RISCBoy Documentation

Luke Wren

March 18, 2019

List of Figures

1 Introduction

RISCBoy is an open source portable games console, designed from scratch:

- An open source CPU, graphics and bus architecture

- Based on the RISC-V open source instruction set

- FPGA synthesis, place and route with icestorm open source FPGA toolchain

- An open source PCB layout

- PCB designed with KiCAD open source PCB software

- It's open source

  If you say open source one more time I'm gonna nut instantly - Oscar Wilde

1.1 Digital Design

Figure 1: System-level block diagram
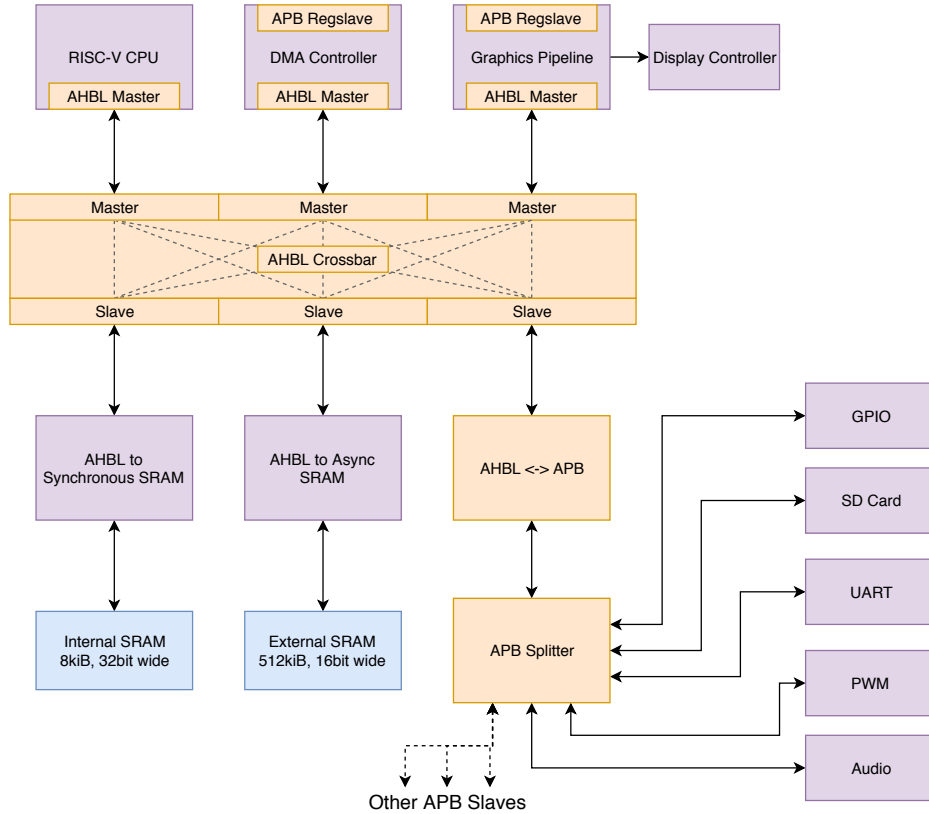
The heart of the design is a Lattice iCE40-HX8k FPGA, containing 7680 LUT4s and flipflops. The logic was designed in synthesisable Verilog, with no dependencies on FPGA vendor IP; the contents of this GitHub repository could be taped out onto a chip. This includes:

- RV32IC-compatible 32-bit CPU design

  - RISC-V instruction set
  - 32I: base integer ISA profile
  - C: compressed instruction extension, for higher code density

- – Vectored interrupts (save/restore of PC, RA only)
- – 5-stage pipeline, similar to textbook RISC
- – Single AHB-Lite master port

- Graphics pipeline

  - – Don't expect much, it's about as powerful as a Gameboy Advance
  - – Includes some MODE7-like functionality which allows drawing perspective-mapped textured planes, by providing per-scanline affine texture transformation. Think MarioKart

- AMBA 3 AHB-Lite compatible multi-master busfabric

- Peripherals:

  - – DMA master
  - – External asynchronous SRAM controller (GS74116 or similar)
  - – Display controller (ILI9341)
  - – GPIO (bitbanging, peripheral muxing)
  - – SD card controller
 - – UART
  - – PWM
  - – Basic audio: voices + samples, noise-shaped PWM output

This document attempts to describe some of these, but if you need nitty-gritty detail, the best documentation is the files ending with `.v`.

