pixels
; hbp = duration of HBP in pixels
; vfp = duration of VFP in scanlines
; vsync = duration of HSYNC in scanlines
; vbp = duration of VBP in scanlines
; ppc = number of pixels transmitted per pixel clock cycle

; VFP, VSYNC, and VBP minus one scanline
000  CALL  022  Repeat=vfp      HSYNC
001  CALL  026  Repeat=vsync   HSYNC,VSYNC
002  CALL  022  Repeat=vbp-1   HSYNC

; Here, we set the initial framebuffer address, fetch the first scanline
; from the framebuffer, skip any padding on the right, and finish off the
; last scanline of the VBP.
003  ADDR   vp_addr/128      HSYNC
004  WAIT   Repeat=hsync/ppc-1  HSYNC,RESET_CURSOR_XY
005  INC    vp_addr&0x7f
006  WAIT   Repeat=hbp/ppc+1
007  FETCH  vp_width*depth/256
008  WAIT   Repeat=(vp_width/ppc)-3
009  INC    (fb_width-vp_width)*depth/256

```

```

010     WAIT     Repeat=hfp/ppc-1

; Vertical active video, except last line
011     CALL     016     Repeat=vp_height-1  HSYNC

; Last active scanline, where we skip the FETCH
012     WAIT     Repeat=hsync/ppc           HSYNC,RESET_CURSOR_X
013     WAIT     Repeat=hbp/ppc
014     SEND     (vp_width/ppc)           DE
015     WAIT     Repeat=hfp/ppc           RETURN

; Active scanline
016     WAIT     Repeat=hsync/ppc-1       HSYNC,RESET_CURSOR_X
017     WAIT     Repeat=hbp/ppc-1
018     FETCH    vp_width*depth/256
019     SEND     (vp_width/ppc)           DE
020     INC      (fb_width-vp_width)*depth/256
021     WAIT     Repeat=hfp/ppc-1       RETURN

; A blanking scanline
022     WAIT     (hsync/ppc)-1           HSYNC
023     WAIT     hbp/ppc
024     WAIT     vp_width/ppc
025     WAIT     hfp/ppc                 RETURN

; A vertical sync scanline
026     WAIT     (hsync/ppc)-1           HSYNC, VSYNC
027     WAIT     hbp/ppc                 VSYNC
028     WAIT     vp_width/ppc           VSYNC
029     WAIT     hfp/ppc                 VSYNC, RETURN

```

6 Source Code

Source code to both the video controller itself and library code for generating video programs can be found on the Open Graphics Project Subversion server, graciously hosted by the Mplayer project.

Instructions for checking out a copy of the OGP repository can be found here:

http://wiki.duskglow.com/tiki-index.php?page=Development_Tools

The root directory of the OGP repository can be found here:

<https://svn.suug.ch/repos/opengraphics/main/trunk/>

Source code to the VC itself can be found here:

https://svn.suug.ch/repos/opengraphics/main/trunk/rtl/vid_ctl/

Library code for generating video modes can be found here:

https://svn.suug.ch/repos/opengraphics/main/trunk/drivers/lib/video_controller/

To use the library code, download or check out all of the files in this directory and type `make`. This will generate, among other things, `progressive.c` that contains a function that will generate a video mode from timing parameters. See `640x480.c` for a concrete example, because not all of the parameters have exactly the same meaning as they do in the Example section above.

7 Cursor Overlay

The cursor overlay mechanism is not fully defined at this time.

8 Licensing

All software released by the Open Graphics Project is released under the X11 license.

All hardware designs (Verilog code, schematic diagrams, etc.) are released under the GNU General Public license. Contact Traversal Technology LLC at <http://www.traversaltech.com> for commercial licensing options.

This work, entitled “OGP Video Controller” is Copyright © January 9, 2007, Timothy Miller. Copyleft: this work of art is free, you can redistribute it and/or modify it according to terms of the Free Art license. You will find a specimen of this license on the site Copyleft Attitude <http://artlibre.org> as well as on other sites.