Project Report - Team N64

Neil Ryan, Steven Kool, Rachel Yanovsky

December 11, 2016

Contents

1 Statement of Use

We grant permission to any person to use any part of this project under the "Beerware" license, detailed below:

This project was written by Rachel Yanovsky, Steven Kool, and Neil Ryan.
As long as you retain this notice you can do whatever you want with this stuff.
If we meet some day, and you think this stuff is worth it, you can buy us a beer in return.

Our project can be found at the following link:
`https://github.com/soctar/GBA`

Feel free to contact us with any questions:
nryan@andrew.cmu.edu
skool@andrew.cmu.edu
ryanovsk@andrew.cmu.edu

2 Origin of the Team Name

Originally, we were floating the idea of extending the N64 project from 2014 instead of a Game-Boy Advance. The N64 project had a mostly working CPU, the main work would be the Reality Co-Processor (RCP), which is the main DSP and GPU for the Nintendo 64. The pipeline of the N64 is such that everything flows through this chip, and the documentation for the RCP is fairly sparse. From the previous N64 group's report, we found cen64, a cycle accurate N64 emulator that currently runs Super Mario 64 and a host of other games with high accuracy.

The Reality Co-Processor is made up of two chips, the Reality Signal Processor (RSP) and Reality Display Processor (RDP). We reached out to Tyler Stachecki, the creator of cen64, to see if he had some sort of emulator for just the RCP, or preferably emulators for the RSP and RDP. For posterity, we've included his response below, with important sections in bold. If future students ignore all of our advice at least heed this: Do not attempt an N64. It is a wholly unreasonable project for a semester. Perhaps if several groups dedicated themselves to sections of it, it might be only a stretch, but even with 8 highly talented engineers, we feel as though the project would be doomed to fail.

Hi Neil,

I still remember working with last years' folks! Very neat to see this project being continued and wish you the best of luck.

The RCP is not completely cycle-accurate yet by any means. CEN64's implementation of the RSP (the vector processor within the RCP, used for both audio mixing and T&L) is written in a quasi cycle-accurate manner [1], as was the VR4300 that last years' students worked on. Probably the biggest difference between the RSP model that I've written and hardware is that the latter will dual-issue instructions under certain conditions, while my pipeline unconditionally single-issues for the time being. I've got the backend logic ready for dual-issue (you will see that rsp_cycle_ invokes both rsp_v_ex_cycle and rsp_ex_cycle), but the frontend is still dual-issue agnostic.

Also regarding the inaccuracy of the RSP, there are also some timing differences – e.g.., CEN64 doesn't simulate the DMAs, it just completes the command in a single cycle. The other components of the RCP (the audio, cart, controller, etc.) blocks are written in a similar manner in that they either use fixed time-delay models or also just complete commands in a single cycle. However, this is something you can easily work around in HDL since everything is interrupt-driven – notice that virtually everything controller or CPU in the console will atomically set local MMIO registers and raise an interrupt when they're done with their DMA operation or whatever else.

The RDP, on the other hand, is considerably more difficult for the following reasons: **1) There only exists one open-source software (pixel-accurate) renderer which all current N64 emulators share (which was originally derived from MAME). I encourage you to look at it, because the coding style used is absolutely horrendous and very well may be the most vile thing you've ever laid eyes on. Trying to digest everything that the RDP is doing, never mind writing a simulator or HDL, will be quite a bit of work in and of itself.**

2) The RDP does *not* raster triangles in a manner similar to virtually any other piece of graphics hardware. Read this forum post [2] as it's a really good summary of how the RDP renders things.

3) Also keep in mind that the N64 uses a unified memory model - not a single CPU has any memory that is dedicated to it (aside from it's caches) – so you would need to arbitrate accesses and get all the timing sorted out for this as well. Since you're working on the clock speeds of an FPGA, SDRAM definitely won't cut it and you'll probably find that you need to use (at least) DDR2 for the bandwidth you're looking for. The RDRAM in the N64 is clocked at 250MHz and operates on both edges of the clock (and since it's RDRAM, the data bus is 9 bits wide – not 8 bits!) The RDP actually uses the 9th bit for coverage information, so you can't ignore that fact and you'd likely have to drive two ranks of memory in parallel. You could also just use blockram on the FPGA, but I'm not sure you'll have access to something with enough blockram available.

tl;dr: My advice to you would be the same as the last years' team and was reflected in their report - do NOT bite off more than you can chew. I don't think it's possible to 'clone' the RCP in one semester, even with a team of 3-4 very talented and dedicated engineers. If you want to do something graphics-oriented, I would ignore the entirety of the RCP and only try to write HDL for the RDP (and that in and of itself is quite a bit of work, for the reasons mentioned above). You can use last year's work to "drive" the RDP – have it write the display lists to memory and treat the RDP as an interrupt-driven signal processor (the console effectively does the same thing). If you hack a little bit on CEN64, you should be able to get it it to dump the display lists (the "instructions") that real game carts are ultimately sending to the RDP, along with all the texture data and whatnot. In theory, you can just "play back" these display lists to the RDP to get it to render realtime content for your demo at the end of the semester demo. FYI: I'd first investigate the size and bandwidth required for these display lists – double check to make sure it's something that's feasible with what your FPGA provides.

Let me know what you end up doing!
Tyler

[1] `https://github.com/tj90241/cen64/blob/master/rsp/pipeline.c#L208`

[2] `http://forums.cen64.com/viewtopic.php?f=14&t=237`

3 Overview

The GameBoy Advance (GBA) was an extremely popular handheld console by Nintendo, released in 2001, succeeding the GameBoy Color. As with previous Nintendo handheld consoles, it was a single-player device, with the option of attaching a link cable between devices for multiplayer interactions. The serial port on the top of the device that the link cable attached to was also used for a wireless adapter in *Pokemon: Fire Red and Leaf Green*, as well as for links between the Nintendo GameCube and the GameBoy Advance. The GameBoy Advance was also backwards compatible with GameBoy Color games.

The Game Boy Advance (GBA) is composed of four major subsystems - the ARM7TDMI CPU, the graphics pipeline, the sound engine, and the Link Cable serial interface. The GBA also includes a Zilog Z80 processor for backwards GameBoy Color compatibility. Additionally, the Game Boy Advance contains a Direct Memory Access (DMA) controller, several programmable timers, an interrupt controller, and 10 buttons. The graphics pipeline outputs to a 240x160 RGB LCD screen; the sound outputs to either a mono speaker, or the GBA's stereo headphone jack. The system clock is the same as the CPU clock, running at 16.78MHz.[2]

On a high level, the GBA architecture is a MMIO register based system. The ARM7TDMI acts as a master to configure the various subsystems. Generally, registers act as control points, but are occasionally used to give the status of subsystems to the CPU.

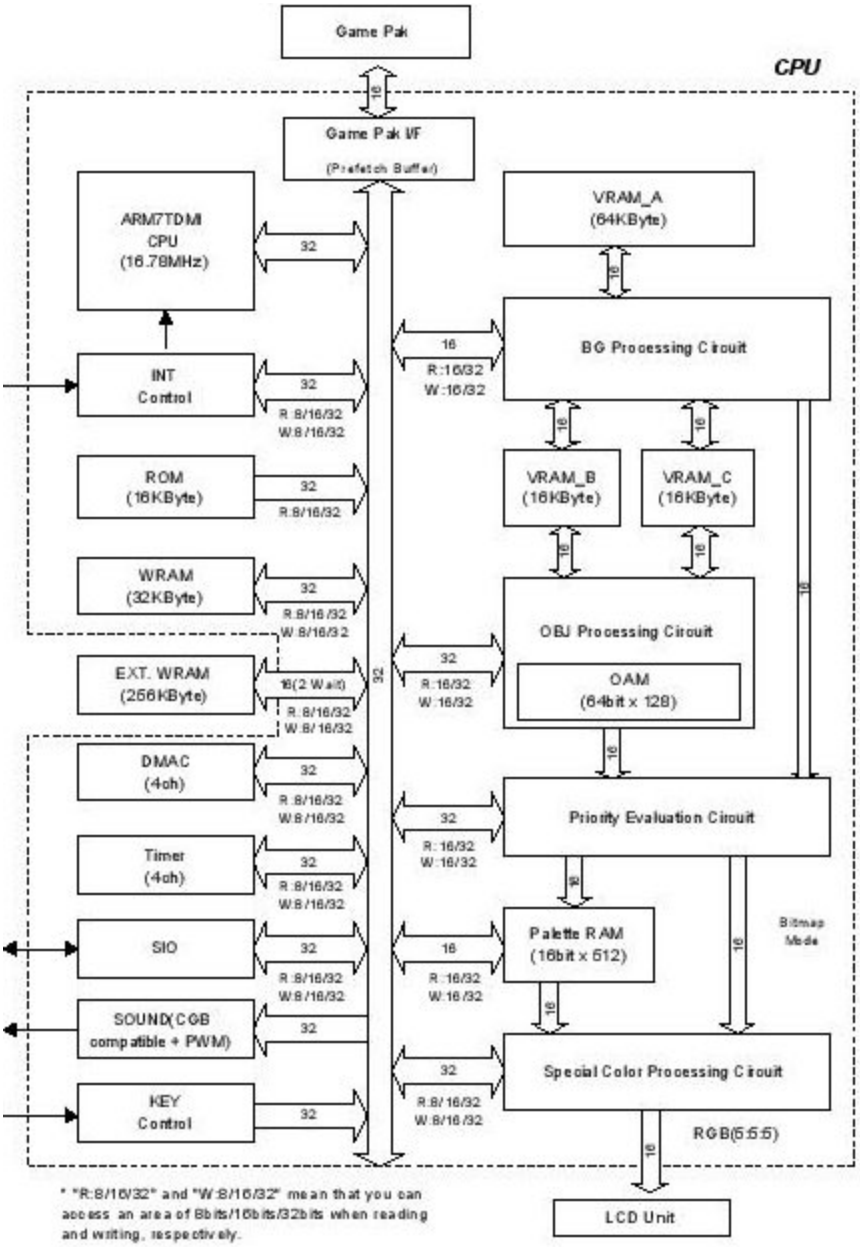
Initially, we had planned to finish the semester with a working GameBoy Advance that could play three games: Pokemon Ruby, Legend of Zelda: Minish Cap, and Kim Possible 3: Team Possible (the favorite childhood games of our group members), but several aspects of the project ended up taking much, much longer than we anticipated. We feel, however, that enough dense roadblocks have been dealt with to make this a very achievable project for future groups to complete.

3.1 GBA Architecture

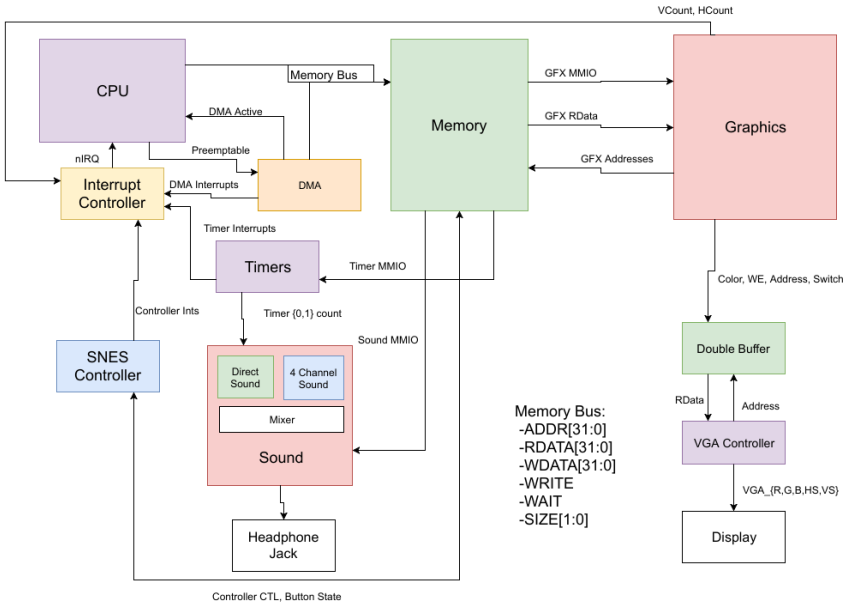
While the GBA generally operates in a master-slave fashion, there are several points of interactions between the subsystems. For the most part in this project, we made no attempt to create a cycle accurate system. It is possible that some games exist that abuse the specifics of the hardware enough to make this a problem, but from our understanding of the GBA specification, there is little incentive to cut game timing close enough to make this a problem - the "hacky" characteristics of some 80s arcade cabinets are unlikely to be present in the GameBoy Advance. Namely, games are likely compiled, instead of hand-assembled.