That a free synthesis tool can cram this into one of the cheapest FPGAs on the market is tremendous. I hope for a situation like software compilers, where free tools such as GCC and LLVM are industry standards.

1.2   PCB

Figure 2: PCB stackup (left), and BGA-to-via critical clearances and dimensions (right)
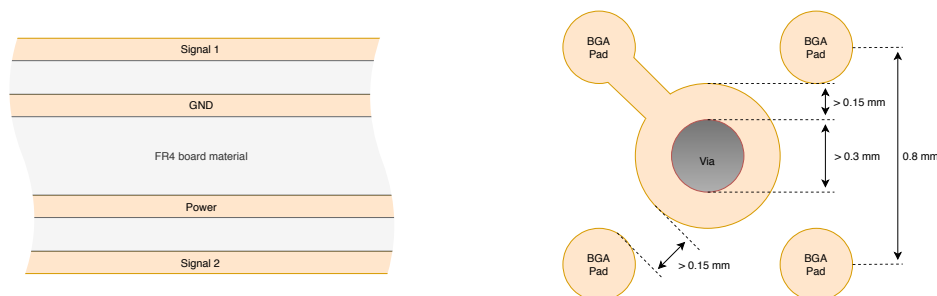
The board has a 4-layer stackup, illustrated in figure 2. It targets low-cost PCB prototyping services such as iTead, and makes compromises to achieve this, chiefly escape routing from the FPGA (11×11 0.8 mm BGA). To meet iTead's copper-clearance, minimum via drill, and minimum annular ring specifications, the BGA pads must be sized down to 0.23 mm. Reflow this component by hand, with a hot air gun and plenty of flux; your toaster oven isn't going to cut it.

Schematic and layout files are in the `board/` subdirectory of the git repository.

1.3 Licensing

The Verilog source to this project has no dependencies, and is distributed under the DWTFPL version 3. This is a *very* permissive open-source licence, and its text is included in full at the top of the source files. This license is very similar to the original DWTFPL, which more readers may be familiar with, but has an added indemnification clause.

This license is also known by its more formal name, as the "Do What The Fuck You Want To And Don't Blame Us Public License".

2 CPU Architecture

Hazard5 is a 32-bit processor based on the RISC-V instruction set architecture. It accesses the system through a single AMBA 3 AHB-Lite master port. Those familiar with the textbook 5-stage RISC pipeline will find Hazard5 mostly straightforward, but hopefully will still find some interesting tricks. We will use the following symbols to refer to the 5 stages:

- `F`: fetch

- `D`: decode

- `X`: execute

- `M`: memory access (load/store)

- `W`: register writeback, fetch address generation

Hazard5 supports the RV32IC instruction set, whose encoding is variable-width. The C extension typically reduces instruction bandwidth by ~25%, which helps to maintain performance when sharing bus access between fetch and load/store.

Branches are speculated, but there is currently no dynamic branch predictor. Instead, we use the static prediction scheme described in the RV ISA manual (based on sign of branch offset).

2.1 Frontend

Figure 3: Hazard5 processor frontend, block diagram
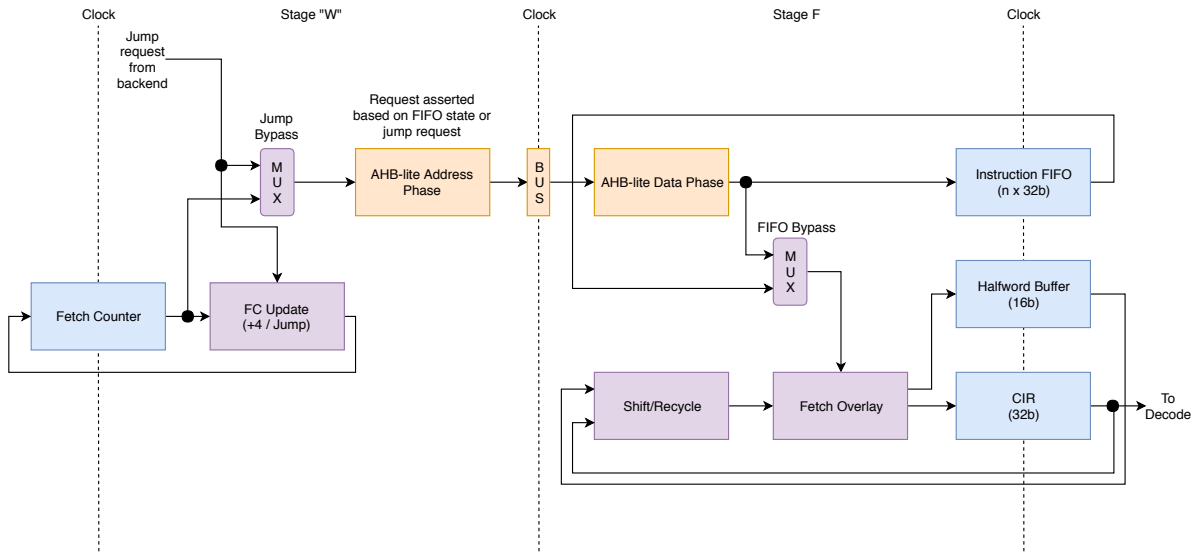
The frontend (figure 3) consists of stage `F` and an additional stage which performs the AHB address phase, and can be considered part of `W`. Its purpose is to feed `D` with instructions, whilst meeting the following constraints:

- No combinatorial path from AHB-Lite data phase to address phase (e.g. `hready` → `htrans`)

- AHB-Lite compliant: no unaligned transfers, no deassertion or change of active requests

- Provide up to 32 bits of instruction data per clock in steady state, even if instructions are unaligned

- 0-cycle jump/flush to AHB address phase assertion (with minimal logic on this path)

- No performance penalty for unaligned jump to 16-bit instruction

- Attempt to maintain performance when competing with the load/store unit and AHB-Lite busmaster peers

The main source of complexity is that a RISC-V *C instruction stream is not naturally-aligned, i.e. instruction address modulo instruction size is not always zero. We spend gates here to optimise the common case of sequential execution, and to lessen the effects of fetch starvation due to load-store activity.

To meet these constraints, the frontend performs almost exclusively word accesses, which must be aligned. The only exception is a jump (or similar, e.g. mispredict recovery) to a non-word-aligned address. In this case, a halfword fetch from the target address is performed.

2.1.1 Prefetch Queue

The frontend queues up fresh instruction data which is waiting to be decoded. The pipelined nature of AHB-Lite means that the bus transfers run ahead of D by at least two clocks, and the prefetch queue is able to buffer these in-flight transfers if D stalls against a later pipe stage. The queue also decouples D's stall logic (which is a function of `hready`) from the address phase request, and finally, the queue helps keep D supplied with instructions while the busmaster is busy with load/stores from X.

There are three parts to the queue:

- A 32-bit FIFO. The depth is configurable, and can be as little as 1 word.

- A halfword buffer which may store the higher-addressed half of a recently-popped FIFO word

- The upper half of the current instruction register (`CIR`), if the previous instruction was 16-bit

These three sources should service the majority of instruction fetches, and fresh bus data is written only to the FIFO. However, following jumps, flushes, or fetch starvation (either due to load/store activity or bus wait states), bus data can be forwarded directly to `CIR`.