As previously stated, the ARM7TDMI core controls most of the system. The GBA only has one memory bus which is shared by all subsystems. To avoid multiple drivers, DMA takes priority over the CPU and pauses the CPU when it is active. Additionally, the CPU and DMA cannot read from graphics memory outside of VBLANK and HBLANK. While not in a display blanking period, the graphics has control over the memory ports to the three sections of VRAM, OAM, and Palette RAM.

3.2 GBA Schematic



3.3 System Diagram



4 Infrastructure

4.1 Custom ROMs

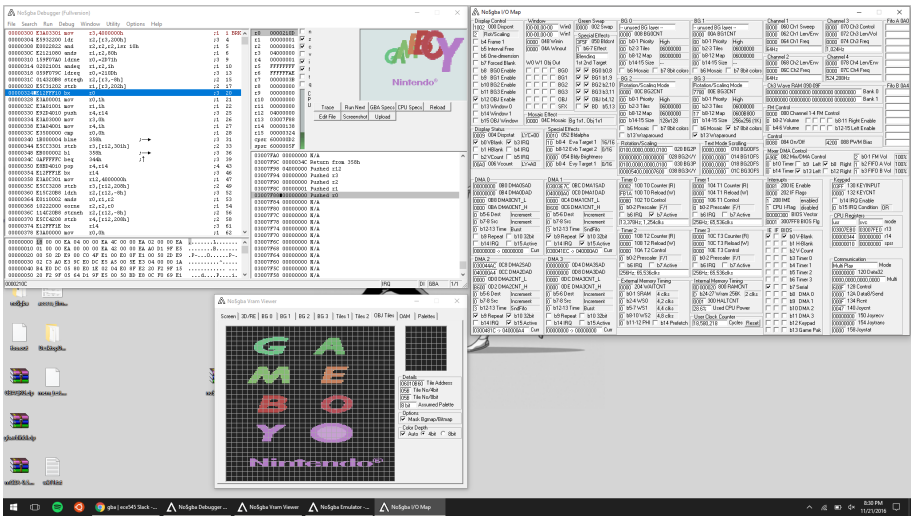
A healthy amount of googling can get pretty much any ROM for the Gameboy Advance, but it can be extremely helpful to be able to write custom ROMs for testing (or if a final demo needs to be scaled back). Conveniently, a surprisingly large group of hobbyists find programming games for the GBA to be enthralling. We used the DevKitARM tool chain - a tutorial is at the link below.

`https://www.reinterpretcast.com/writing-a-game-boy-advance-game`

4.2 Emulators

Many emulators exist for the GameBoy Advance - none claim to be cycle accurate (that we found that worked). For debugging, we primarily used mGBA and no$gba - specifically the "full" version of no$gba. In most cases, no$gba was more valuable, but mGBA gives the current state of memory - no$gba only gives the state of memory at boot (only memory locations defined by the BIOS and the GamePak are valid).

No$gba (pronounced No-cash-gba), however, has some poorly documented behavior. First, to boot the BIOS in No$gba, there needs to be a BIOS file in the same directory as `NO$GBA.exe` named `gba.rom`. Second, breakpoints are set in a format completely unique to no$gba - for example, to break on any writes to addresses between `0x040000A0` and `0x040000A4` (DMA FIFO), the format is `[040000A0..040000A4]!!`. More details are in the breakpoints help dialog in no$gba.
In the strange and bizarre case that future groups fail to realize exactly how powerful no$gba, a screenshot of BIOS debugging is below. A zip archive of no$gba is in our Git repository which includes everything needed to run the debugging version except ROMs (which should be stored in the SLOT folder) and the BIOS (which should be stored in the top level folder, named `gba.rom`).



5 Memory

A word on the GBA is defined as 32-bits, a half-word is 16bits. The Game Boy Advance memory is divided into 11 regions:
System ROM: Mapped from `0x00000000` to `0x00003FFF`. Holds the GBA BIOS.
CPU External RAM: Mapped from `0x02000000` to `0x0203FFFF`.
CPU Internal RAM: Mapped from `0x03000000` to `0x03007FFF`.
I/O Registers: Mapped from `0x04000000` to `0x04000807`. MMIO register mappings.
Palette RAM: Mapped from `0x05000000` to `0x050003FF`. Stores the color palette.
VRAM: Mapped from `0x06000000` to `0x06017FFF`. Video RAM.
OAM: Mapped from `0x07000000` to `0x070003FF`. Object (sprite) attribute memory.
Game Pak ROM, WS0: Mapped from `0x08000000` to `0x09FFFFFF`. Cartridge ROM.
Game Pak ROM, WS1: Mapped from `0x0A000000` to `0x0BFFFFFF`. Cartridge ROM.
Game Pak ROM, WS2: Mapped from `0x0C000000` to `0x0DFFFFFF`. Cartridge ROM.
Game Pak RAM: Mapped from `0x0E000000` to `0x0E00FFFF`. Cartridge RAM.[2]
Additionally, the upper 1M-bit of each ROM region is used as flash memory (presumably for game saves).

Memory from `0x04000000` to about `0x04000807` is mapped to I/O registers (the programmers manual only lists register mappings up to `0x04000208`, but hobbyist reverse engineering shows some internal registers up to location `0x04000800`).

VRAM is split into three regions: VRAM_A{`0x06000000 - 0x0600FFFF`}, VRAM_B{`0x06010000 - 0x06013FFF`}, and VRAM_C{`0x06014000 - 0x06017FFF`}. Palette RAM is split into two regions bg_palette_ram{`0x050000000 - 0x050001FF`} and obj_palette_ram{`0x050000000 - 0x050001FF`}. In graphics, the system expects to be able to read from all 5 of these regions in each cycle - the memory controller needs to allow for this. Additionally, our graphics system required that we be able to read two values from VRAM_A at once, though this is not strictly required by the architecture.

In the process of booting the system, the BIOS checks that a valid ROM is inserted by checking that the byte `0x96` is stored in memory at location `0x080000B2` and that memory locations `0x0BFFFFE0`

- `0x0BFFFFFF` are defined as halfwords `0xFFF0, 0xFFF1, 0xFFF2, ..., 0xFFFF`. Memory controller implementations that do not have a valid ROM mapped in memory need to emulate this behavior.

Memory regions are also mirrored across 24-bit boundaries, along their size. For example The three regions for Game Pak ROM are identical mappings, the difference is the number of wait states used in the access (see Memory Interface). We used the Zedboard's BRAM for every memory region except CPU External RAM and GamePak ROM. Originally it would've been possible to fit External RAM into BRAM, but the double buffer for graphics took up the BRAMs that we would have used. We planned to use the Zedboard's DDR3 RAM for the Game Pak ROM, the Game Pak RAM, and the CPU External Working RAM, but we didn't have time to implement it. MMIO registers will be implemented with the Zedboard fabric SRAM.

5.1 DRAM

Since the Zedboard only has so much BRAM, and asking for the requisite 32MB of BRAM to fit a cartridge is a tall order for any board, we had planned to stored the GamePak ROM and CPU External RAM (also referred to as "On-board RAM") in the Zedboard's DRAM. Since there isn't a way to initialize DRAM upon board reset, our plan was to store the GamePak on an SD card, then, at system reset, have the SD card controller write the contents of the SD card to DRAM. Given that the CPU's clock is only at 16.78Mhz, we would be able drive our DRAM controller with a sufficiently fast clock to make DRAM accesses appear single cycle from the CPU's perspective. This, as it turns out, is surprisingly difficult to implement. The SD card on the Zedboard is only connected to the ARM core, meaning that the programmable logic can not actually interface with the SD card. We were able to set up a Vivado SDK project that would read from the SD card on reset and write to BRAM using an AXI bus. Unfortunately we were never able to properly set up a DRAM controller for the SD card to write to. We planned to create an AXI master using HLS, that would write to DRAM, however we were unsuccessful. We ran out of time, and had other more pressing priorities, to fully learn how to debug the AXI bus.

5.2 Memory Regions Bus Widths

Memory Type	Bus Width	DMA		CPU	
		Read Width	Write Width	Read Width	Write Width
OAM	32	16/32	16/32	16/32	16/32
Palette RAM	16	16/32	16/32	16/32	16/32
VRAM	16	16/32	16/32	16/32	16/32
CPU Internal Working RAM	32	16/32	16/32	8/16/32	8/16/32
CPU External Working RAM	16	16/32	16/32	8/16/32	8/16/32
Internal registers	32	16/32	16/32	8/16/32	8/16/32
Game Pak ROM (Mask ROM, Flash Memory)	16	16/32	16/32	8/16/32	16/32
Game Pak RAM (SRAM, Flash Memory)	8	--	--	8	8

[2]

5.3 Memory Interface

The memory interface is a bus, shared by the DMA controller and the CPU. Bus contention is avoided by priority - if a DMA is ready, the CPU is paused until the DMA completes. Neither the CPU nor the DMA can write to VRAM, OAM, or Palette RAM until either `VBLANK` or `HBLANK` regions of the VGA cycle. Outside of `VBLANK` or `HBLANK`, the graphics pipeline assumes exclusive ownership of VRAM, OAM, and Palette RAM, and the CPU/DMA bus is electrically disconnected (High-Impedence) from the graphics memory. During `VBLANK` and `HBLANK`, the CPU/DMA bus drives the bus for graphics memory as well.

Bus accesses are sent to the memory controller, where they are forwarded to the memory region in which the address lies. If a system attempts to make an access to an undefined region, either the system will enter an error state, or the bus `ABORT` signal will be set high, depending on whether our target games make accesses to undefined regions during standard operation. If the latter is the case, we also have the option of mirroring undefined addresses onto defined regions (a step that the GBA does internally, but that isn't documented in the programmer's guide). The system is little endian.

In our architecture, since BRAMs are dual ported, the CPU/DMA bus is allowed to write to graphics memory outside of `VBLANK` or `HBLANK`. In an ideal system, our memory controller would trigger an `ABORT` or some other error signal if graphics memory was written to outside of `VBLANK` or `HBLANK`, but we didn't have time to add this functionality.

The various memories have different bus widths - to accommodate, the bus includes a `SIZE[1:0]` - where 3 corresponds to a word (32 bits), 2 to a half word (16 bits), and 1 to a byte (4 is reserved). When reading, the lowest two bits of the address will be used to determine the specific memory location, when writing, the combination of the lowest two address bits and `SIZE[1:0]` will be used to address specific locations.

Our architecture gives the graphics pipeline direct access to address and read data ports of the BRAM. The CPU/DMA bus has the following signals:

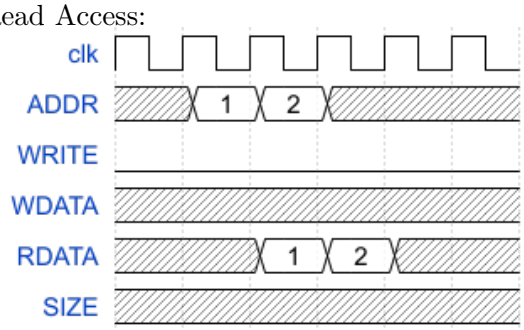
- ADDR[31:0] - address of the memory access (bottom 2 bits ignored for reads)
- RDATA[31:0] - Data read from memory location
- WDATA[31:0] - Data to write to memory location
- WRITE - Write enable; Memory controller determines byte write enable by ADDR[1:0] and SIZE
- SIZE[1:0] - Size of memory write 0 =¿ byte, 1 =¿ halfword, 2 =¿ word, 3 =¿ illegal
- PAUSE - Output. Pause the unit generating the memory access.

5.4 Memory Timings

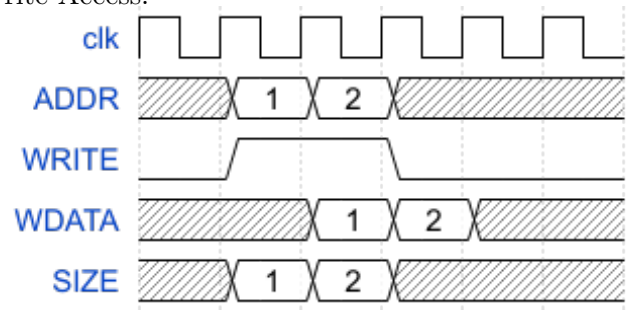
Each region of the memory also has a specific access time. Specifically, this is where the three Game Pak ROM regions differ, in the case of the first region, WS0 denotes that zero wait states should be used in the access, for the second, WS1 denotes that one wait state should be used in the access, and so on. The length of a wait state depends on the state of the `WAITCNT` MMIO register.

In the GBA system, accesses also recieve a speedup when they are sequential. We never planned to implement this, since DRAM and BRAM both give single cycle access. If it turned out that the timing differences are necessary for standard operation, we could have implemented delays for random accesses by asserting the `WAIT` bus signal as necessary. Different regions of GBA memory also have different access times associated with them - our plan of action was the same for this. Read and Write timings for our system are detailed below.

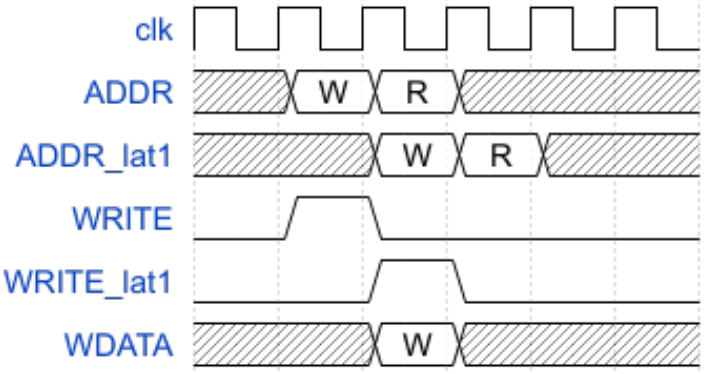
Read Access:



Write Access:



The difference between access timings for writes in our memory system (where write data is presented the cycle after write is asserted) and access timings for BRAMs (where write data is presented on the cycle that write is asserted) was fixed in our system adding several registers to delay address, write, and size by a cycle. This creates a bit of a problem - if a write is immediately followed by a read, both will try to present an address to the BRAM on the same cycle. See the diagram below - W corresponds to the memory write, R corresponds to the memory read. We solved this with a slight hack - `PAUSE` is asserted on every write. It would behoove future groups to redesign the controller to avoid this - it would make it easier to adhere to GBA memory region access timings.



Additionally, with system as described as above, each write will occur twice with the same address and data. Normally, this is fine, but each write will count as two words written to the

Direct Sound FIFO. This is only a problem in our system during the `STM` instruction where `WRITE` is asserted for multiple consecutive clock cycles. Our fix was specific to this case, the write enable for the BRAM is only asserted if `PAUSE` wasn't asserted in the previous cycle. Again, this is a bit of a hack - it would be prudent to implement a fix into the actual design for the memory controller.

5.5 Memory Region Access Timings

```
Region        Bus   Read      Write     Cycles
BIOS ROM      32    8/16/32   -         1/1/1
Work RAM 32K  32    8/16/32   8/16/32   1/1/1
I/O           32    8/16/32   8/16/32   1/1/1
OAM           32    8/16/32   16/32     1/1/1 *
Work RAM 256K 16    8/16/32   8/16/32   3/3/6 **
Palette RAM   16    8/16/32   16/32     1/1/2 *
VRAM          16    8/16/32   16/32     1/1/2 *
GamePak ROM   16    8/16/32   -         5/5/8 **/***
GamePak Flash 16    8/16/32   16/32     5/5/8 **/***
GamePak SRAM  8     8         8         5     **
Timing Notes:
*   Plus 1 cycle if GBA accesses video memory at the same time.
**  Default waitstate settings, see System Control chapter.
*** Separate timings for sequential, and non-sequential accesses.
One cycle equals approx. 59.59ns (ie. 16.78MHz clock).
Sequential vs Random access timings on p.23 of Nintendo GBA doc
All memory (except GamePak SRAM) can be accessed by 16bit and 32bit DMA.
```

[1]

6 CPU

6.1 Overview

The GBA uses an ARM7TDMI processor, implementing the ARMv4t ISA, which operates on a three-stage pipeline (fetch, decode, execute). When an instruction in the execute stage will take more than one cycle to complete, the CPU stalls the pipeline until the instruction completes. For branches, the CPU flushes the fetch and decode stages, reloading from the address that was branched to. The CPU also (per ARM specification) can switch between 32-bit ARM instruction mode and 16-bit THUMB instruction mode via the `BX` instruction. During execution, the program has accesses to 17 registers, with R13, R14, and SPSR swtiched between several banks depending on execution mode.

We had planned to get most of the CPU from an outside source, then implement the necessary features. Our options were either the Storm Core from OpenCores, the ARM9 chip embedded on the Zedboard, or a version of the ARM7TDMI that some engineer wrote between jobs. Interestingly, the core from some random engineer seemed the most promising since THUMB mode was partially

implemented already. The Storm Core had no infrastructure for THUMB mode in place, as well as some other issues, and the ARM9 core had some backwards compatibility issues between ISA versions that we wouldn't be able to implement hardware fixes for.

As we learned, getting the interactions between THUMB mode and ARM mode, especially at the points where the CPU switches between them is extremely tricky, especially when working in the context of a core that someone else wrote. All in all, getting the CPU to a working state took about 12 weeks of debugging 40+ hours a week. Currently, the CPU runs an unmodified BIOS perfectly, as well as several games that we tested, mostly written by outside sources. A warning to future groups - there may still be bugs in the CPU that need to be fixed to run commercial games, but they should mostly be fleshed out.

As a guide, since a decent portion of the core remains unmodified from original version, a short list of helpful information and documentation is below. It likely won't be enough to completely understand the system, but it should be enough to get started.

The CPU has 6 execution modes: Fast Interrupt (FIQ), Interrupt (IRQ), Undefined Instruction (UNDEF), Supervisor (SVC), Memory Abort (ABORT), and User (USER). Per the ARM specification, there should also be System Mode (think kernel mode in x86) where memory accesses to certain regions and writes to certain registers are unrestricted, but this and User mode are treated the same in this implementation.

In our implementation of the GBA, DMA can pause the CPU to preempt it so that DMA can take control of the memory bus. To avoid this preemption interrupting multi-cycle execute instruction or branches, we added a `preemptable` output that signals to DMA when the CPU can be preempted. Since multi-cycle instructions stagnate the pipeline (signal `StagnatePipeline` and `StagnatePipelineDel_Int` in `ControlLogic.vhd`) and branches refill the pipeline (with signal `PipelineRefilling` in Instruction Pipeline/Data In register (`IPDR.vhd`)), the CPU is preemptable when none of the signals listed above are asserted.

During normal program execution, PC (`REGFILE_INST.UM_REGISTER_FILE(15)`) points to the address of the instruction that is currently in the Fetch stage of the pipeline, which is either 8 or 4 greater than the address of the instruction that is currently in the execute stage of the pipeline, depending on whether the CPU is in ARM or THUMB mode. `ADDR` points to the next instruction to be fetched, either PC+4/PC+2, or the target of a taken branch or a jump.

6.2 Pipeline Overview

The core has three pipeline stages - Fetch, Decode, and Execute. Since accesses to memory have a 1 cycle latency, RDATA acts as the register for the instruction in the Fetch stage. In the case of instructions that are multi-cycle in the Execute stage and present addresses to memory, the Fetch stage instruction (and relevant signals) will be stored in the `PrefetchedInstruction` register in Instruction Pipeline/Data In register (`IPDR.vhd`) until the end of the multi-cycle instruction, then muxed into Decode stage registers.

The Decode stage instruction is output from the Instruction Pipeline/Data In register (IPDR, `IPDR.vhd`), through the THUMB Decoder (`ThumbDecoder.vhd`). The THUMB Decoder, if the

CPU is in THUMB mode, determines which half of the 32-bit word is the instruction for the current cycle from `HalfWordAddress`, then translates the THUMB instruction into a 32-bit ARM instruction. If the CPU is in ARM mode, the instruction is passed through unchanged. From the THUMB Decoder, the instruction is sent back to the IPDR then to `ControlLogic.vhd`, where the `IDC_*` signals are set based on what type of instruction it is.

For the Execute Stage, all `IDC_*` signals are passed to the same `IDR_*` signal. The ALU computes on the values on the ABus and the BBus, which are set by the ABusMultiplexer (`ABusMultiplexer.vhd`) and BBusMultiplexer (`BBusMultiplexer.vhd`), respectively. The output of the ALU is wired to the input of the Program Status Registers (CPSR, SPSR in `PSR.vhd`) and the input of the Result Bit Mask (`ResItBitMask.vhd`) where, selectively, bit 0 is cleared/set and bit 1 is cleared, based on signals from `ControlLogic.vhd`. The output of the Result Bit Mask is an input to the Address Mux (`AddressMux_Incrementer.vhd`)and the Register File (`Regfile.vhd`)

6.3 THUMB Mode

In THUMB mode, instructions are 16-bits instead of 32-bits long. THUMB instructions, however, are a subset of ARM instructions. Since CPU size wasn't a design concern, and THUMB mode was unimplemented in the core we started with, instructions go through a Decoder that translates the THUMB instructions into equivalent ARM instructions, so they can be passed to the Decode stage of the processor as ARM instructions. In reality, the ARM7TDMI likely uses muxes on each signal in the decode stage, either using the signal from the ARM decoder or the THUMB decoder based on the CPU mode, but implementing the Thumb Decoder this way allowed for minimal modifications (and didn't require a complete understanding of the processor).

6.4 Branches

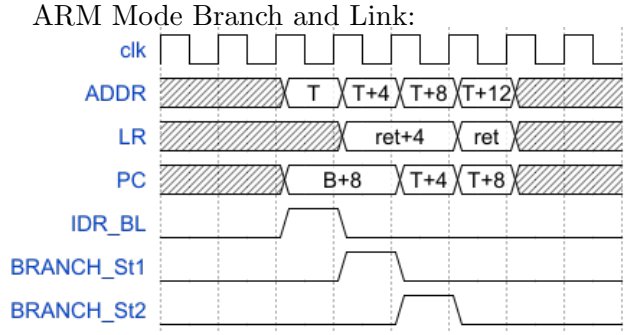
On a high level, there are three types of branches: Normal branches, Branch and Link (BL), and Branch and Exchange (BX). In ARM mode, branches specify a 24-bit offset relative to the value that PC will have when the branch is in the execute stage of the pipeline. The offset is shifted left by two bits, then added to PC (which is the address of the branch plus 8) to get the target. In THUMB mode, the offset is shifted left 1 bit and added to the PC (which is the address of the branch plus 4).