2.1.2 Program Counter

Hazard5 does *not* use the program counter (`PC`) for code fetching, during sequential execution. `PC` is used exclusively for the link value in `JAL(R)`, mispredict recovery, and PC-relative addressing; it is physically located in D.

The frontend fetches instruction data from consecutive word-aligned addresses, paced by backpressure from the instruction FIFO; `PC` is not involved. However, as a special case, it *does* need the full jump target address (which becomes the new `PC`), as unaligned jumps require special attention.

2.1.3 Arbitration of Fetch and Load/Store

The single AHB master port asserts transactions from two sources: the frontend, whose address phase is in W, and the load/store unit, whose address phase is in X. Frontend requests may be linear or non-linear (e.g jumps). The rules are:

1. If a jump or mispredict recovery is asserted by M, this wins.

    - Any requests from earlier stages are logically later in program order.
    - If M wants to jump then these instructions are being executed in error, so should certainly not be permitted to access the bus.

2. Else if a load/store is asserted by X, this wins.

    - Stalling instruction fetch *may* be covered by the prefetch queue, in which case we've lost nothing
    - Stalling a load/store will always increase execution time
    - If instead X stalled, and instruction fetch ran ahead, what would we do with the fetched instructions?

3. Otherwise, perform any other access requested by the frontend.

    - Always Be Fetching

The fetch and load/store interfaces are well-decoupled; it would be simple to remove the arbiter and create a 2-master processor configuration. (TODO: add a wrapper that does this!)

2.1.4   Jumps and Branches

Due to the pipelined nature of AHB, we are unable to jump or to take branches in fewer than 2 cycles (without adding sophisticated prediction):

- Cycle 0: AHB data phase for fetch of jump/branch. Next instruction is in address phase concurrently.

- Cycle 1: Jump/branch instruction is now available to `D`

    - (Quickly) use to control the new address phase
    - The immediately following instruction is already in data phase

- Cycle 2: Data phase for jump target instruction

- Cycle 3: Jump target is presented to `D` and decoded.

We knew the jump target on cycle 1, but did not begin decoding the targeted instruction until cycle 3. We also made one wasted code fetch. This is suboptimal, but fetching the jump target *before* decoding the jump is tricky, and there are lower-hanging fruit in terms of performance per LUT.

Jumps physically occur in `W`, directly in front of the fetch address generator. There are two reasons to jump:

- Inspecting the `CIR` in `F/D` pipe register (`JAL`, speculated taken branches)

- Inspecting `X/M` pipe register (`JALR`, branch mispredict recovery)

`JALR` (indirect jump) is taken later because it uses the register file and the ALU to compute its target.

If both of these sources attempt to jump in the same cycle, `X/M` takes priority, since it is executing an older instruction. In both cases, the part of the pipeline in the hazard shadow is invalidated; i.e., `W → F`, or `W → X`. Invalidation is performed by clobbering the pipeline control signals in such a way that these instructions will have no side effects.

The branch prediction scheme is static: take backward branches, and do not take forward branches. The cycle costs are as follows:

Jump Type	Cycles (Execution + Penalty)
Direct jump	2
Predicted, non-taken branch	1
Predicted, taken branch (same as jump)	2
Indirect jump	4
Branch mispredict	4

Upon jumping, we need some mechanism to invalidate parts of the pipeline: this is described in section 2.3.

2.1.5   CIR Locking

There is a landmine in the following tableau:

- `M` contains a load (in data phase), and the bus is stalled

- `X` contains an instruction dependent on the load result; say an `AND`

- `D` contains a branch which is mispredicted taken

For example, say we are polling a status bit in an IO register, looping until some bit is high. The instruction in `X` must stall for at least two cycles due to the bus stall and load-use hazard. Due to a frontend design constraint from 2.1 – "No combinatorial path from AHB-Lite data phase to address phase (e.g. `hready → htrans`)" – we must *not* use `X`'s stall signal to gate the jump request, as this would create such a path. However, *not* gating the jump request is fatal, as new fetches will clobber `CIR` before the branch instruction can proceed into `X`, so the mispredict will never recover. `JAL` has the same problem: it will jump, but not produce a link address.