6.4.1 Branch and Link

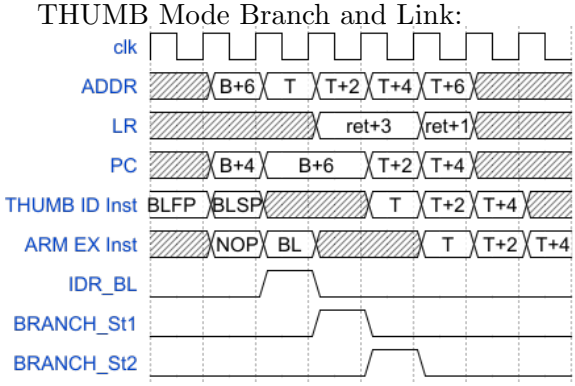
In ARM mode, Branch and Link instructions in ARM operate the same as normal branches, but store the return address (current PC plus 4) in R14. In THUMB mode, branch and link instructions occur over two instructions. The first instruction in the BL causes the bottom 11 bits of the instruction to be stored in the `ThBL_Reg` register in `ThumbDecoder.vhd`. The second instruction's offset minus 1, shifted left 1 bit is added to the value in `ThBL_Reg` shifted left by 12 bits to get the BL offset, which is passed to the ARM decoder as a BL instruction. The first instruction in a THUMB BL is passed as a NOP to the ARM decoder. The ARM decoder doesn't shift Branch instructions in THUMB mode - the shifter control treats it as a default case. When we started work on the processor, two signals `ThBLFP` (THUMB BL First Part) and `ThBLSP` (THUMB BL Second Part) were outputs of the THUMB Decoder to account for BL in THUMB mode, but it was easier (and required fewer deep dives into the details of the processor) to write custom logic to handle this case. Timing diagrams are below: **B** denotes the branch instruction (or address) and **T** denotes the branch target instruction (or address). `IDR_BL`, `BRANCH_St1`, and `BRANCH_St2`

are signals in `ControlLogic.vhd`. `PipelineRefilling`, another signal in `ControlLogic.vhd`, is asserted when either `BRANCH_St1` or `BRANCH_St2` are asserted. In THUMB mode, the lowest bit of the return address is set - see Branch and Exchange.

ARM Mode Branch and Link:



THUMB Mode Branch and Link:



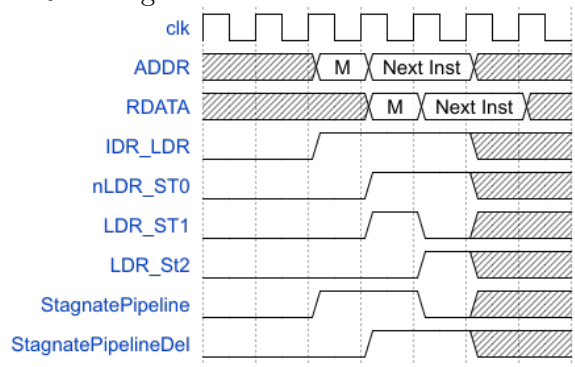
6.4.2 Branch and Exchange

Branch and Exchange is the same between ARM and THUMB mode, except when the register being branched to is R15. ARM discourages this use, so, assume that BX is the same between modes. Branch and Exchange acts the same as a normal branch, with the same timings as detailed above (with `IDR_BX` instead of `IDR_BL`), except that the CPU mode is set based on the lowest bit of the register being branched to - THUMB mode if the lowest bit is 1, ARM mode otherwise.

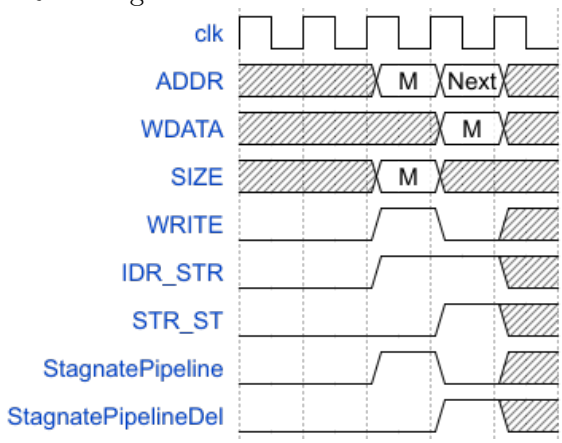
6.5 Load/Store Timings

For the sake of clarity, below are access timings for a LDR (load register) and STR (store register) instruction. Both are good example of multi-cycle execute instructions. **M** refers to the load/store access, **Next** refers to the next instruction.

LDR Timing:



STR Timing:



6.6 Errata

Most fixes to the CPU that made it into the final implementation were done as cleanly as possible and should not be cause for future bugs. One fix, however, needed to be done less cleanly since time was running out in the semester. So, as a warning, the logic for `SngMltSel` - the select line on the output multiplexer in the Load Store Address Generator (`LSAdrGen.vhd`) - may be incorrect. The problem was that an STM after an LDR would have the wrong address, the fix was to add "and (not IDC_STM)" to the end of the condition for `SngMltSel`. This fix felt like it might cause the instruction sequence STM, LDR to break the system, but that might just be paranoia. Either way, be careful. Additionally, the logic for `AdrIncStep` in `ControlLogic.vhd` caused 7 bugs. It seems to be working, but if it needs to be fixed in the future, be extremely careful - make sure that all the cases (LDM, STM, BX, etc.) are accounted for.

6.7 CPU Interface

The ARM7TDMI core that we got didn't have all the signals that are described in the ARM7TDMI specification. In fact, most of the signals were missing. The ARM7TDMI was a fairly popular processor, designed for a much more general purpose than the GBA, so many of the signals were unnecessary for our context. Namely, the JTAG debug interface (the D in TDMI), and the EmbeddedICE macrocell (the I in TDMI) were intended for post-silicon verification - a process that is wholly unnecessary for our project. The processor also allows for a choice between unidirectional and bidirectional data buses, which we also don't need - we picked unidirectional and stuck with it. We also don't use the coprocessor interface - it was used in the GBA to provide backwards compatibility with Game Boy Color games (interfacing with a Z80 processor). All in all, the signals that we actually need for our interface is an aggressive subset of the signals specified in the ARM7TDMI documentation - if the core we got online didn't have it and we could justify not including it, we didn't. We lose cycle accuracy, but that was never a primary goal of the project and it seems easier to adhere to approximate cycle timings in the context of our system than try to replicate cycle timings that we don't have access to. Again, the primary issue due to our lack of cycle accuracy is memory accesses occurring faster than they should, but we can add wait timings to try to replicate the original timings (or to get closer to them).

The interface from the CPU to the rest of the system is as follows:

```
input  logic CLK,           // The system clock
input  logic PAUSE,         // Pause the CPU when high (for wait states)
input  logic DMAActive      // Pauses CPU and sets memory output bus signals to 'Z
input  logic nRESET,      // System reset

-- Interrupts
input  logic nIRQ           // Interrupt Request

-- Memory interface
output logic [31:0] ADDR,  // Memory Address
output logic [31:0] WDATA,// Write Data
input  logic [31:0] RDATA,  // Read Data
input  logic ABORT,        // ABORT memory access
output logic WRITE,         // Write enable
output logic [1:0] SIZE,    // Memory write size

-- Status Signals
output logic [4:0] mode         // Current execution mode of the CPU
output logic       preemptable  // If the CPU can be paused (so pauses are synchronous to the
```

The module header for the core also includes a `FIQ` signal for fast interrupts, but the GBA doesn't use these. The signal's integrated in the logic for the core - it seemed like an unnecessary amount of work to remove it.

6.8 Testing

We put a fair amount of work into getting some sort of automated testing infrastructure for the CPU in simulation. This involved running a similar core to the ARM7TDMI in QEMU, attaching GDB to the QEMU process, printing out registers every cycle, and writing the output to a text file. The testbench for the CPU would also write register states to a text file every cycle. Both these files would be processed to give a list of register value changes, then the two resulting files would be diffed.

This was helpful - it gave a decent level overview of where output differed, but took several days to setup and didn't end up being all that useful. For the most part, a bug in the CPU would either cause an infinite loop or the CPU to enter an undefined state, so looking for the first register value that differed wasn't needed to track down cause. Additionally, PC in the CPU always differs from PC in the emulation (since PC in the CPU actually points to two instructions ahead of the current instruction).

In the end, we exclusively used NO$GBA. There weren't any bugs where the CPU followed the same control flow and gave the wrong answer for some calculation, so doing a diff of register states by hand was enough. This was also after we moved the CPU onto the board - it took about 5 minutes to run a simulation of 135000 instructions of the CPU - there weren't a lot of other options.

6.9 Interrupts

The CPU has a single interrupt request line, but can receive interrupts from 14 different sources. This is managed via the Interrupt Controller, along with a couple MMIO registers. The CPU's

nIRQ line is set to the OR of all incoming interrupts that have not been masked (masking in Interrupt Master Enable and Interrupt enable registers). When an interrupt is received, the CPU enters nIRQ mode, then examines the Interrupt Request (IF) MMIO register to determine the source of the interrupt. Interrupt priority is handled by software [2].

An interrupt can be generated by the following sources:

- GamePak (IF[13])
- Key Press (IF[12])
- DMA0-3 Transfer (IF[11-8])
- Link Cable Communication (IF[7])
- Timer Overflow (IF[6-3])
- V Count match (IF[2])
- HBLANK (IF[1])
- VBLANK (IF[0])

Since our final design did not include a GamePak, we never looked into details of how GamePak interrupts are implemented, where interrupts are routing from, etc. Presumably, game specific hardware in the cartridge (like the real-time game clock in Pokemon Games) could interrupt the CPU, but we don't have a basis for this. Future groups should definitely look into this if they plan to implement game cartridges.

As a note, if the `nIRQ` line is held low for too long, the CPU will enter into an undefined state. Our solution was to hold the line low until the CPU entered IRQ mode (shown by a change on the `Mode` output of the CPU), more elegant solutions likely exist. In any case, our interrupt controller likely has bugs - the BIOS only needs to generate interrupts on VBLANK to run; many interactions were untested.

7 DMA

The GBA has 4 DMA channels numbered 0 to 3. Only one DMA channel can be operating at once, so DMA channel 0 has the highest priority, 1 has the second, and so on. When a higher priority channel preempts a lower priority channel, the lower priority channel is paused until the higher priority channel completes. Preemption of a lower priority channel can not actually happen until it has finished its current read write sequence. As mentioned before the DMA has priority of the memory bus and can pause the CPU, however preemption of the CPU can only happen synchronous to the instruction stream.

Individual DMA channels differ slightly in expected use. DMA0, naturally, is used for high priority transfers (feeding graphics during HBLANK) and cannot be used to transfer from GamePak ROM, DMA1 and DMA2 can be used to feed DMA sound channels, and DMA3 can be used to transfer data from GamePak ROM.

For each DMA, the source address is specified in MMIO `REG_DMA*SAD`, the destination address in `REG_DMA*DAD`, the number of words or half-words to transfer in `REG_DMA*CNT_L[13:0]`. General controls are set in `REG_DMA*CNT_H[15:0]`, including whether the count is the number of words or half-words, whether the transfer will repeat, when the DMA will start, enables, and whether the DMA will generate a CPU interrupt upon transfer completion [2].

8 Graphics

The Gameboy Advance graphics system consists of four major parts. They are the background processing circuit, object processing circuit, priority evaluation circuit, and special effects circuit. The Gameboy advance supports display of upto four backgrounds and 128 objects simultaneously.

At a high level, the background processing circuit renders each background in the frame pixel-by-pixel. The object processing circuit uses a row buffer to render objects line-by-line. The priority evaluation circuit determines which background or object to actually display at each pixel. The special effects circuit mixes colors to perform effects such as alpha blending. The final color for each pixel is streamed out. One pixel color is produced every four clock cycles. In our implementation we store the final color values into a BRAM double buffer to interface between the GBA clock and the VGA clock driving the display circuitry.

Graphics processing is controlled through MMIO registers. Chief among these control registers is DISPCNT, which is the single most important MMIO register in the console. DIS contains master flags to enable/disable each backgrounds, all objects, each window, the memory mapping mode for objects, whether to process objects during hblank, which frame to display for bitmapped backgrounds, and operation mode for backgrounds. Other important MMIO registers for graphics include the background control registers (BGXCNT), which control displaying each background, BGXHO,VOFS, which control horizontal and vertical offset for a background, and WINXH,V, WININ, WINOUT which control the windows.

8.1 Graphics Memory

There are three major memories used by the graphics system. They are VRAM, OAM, and Palette RAM. VRAM is a 96KB segment subdivided into three regions: A, B, and C. VRAM A contains 64KB of data and is used exclusively by the background processing circuit. VRAM B is used primarily by the object processing circuit to contain object data, though it may be used by the background processing circuit for additional space for bitmapped backgrounds in some modes. VRAM C is used exclusively by the object processing circuit. Both VRAM B and C contain 16KB of data. OAM is 1KB, and contains attribute data, such as location and size, for 128 sprites, as well as upto 32 sets of parameters for object rotation/scaling. Palette RAM is subdivided into two blocks of 512 bytes each, so that objects and backgrounds may have entirely independent color palettes. All of these memories are implemented as dual-ported BRAMs in our implementation, with the exception of VRAM A. One of the ports is exposed to the memory controller, to allow the processor/DMA engine access. The other port to each memory is reserved for the exclusive use of the graphics processing system. This allows us to avoid time-multiplexing access to the memories, which would make life way harder. As mentioned, VRAM A is somewhat special. VRAM A is actually duplicated in out system, so that it may expose two read ports to the background processing circuit.

Since all of the memories in our GBA implementation are implemented with a 32-bit word size, many of the memory accesses made by the graphics system to VRAM are not word aligned. To abstract that away, we created a simple memory controller to interface between VRAM and the graphics system. The memory controller accepts unaligned addresses, and rotates the . Thus, the

least significant byte/half-word/word of the data beat always contains the byte/half-word/word requested. That is simply fantastic for simplifying the logic within the graphics processing circuitry. It's lightweight and keeps everything else clean. Win. It also serves to arbitrate access to VRAM B between the background processing circuit. Double win.

Palette RAM stores the color palettes. Essentially, the background processing circuit and the object processing circuit have their own separate 512 byte palette RAMs. Palette RAM contains colors, which are stored as half-words. Each palette RAM can function as either a single color palette of 256 colors, or as 16 palettes of 16 colors each. Actually, they can each function as both simultaneously. There is the additional stipulation that color 0 in every palette represents transparency. More on color palettes later.

8.2 Backgrounds

There are six modes of operation for the background processing circuit. The different modes determine which of the four background layers (BG0-BG3) are actually used, and what type of background each layer is. There are three different types of backgrounds: text, rotation/scaling, and bitmapped.

8.2.1 Bitmapped Backgrounds

Bitmapped backgrounds are the simplest. Each pixel maps directly to a color stored in VRAM. Up to two bitmapped frames may be in use at any time. If two frames are in use, then a flag in the DISPCNT register specifies which frame to display. This allows the system to effectively double buffer bitmapped frames, writing to one frame while displaying the other, and vice versa. In all bitmapped modes, the background processing circuit uses VRAM B for extra memory. Modes that support two frames trade-off either screen-size (160x128 instead of 240x160) or color (256 colors instead of 32678) to fit within the 80KB available. If only a single frame isn used, then the screen can cover the full 240x160 display with 32678 specifiable colors. Since the colors are specified directly in VRAM, bitmapped backgrounds can bypass palette RAM.

8.2.2 Text Backgrounds

Text backgrounds divide the screen into 8x8 tiles. For each tile, there is a two-byte entry in the screen data section of VRAM A that specifies a particular character to be drawn in that tile. The screen data for each background layer begins at the screen data base block specified in the background's control register. There are 32 base blocks, each spaced 2KB apart, though data for a background may span multiple blocks. The character name obtained for a particular 8x8 tile is used to map into the character data section of VRAM A. Like the screen data section, the base address for the character data is specified in the background control block. There are four character data base blocks, spaced 16KB apart. Each character consists of a single palette index for each pixel in the character. The pixel will then be colored with whatever color is stored at that index in the color palette.

Text backgrounds may use either a single 256 color palette, or 16 palettes of 16 colors each. If the background is used in 16 color mode, than the screen data chunk that specifies the character name for the tile will also specify which of the 16 palettes to use (the screen data also specifies

optional flags to flip the character horizontally or vertically). Note that while eight bits are needed to store the palette index of each pixel in a character in 256 color mode, only four are needed in 16 color mode. In this way, the size of a character is dependent on the color mode.

Text background screens may range in size from 256x256 to 512x512. Those are all larger than the actual display screen. The size refers to the size of the virtual screen. Essentially, only a part opf each background is displayed on the screen at once. This allows screens to be easily scrolled by changing the offset of the display screen within the virtual screen. These offsets are controlled by MMIO registers, as with literally everything. If the screen location overflows, it may optionally wrap around or be padded with transparencies. Backgrounds may also be mosaiced, horizontally and/or vertically. Mosaicing undersamples the background, resulting in a block-ier image.

8.2.3 Rotation Backgrounds

Rotation backgrounds are extremely similar to text backgrounds. The major difference being self-evident. Rotation is specified by four parameters. DX, DMX, DY, and DMY. (Well, actually.. they are A, B, C, and D. I'm not kidding.) These parameters are set in MMIO registers for each rotation background. Together, these four parameters describe how offsets within the display screen translate to offsets in the virtual screen. DX and DY are the horizontal and vertical displacements respectively in the virtual screen from a single pixel horizontal offset in the display screen. Similarly, DMX and DMY are the displacements caused by stepping to the next row in the display screen. This is an elegant solution to the problem of doing the necessary trigonometry to rotate objects in hardware: make the software do it. The coordinates in the virtual screen of the upper left corner of the display screen are set in MMIO registers to serve as a reference point for the rest of the image.

Once the display screen pixels have been translated to locations in the virtual screen, the same process as for test backgrounds is used to lookup palette information. Notable differences: Only 256 color mode is support for rotation objects, characters may not be horizontally/vertically flipped, and only 256 characters may be used, down from 1024 for text backgrounds.

We implement the background processing circuit as a three stage pipeline. The background processing circuit streams across the display screen row by row, computing a full row of the display area during the active time of each hblank cycle. At the end of hblank, the background processing circuit steps to the next row. The background processing circuit can process one pixel every four clock cycles. The three stage pipeline reflects the latency cost of two sequential accesses to VRAM necessitated by looking up the screen data before the character data for text and rotation backgrounds.

8.2.4 Background Pipeline

The first step in the pipeline are the row/col/bgno counters. Row and col intuitively enough are the row and column of the display screen pixel being processed. Bgno on the other hand keeps track of which background layer is being processed at that location. Bgno is also the source of the four clock cycle throughput of the circuit. One background layer is processed each clock cycle. (Also relevant that 4 clock cycles is minimum required throughput to keep up with the display).

Having figured out which layer is to be processed, we decode the MMIO registers, chiefly that layers BGXCNT register for control points to our data path. The row/col position is fed into

both the bg_scrolling_unit, and the rotation_scaling_unit, the outputs of which are selected between by mux. This gives us the location in the virtual screen. Or not quite. Still we (may) need to mosaic the image, so the computed location is fed into the mosaic_processing_unit which computes the mosaic'ed location, if enabled. Now that the actual location in the virtual screen has been obtained, we are ready to look up the screen data. The screen_lookup_unit converts the virtual screen location to an address in VRAM A. This marks the end of the first stage of the pipeline.

Before going further, I'd like to throw in a note about the operation of the rotation_scaling_unit. The rotation_scaling_unit is implemented as a set of four accumulators, and some adders. The accumulators sum the total displacements in the virtual screen due to the display screen location. They simply step by the values specified by DX, DMX, DY, and DMY whenever enabled. The DX and DY accumulators step by DX and DY each each time the display screen column is stepped, and clear at the end of each row. The DMX and DMY accumulators step at the end of each row, and clear at the end of each frame. A separate overflow_handler signals to the rest of the circuitry if the final location is outside of the virtual screen.

Cool. Back to the screen_lookup_unit. So we found the address of the screen data and waited one clock cycle (yay pipelining) to actually get the screen data. The screen data is passed into the char_data_lookup module to figure out the address of the character. More pipelining happens, and we get the character data we need. That data, along with a large mess of control signals and flags gets fed into the data formatter. Actually I lied. That data is muxed against the color obtained from the bitmapped background logic (heretofore not described. I didn't forget it, promise), and the muxed result goes into the data formatter. The data formatter was not one of my better decisions. It was probably one of my worse decisions. Essentially, the data formatter packs all of the information that the priority evaluation circuit could possibly want to know about the background into a single bitvector. Why didn't I just output each signal separately? I don't know. I should have. Random bitvectors are problems. They make the design harder to understand, harder for other people to understand, harder to debug, harder to bring in help when needed. Don't do that. Seriously. At least it converts the palette index and palette . Anyway, the bitvector contains information such as whether the background is visible at the current pixel, the background's priority, the index into palette RAM to use, if it's bitmapped, and so on and so forth.

The logic for bitmapped backgrounds is very simple. The bitmap address unit computes the address of the current pixel in VRAM. Then, pipelining happens, and the color and control signals arrive at the data formatter a couple of clock cycles later. Yeah, bitmapped isn't very hard.

8.3 Objects

Time to switch gears to objects. Objects are what other systems think of as sprites. They are small(-ish) objects that may overlaid onto the display. Each object is contained within a rectangular bounding box. Objects may range in size from 8x8 to 64x64, and need not be square. An optional double size flag may be set to extend the size of the bounding box up to 128x128. Objects may be scaled and/or rotated about their centerpoint. Rotation/scaling for objects is set in the same way as for backgrounds, by specifying DX, DMX, DY, and DMY.

The GBA renders objects through a row buffer system. While each line is being displayed, the object processing circuit is calculating the location and colors of the objects on the next line. The number of objects that may be displayed on each line is limited by the number of clock cycles per

line. Usually there are 1232 clock cycles of rendering time per line, corresponding to the 240 pixels per row plus 68 pixels of hblank time. This enables display of upto 128 8x8 objects on a single row. The processing may be shortened to only the active phase of each row (240 pixels, or 960 clock cycles) by the DISPCNT register. If the processing time is shortened, then fewer objects can be displayed. Processing time per object is proportional to the width of the object's bounding box. One clock cycle is required to process each pixel of the object's width.

Each time the object processing circuit advances to a new line, processing beings with object 0. The object processing circuit then iterates one-by-one over every object until it runs out of processing time. If an object's bounding box does not intersect the line being processed, than that object will be skipped, rather than spending the full time to process the object. If the object's bounding box intersects the line being processed, than the full processing time for the object will be used, even if the object is entirely off of the display screen (objects exist within a 512x256 virtual screen). At the end of each line, the object processing circuit stops processing wherever it is to get ready to begin the next line.

The first step in processing each object is to lookup the object in OAM (object attribute memory). Objects are stored as 48-bit structures aligned to 64-bit boundaries in OAM. Objects are stored in strictly increasing order, so object 0 is stored at offset 0 within OAM, and so on. The remaining storage space is used to save the object rotation/scaling parameters. Up to 128 objects and 32 sets of object rotation/scaling parameters may be stored in OAM. Object attributes specify control information for each object, such as size, shape, whether they are to be rotated/scaled, which set of rotation/scaling parameters to use, whether the object should be double-sized, the location to draw the object, the character name of the object, type of color palette to use, etc...

Having looked up all of the control information for the object, the object processing circuit then does a quick spot check to see if the object's bounding box intersects the current row. If it does not intersect, than the object is not visible on the current row and can be skipped. If the object does intersect, than the next step is to check if the object is enabled for rotation/scaling. If the object is rotation scaling, than the the four rotation scaling parameters are looked up in OAM. This lookup requires four clock cycles, since each parameter is strided 64 bits, so only parameter can be looked up per clock cycle.

Once parameter lookup has completed (if needed), than the object can begin to be processed. In our implementation, the object processing circuit steps horizontally across each pixel location in the bounding box that appears on the current line. For each pixel, the pixel is mapped to the location of its preimage within the bounding box, accounting for effects such as rotation, horizontal/vertical flip, and mosaicing. If the pixel's preimage is outside of the bounding box, than the pixel will be transparent. The preimage is then mapped to VRAM to lookup the palette index for that pixel.