Hazard5 resolves this with `CIR` locking. `D` signals to `F` that `CIR` and its validity count must not change on the next clock edge, and uses this same signal to inhibit repeated assertion of its own jump request. In the fetch path, this is achieved by steering the controls on the existing shift/overlay logic in the frontend, and requires no additional muxing.

Whilst the `CIR` is locked, the frontend is still free to act on the jump request, and fetch ahead along the new code path (more useful for `JAL`); this data is buffered in the FIFO. Once the roadblock ahead of `D` clears, the branch instruction proceeds down the pipeline, and the lock is released simultaneously.

Note also that a bus stall does not cause the frontend to block jump requests, as this too would create a combinatorial path, since requests are forwarded straight to the bus (due to another constraint from 2.1). The frontend's response to bus stall on jump request is simply to not increment the target before storing to `FC`, so that the target address continues to be asserted on subsequent cycles. Registered feedback blocks *subsequent* jump requests until the first request completes its address phase.

2.1.6 Instruction Barrier (FENCE.I)

If the program stores to instruction addresses about to be executed from, which potentially exist in the prefetch queue, a stale instruction will be executed. `FENCE.I` cannot be decoded as a nop, and requires special handling.

Hazard5 decodes `FENCE.I` as "jump to `PC` + 4". The jump invalidates the prefetch queue, and the following instruction will be re-fetched from memory.

Timing analysis shows that calculating a jump target, between `CIR` and the address bus, is generally on the critical path. To avoid more logic on this path, `FENCE.I` is instead implemented by spoofing a branch-taken mispredict, which has the same effect. This adds two cycles to the execution time, but performance of this instruction is decidedly noncritical.

2.2 Operand Bypass

Hazard5 possesses an operand bypass (forwarding) network. Register writes by one instruction must always be visible to later instructions, even before the first instruction reaches register writeback. This is shown as multiplexers on the ALU inputs in figure **??**.

This reduces (and often eliminates) the penalty of read-after-write data hazards in the pipeline, and allows us to approach one-clock-per-instruction (CPI) execution rates. Without bypass, only one instruction could exist present in $\{D, X, M, W\}$ at a time, giving a CPI of 4.

The following bypasses are available: (notation: pipe register $\rightarrow$ pipestage logic)

- `X/M` $\rightarrow$ `X`

- `M/W` $\rightarrow$ `X`

- `M/W` $\rightarrow$ `M`

- `W/X` $\rightarrow$ `X`

The last is in lieu of a write-to-read bypass in the register file, to avoid difficulties with block-RAM inference on iCE40.

To control the bypassing, some of the register specifiers from `CIR` are passed down the pipeline alongside the data. `rs1, rs2, rd` (operand sources and destination) are passed down as far as `X`. `rs2, rd` make it to `M`, and only `rd` makes it to `W`.

The upshot is:

- Back-to-back ALU operations execute at 1 CPI

- Loads insert 1 stall cycle if immediately required by the ALU. 1 CPI otherwise.

- Stores execute at 1 CPI (bus stall notwithstanding)

- In a load + store pair, the load takes only one cycle, since the `M` stage has self-forwarding

Various interesting strategies can alleviate load-use penalty in in-order pipelines, such as adding a second, "late" ALU in the M stage. In our case we judge this to not be worth the LUTs.

2.3   Pipeline Stalling and Flushing

Our terminology: stalling means a pipeline stage does not advance its state until some blocking condition has cleared. The instruction residing in this stage will not progress to the next stage, and the previous stage will not write *its* instruction into this stage. Flushing is when in-flight instructions in some stages are replaced with NOPs, and their results are discarded.

The frontend is decoupled from other stages' stall logic via the prefetch queue. This is important: hready is an input to that stall logic, and the the frontend's address-phase request must not be a function of hready.