As with text and rotation background, objects are specified by 8x8 tiles in VRAM B/C, where each tile specifies a palette index for each pixel. The locartion of each tile in VRAM may be computed in either one-dimensional mode or two-dimensional mode, as laid out in DISPCNT. one-dimensional stores each tile of object contiguously in memory in row-major order. Two-dimensional mode lays out each row of tiles in an object contiguosly, but strides the starting address of each row by 1KB. Two-dimensional addressing basically envisions VRAM as a two-dimensional array.

We implemented a simple FSM to control the object processing circuit, governing when to lookup a new object, when to lookup rotation/scaling parameters, and when to step across the object. Rotation/scaling of objects proved to be difficult to implement. We tried to implement the calculations using multipliers, but wound up having difficulties detecting when the rotation/scaling calculation overflowed the bounding box. I imagine Nintendo found a much more clever way to handle this, probably more along the lines of what we did to implement background rotation/scaling. Improper handling of rotation/scaling objects was probably our biggest bug on demo day. To any future groups, make sure your implementation of rotation/scaling objects is rock solid.

Our object processing circuit was not heavily pipelined, only requiring pipeline for the memory accesses to VRAM. The rest of the circuitry was essentially combinational to map pixel locations on the display screen to locations within the bounding box to locations in VRAM. Object processing at it's heart is really dominated by the control FSM. Make sure the control FSM is solid, and 90% of the object processing falls into place. The remaining work is dominated by the rotation/scaling calculations, which are hard. The rest of it is simple and straightforward. We did run into problems with our control fsm, in particular the timing of starting a new line. For object lookups we required two cycles to read the attributes from OAM, since our OAM was implemented with 32-bit words (64 bit words make so much more sense for OAM, to anyone doing this project in the future, give OAM a 64-bit read port, it makes the object processing FSM so simple).

The priority evaluation circuit is responsible for determining which color actually shows up at each location in the display screen. Like the background processing circuit, it streams across the display area pixel-by-pixel, row-by-row. It spends four clock cycles processing each pixel. On each clock edge, it receives the palette and control information from one of the four background layers. Only one set of palette information is received from the object processing circuit, and it is valid for all four clock cycles of processing. The object processing circuit only needs to send on one set of data, since the GBA only allows one object layer, and the object processing circuit breaks priority ties by always preferring lower numbered objects. Each background and object is tagged with a two-bit priority. 2'b00 represents the highest priority. Whenever an object and background have equal priority, the object takes priority. The final color of a pixel is the given by the palette information (or bitmapped color) of the highest priority layer that is non-transparent at that pixel.

We implement the object processing circuit as a two register king-of-the-hill system. The TOP register stores control information about the highest priority layer received thus far, while the BOT registers stores the second highest priority layer received. The registers are cleared for each new pixel. We store two layers for the special effects cricuits. On the last two clock cycles of priority evaluation, the priority evaluation circuit looks up color information of the highest priority layer for which the color has not yet been looked up. The top two colors are passed along to the special effects circuit.

The priority evaluation circuit also handles windowing. Windows in the GBA allow for display of specified layers to be disabled in certain areas. The GBA supports four windowing regions: WIN0, WIN1, WINOBJ, and WINOUT. WIN0 and WIN1 are rectangular regions with their location and size specified in MMIO registers. Actually, they are not necessarily rectangular, only rectangular if the bottom right corner of the window is actually below and to the right of the upper left corner. The selection region is inverted if the bottom right corner above and/or left of the top left corner. WINOBJ is specified as the collection of pixels which are covered by non-transparent objects in obj window mode (one of three object modes). All obj window objects are entirely trans-

parent, but the palette information is used to select (or not) pixels for WINOBJ. WINOUT is the region outside of the three other windows. Each window may be disabled through DISPCNT. The priority evaluation circuit accounts for windowing in determining which layers are non-transparent at each location.

That's our implementation. But it's wrong in two ways at an architectural level. First, the special effects circuit may blend colors between non-consecutive layers (The BIOS does this actually. %$&÷) So, color information for all five layers (four background layers + one object layer) must be stored and passed along to the special effects circuit. Secondly, layers that are disabled through windowing may still be used for special effects processing (Again, the BIOS does this. %$*&.). We simply throw them away. Not great.

8.4 Special Effects

The special effects circuit is actually very simple. Special effects allow for three types of effects: alpha blending, fade-in, and fade-out. Actually, all three of these are essentially the same computation. They are simply a weighted average between one color and another. Weights are specified in MMIO registers (surpise). In alpha blending, the colors are taken from two layers, for example background 0 and background 3. If the first target layer is a semi-transparent object (one of the three object modes), than alpha blending is always performed. For fade-in / fade-out, the second color is either white or black. Blending weights range from 0 to 1, inclusive, in steps of 1/16.

The special effects circuit produces the final color for each pixel on the display screen. These colors are then written into our frame double-buffer, to be displayed by the VGA. The frame buffer consists of two memories of 240x160 2-byte words each. One memory is read from by VGA, while the other is written to by the graphics system. The buffers switch each frame. The VGA and graphics system are kept in essentially lockstep since the vga_clock is triple the rate of the gba clock, and the VGA takes triple the number of clock cycles to display a frame as the GBA requires to write one. Thus, the VGA and GBA sync up once per frame at the same location each time. This prevents the display memory from ever being changed mid-frame.

8.5 VGA Display

Natively, the GBA expects a 240x160 resolution - a 3:2 aspect ratio. This doesn't exist in modern monitors. Originally, we had planned to output 480x640 VGA, but the GBA documentation didn't give us pulse width timings. In the end, we decided to double buffer the display, storing the double buffer in the Zedboard's BRAM. The graphics pipeline writes the color data that will be output to the double buffer controller and asserts that the buffers should switch at the end of a display cycle (`toggle`). The VGA controller presents the buffer address for the next pixel (since reads are sequential from BRAM) and writes the data read from the buffer to `VGA_R[3:0], VGA_B[3:0], VGA_G[3:0]`. The VGA is clocked at three times the system clock (16.78Mhz), so that both the VGA controller and the graphics pipeline stay coordinated on a per-frame basis.

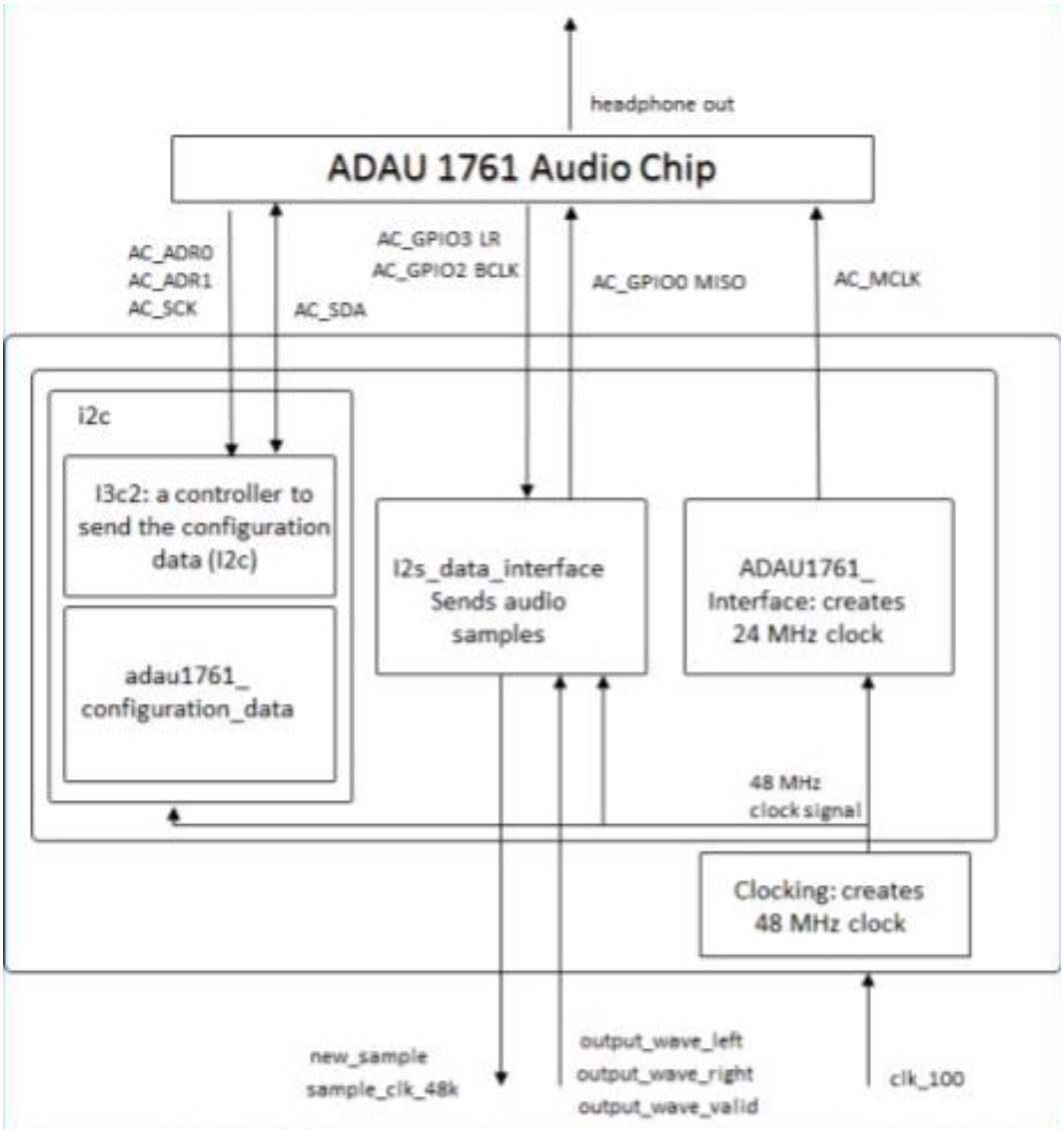
9 Audio

The GBA has four wave sound channels and two DMA sound channels. The wave channels are identical to the Game Boy Color, with each channel having slightly capabilities. Channel 1 generates a square wave tone with optional frequency sweep, channel 2 generates a rectangular tone, channel 3

generates a sawtooth wave, and channel 4 generates white noise. The frequency sweep in channel 1 is not implemented in our design, because it requires updating the SOUND1CNT_X with new updated frequency values. None of the games in our demo used a frequency sweep, so this piece of integration was not something that took priority. Channel 1 and 2 are implemented with a square wave module that generates the square wave, the square wave then gets passed through a length counter that cuts stops the square wave at the given length. Lastly the square wave gets passed through a volume envelope, that changes the volume at a specific rate. Channel 3 is implemented with a wave channel, that reads the sound data from WAVE_RAM0-3 and then sends it through a length counter before sending it to the sound circuit. Lastly, channel 4 creates a noise wave using a linear feedback shift register (LFSR) to generate a pseudo-random bit sequence. The noise channel then passes through a volume envelope and length counter. Each of the 4 channels above also has a clock divider, because the volume envelope is clocked at 64 MHz and the length counter is clocked at 256 MHz.

Direct sound was a new feature added to the GameBoy Advance to add more sound control, over simply mixing together 4 channels of sound. Games could now put streams of 8 bit sound data into memory, and schedule it to play without any more CPU interaction. The two direct sound channels can be configured to use any combination of the two timers, and two DMA channels. The given timer is set to choose the number of cycles between samples. On every timer overflow the audio data is passed from a FIFO to the mixer. When the FIFO has only 4 words left, a request will be made to the DMA channel to transfer four more words of data to the FIFO. The FIFO has a maximum depth of 8 32-bit data and is written to directly when the DMA channel writes to the Sound FIFO Input MMIO register. The mixer varies the volume, and ratio of direct sound to 4-channel sound, then outputs to the sound circuit. [2].

9.1 Zedboard Audio Codec



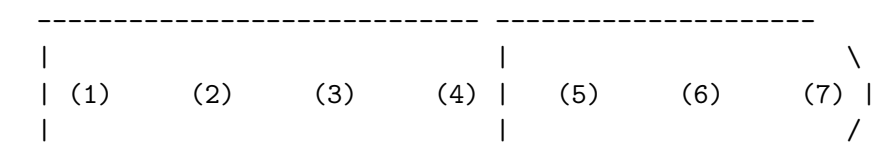
The sound circuit[1] uses an I2C interface to configure the ADAU1761 Audio Chip on the Zedboard. The audio is then transmitted to the chip using I2S. The sampling rate at which data is passed to the audio chip is 48KHz.

10 Timers

The Game Boy Advance, per specification, has 4 16-bit timers that are programmed via registers TM0-3CNT_L,H. These timers can be programmed to count at the system clock rate, or at 1/64th, 1/256th, or 1/1024th of the system clock rate. Each timer can also generate an interrupt when the timer count overflows the 16 bit TM0-3CNT_L register. The counter just changes its current value in the given MMIO register at the specified rate. If interrupts are not enabled, other modules have to poll the register to see when the timer has overflowed. Multiple timer registers can be added together to allow for longer timing delays by setting count-up timing, which only allows timer X+1 to start counting after timer_X completed.

11 Controller

We use an SNES controller connected to one of the Zedboard's PMOD ports. The controller is connected as follows:



Pin	Description	Zedboard Connection
1	+5v	Zedboard +5V
2	Data clock	JA3
3	Data latch	JA2
4	Serial data	JA1
5	? (+5v)	no wire
6	? (+5v)	no wire
7	Ground	JA GND

As detailed above, the Zedboard expects a +5V VDD, whereas the PMOD connectors output at +3.3V. To avoid frying the Zedboard, we used a *Sparkfun Level Converter*. As a note, to properly use the level converter, ground on either side should be connected to ground on the Zedboard (either one of the ground pins, or ground on one of the PMOD connectors), +:q3.3V should come from the PMOD connector, and +5V should come from the 5V pin on the Zedboard.

The serial protocol between the controller module on the Zedboard and the SNES controller begins with the CPU sending a 12us wide positive pulse on the Data Latch pin. The CPU wait 6us, then sends 16 data clock pulses, beginning with the negative edge of the clock, that are 50% duty cycle with a 12us period on Data Clock (Data Clock is high when idle). The Controller shifts latched button states on the positive edge of the Data Clock, and the CPU samples the Serial Data line for button state on the negative edge. Button state is reported active-low. In our implementation, button state is reported in the order below:

Clock Cycle	Button Reported
1	B
2	Y
3	Select
4	Start
5	Up on joypad
6	Down on joypad
7	Left on joypad
8	Right on joypad
9	A
10	X
11	L

```
12              R
13              none (always high)
14              none (always high)
15              none (always high)
16              none (always high)
```

The serial protocol repeats at 60Hz. The GBA buttons register expects (from low to high) {A, B, Select, Start, Right, Left, Up, Down, R, L}, so we reorder the outputs in our controller module.

11.1 Status & Future Work

At demo day, our system ran the BIOS and booted one of two different ROMs: *WolfARMStien* - a 3-D maze to explore in the style of Wolfenstien (written by an outside source), and *Nintendo Pong* - a version of Pong with Mario and Luigi as paddles and a Pokeball as the ball. The system ran without any processing bugs, however the visuals were still a bit glitchy. Everything was, however, deterministically glitchy which means our system was stable, if nothing else. DMA sound played for the BIOS, but was a bit choppy. *Nintendo Pong* used 4 channel sound, which was identical to an emulator.

12 Vivado Tips and Tricks

Unpopular opinion time: Vivado is a pretty good piece of software. Synthesis runs can be upwards of 20 minutes for full system implementation, but considering that an entire GBA is being synthesized, this is perfectly reasonable. However, it can be quirky in some cases. Like VCS, Vivado is the only program for compiling to the Zedboard, so many features aren't as polished as they could be. For instance, disconnecting the programming cable while an ILA is active will cause Vivado to segfault - fun for a security researcher, less fun for computer engineers. Like it or not, though, Vivado is what every 545 group will likely be using. With that in mind, a few tips:

ILAs can either be created by selecting nets after synthesizing a design (in "Synthesized Design") or by using an ILA IP block. The latter will have faster compilation times for implementation, but will required re-generating the IP block if the number of signals or width of signals changes. It also needs to be wired up by hand, which can make debugging large numbers of signals tedious. Selecting nets in a synthesized design gives much more flexibility in choosing debug nets. As a note: nets in "Synthesized Design" will not always (and will often not be) named the same as in the source code. To guarantee consistent name resolution, prefix a signal declaration for a signal that will be debugged with "(* mark_debug = "true")" in System Verilog, or add the line "attribute mark_debug of ¡signal_name¿ : signal is "true";" in VHDL (with "attribute mark_debug : string;" at the top of the VHDL file). We recommend this method - signals will be auto-selected for debug cores and can be removed on a per-run basis. During integration, we regularly had 1500+ signals in an ILA so we could work on multiple bugs per synthesis run.

We had issues using Block Memory Generators with 32-bit address interfaces - mappings to data weren't working as they should. It's easier to just have address lines only as wide as necessary for the memory and add some extra logic.

Vivado is pretty bad with source control and transferring projects between systems, especially

with IP cores. As a "nuclear option" an entire project can be exported as a ZIP archive and reopened on another machine. Adding all the files in a Vivado project to Git is an option, but a bad one - commits will give thousands of lines over dozens of files. Vivado also dumps a bunch of files in the directory that it's run in (Vivado_*, webtalk_*, etc.). These can be safely deleted, but they're a bit annoying.

Vivado is also a bit weird with inferring clock domains, as well as inferring which signals are clocks. If possible, never implement clock dividers from scratch, just use the Clock Generator IP block. Also, if possible, only use one Clock Generator for all clocks - switching to this fixed a couple non-deterministic bugs in our system. Only clock registers off of signals coming from the clock generator module.

Many Vivado warnings, like "always_comb does not infer combinational logic" should probably be errors or, at least critical warnings. In fairness, Quartus labels similar things only as warnings. Our advice is this: if a system is buggy, the first step should be to look at Vivado warnings. Every warning from synthesizing a project should be looked into. If finding the source of the warning (or inferred latch) is difficult, VCS sometimes gives better descriptions of where the problem is.

13 Lessons Learned

System integration without a CPU is not really possible, so it is extremely valuable to get a working CPU fast. In order to leave enough time for system integration every part of the project should be completely done well before the deadline. Future teams attempting the GBA project should use our ARM core.

Fixing bugs in a system without understanding everything leads to more bugs, which then leads to fixing the same line of code over and over again.

When starting the project if you are completely lost and everything seems magical, it's probably pretty simple all the way down. Just start with one small piece at a time until the project starts building on itself.

We ran into problems were the spec was ambiguous, so we just made assumptions on what should happen. We would figure out if our assumption was wrong during integration. However, because integration did not happen until a long time afterward, it took some time to remember what assumptions were made.

Testing understand of a spec after reading it with writing custom games is a good idea.

Schedules and Gant Charts are nice for reassuring people that everything will be done on time. They aren't good for much more than - most assumptions about how long things will take will be wrong, especially with a project of this complexity. Having an idea about what will get done in the coming week is good - there isn't any need to plan farther than that in advance.

14 Personal Statements

14.1 Rachel

At the beginning of this project we each had clear cut parts of the system we were each going to work on, but as the class progressed the line became fuzzier. I started the class working on the sound system. I was not familiar with any sound in general and wasn't sure where to start. I found it helpful to look online for other examples on how to use the audio chip to output sound, and ended up using what I found online to program the audio chip. Once I was able to get a sound output on the board (I didn't care what sound it made), the task became less daunting. I highly recommend for future groups working, no matter what part of the system you are working on, try to get some sort of output as soon as possible. Once you have an output you see a clear cut goal on what should be happening and you can more easily work toward that goal.

Next I worked on direct sound, which involved adding more mixers to the system, and sending values from the sound FIFO register. I also created a timer system to sample values in the register at a certain rate. The main documentation that we were following was the programmers reference manual. While the reference provided a good coverage on how to program the system, it did not give any exact details on how everything worked in hardware, so I made a lot of assumptions. For example, when integrating the direct sound at the end of the semester, we found out that the sound FIFO register was not actually supposed to be the FIFO, but actually any value written to the FIFO register should be entered into an actual FIFO. If I was starting the project over I would have started the semester by making a small game in an emulator, and test all of the assumptions I made; instead of waiting until system integration to test assumptions.

After I finished the sound system and tested it to the best of my abilities (it was not possible to test direct sound without the CPU, DMA, and memory controller working), I started working on the SD card interface. The SD card interface was a challenge, because the SD card on a Zedboard is not actually connected to programmable logic. I spent a week learning how to use the Vivado SDK and the basics of AXI to read from the SD card on a button press and send the contents to BRAM. At points like this Google is your best friend, there are lots of example projects and tutorials teaching you the basics, which you can adapt to your project. I thought the SDK was very frustrating, because I was just running the arm core bare metal with the SDK's startup code, but there were times the code would reach a forever while loop before even entering "main." After doing a power cycle of the board and Vivado, all of the sudden my code would work. I am still not sure what was happening, but the Vivado SDK does seem very finicky.

After figuring out the SD card interface, I started working on the DRAM controller. Steven is more familiar with the Vivado SDK and helped me use HLS to create an AXI master that would drive the AXI bus to write to DRAM. I spent a 1.5 weeks poring through documentation trying to make the AXI master work. I removed everything in the project except for the AXI master and tried just writing to BRAM, and even that didn't work. After 1.5 weeks, I got very frustrated, because I was making no progress. I knew that the AXI master and DRAM controller are something that should be very quick to make if I knew what I was doing, so spending so long on it without any results was frustrating. As my frustration grew, I talked to my teammates and asked them if there was something else I could work on, and then someone else can go back to working on the DRAM controller. I am glad that I moved on from the DRAM controller, even though we didn't actually have time to finish it at the end of the semester, because it would have been a waste of time to

continue to spend weeks working on it without any progress. If you feel like you are working on something without progress, there is a time when you should find more help from your teammates, or move on to a different part of the project. At this point it was about the beginning of November, and there was still lots to do, the CPU was still not working, and graphics was really far behind. We all decided that we should get together and just bang out the entire graphics pipeline. It was really helpful that Steven created loads of documentation on what he wanted the graphics pipeline to be, so we could all just code it out even if we did not fully understand the entire pipeline. I think this method was very effective, and would highly recommend others to do the same. If you are feeling one teammate is falling behind on a part of the project they are most familiar with, if they have good documentation everyone else in the group can help speed up the coding process.

Afterward I started working on the DMA, we had a code complete implementation from the beginning of the semester, but we did not have a memory controller to test it with. Testing the DMA with the memory controller was a little frustrating, because no matter how well written code is, it is always a little frustrating to debug code that you did not write yourself. It seems like a lot of time is wasted learning how the code works before any bug can be fixed. I would recommend to all future groups, when possible have the person who wrote the module debug it immediately themselves, and save the unnecessary time waste that comes from someone else debugging the module months later.

Coming into Thanksgiving I was fairly worried, not only had we not even start integration but there were parts of the system that weren't even complete yet. I started working on a small game in an emulator, that only used parts of the system that I knew where tested the most. This way even if the entire system was not complete we would still have a minimum viable product that would demonstrate minimum system integration, and the parts of the project that were complete. The CPU was officially complete right after Thanksgiving, and could run through the entire BIOS correctly. The end of the semester was a mad rush to complete system integration. There were a few design decisions that had to be changed for example we found out that the DMA had to be synchronous to the instruction stream, but overall they were small fixes that were made. And by some miracle, we managed to pull out a mostly working GBA by the demo day.

I think the main advice that I have to any future groups is that if there is a part of the project that you think will take all semester to do (especially if it is something as central as a CPU), a different alternative should probably be found. System integration, and debugging, especially if you want to play entire games, takes a really long time. If you only have time for one week of system integration at the end of the semester, it will be a very bad week.

14.2 Steven

I was responsible for the graphics system of the Gameboy Advance. I designed the entire graphics system from scratch in the first few weeks of the class. Having physical schematics of the entire design proved very helpful later during implementation and debugging. Having designed the graphics system, I did the only logical next step: I wrote the DMA module. I was largely AWOL during October (sorry for that group). I did get the VGA double buffer interface working during that time, and did some consulting on the side trying to get DRAM working. the VGA double buffer got held up for a while debugging a bug that I couldn't sort out. Turns out the Zedboard VGA interface just doesn't deal with solid horizontal bars very well. It's a bug with the board, apparently. In early November I got back to working actively on the graphics system. (Good idea:

design a system, then wait a month to implement it. No. Please no.). That got kick started by the rest of the group donating an afternoon to make graphics code-complete. I'm really grateful for that help. The rest of November and December was a lot of debugging graphics. I started out trying to get backgrounds working in all of their devil incarnations. many bugs were found in the priority evaluation circuit. Many bugs were found to be mistakes in the .coe files I was testing with. Backgrounds were very solid by demo day (The BIOS does not have any -visible- backgrounds). Testing moved onto objects next. I focused on getting normal objects working first. (The BIOS uses rotation/scaling objects). Normal objects worked by demo day. Unfortunately I was not able to finish debugging rotation/scaling objects before demo day. The night before demo day I found a couple of small ROMs that could fit in the remaining BRAM on the board. Their graphics were all exclusively bitmapped backgrounds. Which were untested at that point. So I spent the night before demo day debugging bitmapped backgrounds. Random aside, several older project reports said that only having one person who knew how XYZ worked was a problem for them. I found that being the only person who knew how graphics worked was the only thing that kept me working on this project through the end, instead of trying to not fail my other classes.

This class was incredibly stressful and made for a very unpleasant semester. This project was an enormous time sink, and even with my not-terribly-productive October, I was still easily putting 20+ hours a week on average. The project will always demand more time than you are willing or able to give it. Working relationships in the group were often not very good. Especially when my teammates were frustrated with me for not pulling my weight. And I understand that frustration. We were all extremely invested in this project emotionally and I think we all experienced at least one mental breakdown over the course of the semester. Personally I was stressed out of my mind for the last couple of months of the semester. I felt like I was doing too many things and failing at every single one of them simultaneously. This project did not help with that at all. My other classes, research, and activities all suffered greatly this semester from this project. I felt like all of my dreams had died. The stress culture is very real.

That said, Gameboy advance was a tremendously fun project to work on. If I had not been so over scheduled this semester, it would have been an absolute joy to work on Gameboy advance. So, the lesson is, if you want to do the Gameboy advance, take a light course load. I do highly, highly recommend going after the Gameboy advance. I think that it is absolutely achievable in a semester. We were handicapped by needing to spend three months debugging a CPU that an unemployed Russian guy wrote, and having one group member (me) basically snowed under from classes all semester. A competent and motivated group can definitely put together a polished demo.

Here are my takeaways for future groups Don't leave important stuff in the lab overnight. I was very sad when FMS threw out all of the schematics for the graphics system. Do make copies of any and all important data. I was very happy to still have (several) electronic copies of the schematics. Do read Nintendo's programming manual. That thing is seriously a gem. Do make schematics for your system. A cohesive design is easier to debug. Especially if you can how the entire system fits together at a glance. It also makes it very easy for teammates to bail you out if need be. If you can't draw the schematic for part of the system, you don't understand it well enough. Do test the things that will show up in the demo first. No one cares how well you can rotate backgrounds if your demo doesn't have rotated backgrounds. We changed our target demo to the BIOS a couple of weeks before demo day. None of the graphics functionality used by the BIOS had been tested at that point. Made me really really really sad. Do develop a solid testing suite. The more testing you can do in simulation the better. Synth runs are expensive. Static images are the best for debugging graphics.

Trying to debug rotation/scaling objects as they zoom across the screen in the BIOS is a nightmare.

Here's my suggested division of labor for this project for future groups
1. Use Neil's CPU.
2. Use Neil's CPU.
3. Put person B on graphics. They should do literally nothing else. Only graphics. They should know everything there is to know about graphics. They should implement the object processing circuit first. Then the priority evaluation circuit second. They should design the entire graphics system before doing any implementation. Person C should be involved in the design of the background processing circuit.
4. Put person A on audio, DMA, DRAM, and the SD card. If they have time, they can try to figure out the link cable.
5. Put person C on the memory controller, CPU, and graphics. This person should do the memory controller first. They should do the background processing circuit second. Then they should figure out enough about the CPU to be able to debug it if it breaks.
6. Use Neil's CPU.
7. See above.

14.3 Neil

I was responsible for the CPU, the memory controller, and the SNES controller. I also worked with Rachel to get DMA sound working, as well as wrote part of the object circuit. I did my best throughout the semester to shield my groups members from the more managerial tasks that we had to complete by doing the presentations and reports to the best of my ability, then asking them to add details on modules that I didn't fully understand.

I spent the vast majority of the semester debugging the VHDL ARM core that we found online. I had to switch gears several times to make sure that other parts of the system were moving forward - one week towards mid-semester I took a break from the CPU to write a memory controller for the GBA memory regions that would map to BRAM, as well as hookup the MMIO registers. For the most part, however, I spent a semester debugging a CPU.

To be fair, my hardware debugging skills have never been better, but once you've logged 500 hours in Neil's favorite game ("Find a bug, fix a bug"), it starts to get old. It's especially difficult when any bug could be the one that fixes the system and it could be that there are only a couple bugs left, but it's impossible to say. This makes status updates really fun - being able to tell your group that "we might be really close and we might be two months out and I have no idea which it is, but I'll keep logging 40 hours weeks until it works" is really fun, and just fabulous for team morale.

I was extremely grateful for having TA'd 18240 three times before going into this semester. Finding and fixing bugs in someone else's code is one of the more difficult aspects of hardware engineering and being able to come into this project with some experience was invaluable.

With Steven on graphics and Rachel on sound, I tried to approach the project from the architectural level and ensure that all the individual pieces would fit together at integration time, especially with the GBA's non-standard memory interface and timings. For future groups, I'd highly recommend this - don't just have everyone understand the system, have someone who's job it is to understand

the system, it makes design decisions much easier.

After I finally got the CPU working, I worked on integrating the system and making sure everything fit together on the top level. Debugging was done jointly at the end - to fix a bug that could be in any part of the system, it's important to have everyone there.

This class was a mammoth amount of work. It was a fun mammoth amount of work - it's really great for the 'ol existential dread to throw yourself at a big project and not think about anything else. Burnout is a huge problem, however. Also, the realization that the project won't mean anything after demo day makes it hard to approach the project with purpose. I'll say it here for posterity, but it's standard advice - remember to take breaks. 18545 isn't like a normal class, there are no days off, no assignments to be completed, the entire semester is one big homework that doesn't really end. Approaching it like a homework (working all out on it until it gets done) works for about 5 weeks, then fails miserably. Yoga is good though. Most of the last half of the semester I felt like I was pretty close to some sort of mental breakdown, but a yoga class would push the mental breakdown back a couple days. The post-yoga high is pretty nice too.

This may seem like a bit of doom and gloom, and it is, but the GameBoy Advance was a pretty fun project. The last week where everything worked and bugs were getting fixed in a couple hours was wonderful - the entire system was coming together in front of us. The hard part, for me, was the combined stresses of needing to get the CPU working so that any sort of system integration can happen, realizing that the project doesn't really matter in the end, and trying to keep the rest of my life in some sort of order. I easily worked 40 hours a week on this class alone, probably closer to 60 in peak-debugging season, and close to 90 in the last week (yay?). Again, please use my CPU. Unless you want one member of your group to put several hundred hours into debugging, use my CPU.

This class was enjoyable, but there are several menial tasks (presentations and reports) that are helpful to the course staff, but don't really serve any purpose beyond that. Being told that we were behind was meaningless, being told that we need to address some aspect of our design was meaningless - we couldn't really work any harder or any faster and there wasn't really a chance that anyone would catch a one-off design mistake by listening to one of our presentations or reading our design review. For future groups - the course staff will be some help, but you have the imperative to ensure that your design is correct. If anything design flaw comes up in your design review, status report, or presentation, you have at least six problems, and at least three are serious flaws in your approach to this class. Demos are helpful for reassuring course staff, but again, for us at least, there isn't any way to make us work harder - there aren't any hours left in the day.

Finally, Vivado isn't half as bad as anyone makes it out to be. Embrace the toolchain for the course and it will serve you extremely well.

15 Future Groups

This is a very doable project for a determined group that wants to play Pokemon Ruby or any other fantastic game made for the GBA. Please, please, please, please use our CPU. Not being able to actually start system integration until the CPU was done (which was after Thanksgiving) puts a project on precarious grounds - no one has any idea if anything will actually work until the last week. Read this report several times, we worked hard to give as much useful information as

possible. Then, read Nintendo's Programming manual at least once. GBATek is a good companion to the manual when the manual is less clear. Basically, sit in lab with this report, the Nintendo manual, and GBATek open and stare at everything until it makes sense.

The graphics system needs one dedicated person to make the object circuit (yes, it's that complicated), and half a person to make the background circuit. There isn't a ton of coordination between the two needed (with the exception of priority control and windowing). Whoever does the background circuit should have enough time to do audio as well, and perhaps DMA. The third team member should be able to take our CPU and add a memory controller that handles accesses to BRAM and the DRAM/SD card interaction. Had we been able to put 1.5 team members on graphics, we'd have more confidence about being able to play a commercial cartridge without graphics bugs, but we couldn't spare anyone.

Embrace NO$GBA and DevKitARM. A zip is included our github repository for the "full" version of NO$GBA - it's an extremely useful debugging tool, especially when custom Roms can be made to test specific features. You'll need to find a GBA BIOS and Roms, however.

15.1 Resources

`https://www.manualslib.com/manual/448341/Nintendo-1504166-Game-Boy-Advance-Sp-Edition-Console.html#manual` - Nintendo's manual i.e. the holy grail

`http://problemkaputt.de/gbatek.htm` - A definitive hardware overview

`https://emu-docs.org/Game%20Boy%20Advance/CowBiteSpec.htm` - Another good hardware overview

`http://www.gbadev.org/docs.php` - A collection of documentation, slightly disorganized

`http://devkitpro.org/wiki/Getting_Started/devkitARM` - DevKitARM (GBA compiler/linker) setup

`https://www.reinterpretcast.com/writing-a-game-boy-advance-game` - GBA compiler/linker usage

`https://github.com/mgba-emu/mgba` - mGBA emulator (see emulators)

`http://problemkaputt.de/gba-dev.htm` - No$GBA emulator

`http://www.akkit.org/sGba/compat.html` - Lots of hobbyist games, good for testing (home of WolfARMStien)

`http://belogic.com/gba/` - The Audio Advance: detailed audio system description with C source code for test ROMs

`http://en.pudn.com/downloads168/sourcecode/embed/detail775435_en.html` - Where we got our original CPU

https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf - ARM Architecture Reference

http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/DDI0210B.pdf - ARM7TDMI reference

https://www.gamefaqs.com/snes/916396-super-nintendo/faqs/5395 - SNES controller spec & pinout

References

[1] Martin Korth. *GBATEK. Gameboy Advance / Nintendo DS / DSi - Technical Info*. 2014. URL: http://problemkaputt.de/gbatek.htm.

[2] Nintendo. *AGB Programming Manual*. Version 1.1. Nintendo of America Inc. URL: https://www.manualslib.com/manual/448341/Nintendo-1504166-Game-Boy-Advance-Sp-Edition-Console.html#manual.