The frontend may not be able to immediately accept a jump request, which may cause other pipe stages to stall if it is low. One cause is the frontend holding an existing address-phase request stable until the cycle *after* hready, which is required for AHB-Lite compliance.

For the backend, the stall logic is more intricate, as signals such as hready are used in-cycle to determine whether an instruction progresses to the next pipeline stage:

- D:
    - CIR does not contain a valid instruction (either no data, or half of a 32-bit instruction)
    - D asserts jump, but frontend rejects the jump request
    - X is stalled

- X:
    - hready low and X address-phase request asserted
    - RAW hazard on M (load-use)
    - M is stalled

- M:
    - hready low and data-phase active
    - M asserts jump, but frontend rejects the jump request

- W: does not stall

If a given stage is stalled, but the following stage is not, it must insert a bubble. Bubbles are created by zeroing out control fields, such as rd, so that the instruction cannot affect system or processor state.

There are two cases where we must flush:

- Branch/jump taken from D; frontend invalidates prefetched data

- Jump/mispredict taken from M; must flush frontend, D, X

And the flushing mechanisms for each stage are as follows:

- D: destination register rd cleared, which makes result invisible to register file and operand bypass. memop, branchcond pipe flags are cleared.

- X: same as D (except for branchcond, which does not pass on to M anyway.

Flushing and bubble insertion are very similar in mechanism.

2.4 Unaligned Memory Accesses

Alignment is the constraint that the address of a memory access be equal to zero, modulo some size. Where no size is specified, we refer to *natural* alignment, i.e. modulo the size of this particular memory operation. RISC-V requires that memory is byte-addressable.

The frontend goes to some length (section 2.1) to maintain high throughput. RV-C instruction streams are *always* unaligned, and *every* instruction must be fetched before it is executed, so Amdahl says it's worth it.

As load/stores are less than 100% of all instructions, and generally much fewer than 100% of these are unaligned, Hazard5 does not have hardware support for unaligned load/stores. These are trapped (TODO) and handled in software if and when they occur.

2.5 Interrupts and Exceptions

This section is very much still TBD. (TODO)

Hazard5 will have non-nested priority vectored interrupts, with no register saving, apart from `ra` (`x2`; link/return address register) and the `PC`. Interrupts and exceptions will be implemented with the early jump hardware, so behave as a `D`-sourced jump.

Interrupts/exceptions are implemented as a jump into the vector table. The table is a block of aligned 32-bit instructions, which will most likely be JAL, meaning ISRs must be within ±1 MiB of the vector table. As well as jumping into the table, the interrupt hardware stashes the `PC` and `ra` into shadow registers; the second requires a register file read, and the pipeline slot of the (victim) instruction in `D` is used to perform this read. `PC` is immediately clobbered by the jump into the table, and `ra` is clobbered with the magic value 0xffffffff, which is otherwise an invalid return address as its LSB is set.

Upon encountering a `jalr` to 0xffffffff (most likely a `ret`), the saved `PC` and `ra` are saved.

Consequently, ISRs may not generate exceptions, such as unaligned accesses. TODO: should have some kind of non-returning hardfault exception to deal with things like this.

3 Bus Fabric and Memory Subsystem

Bus fabric is digital plumbing. A master, such as a processor, requests a read or write on some address; the bus fabric routes the request to the correct slave device, and routes the response back. RISCBoy implements two bus fabric standards:

- AMBA 3 AHB-Lite connects masters to high-performance devices such as SRAM controllers

- AMBA 3 APB connects to simple devices such as a UART

Figure 4 shows the structure of the AHB-Lite crossbar (`ahbl_crossbar.v`). The crossbar is shown in context in figure 1. An independent AHB-Lite datapath connects each of m masters to each of n slaves. One master can address one slave at a time, and one slave can be in data-phase with one master at a time; subject to these constraints, up to $\min(m,n)$ independent transfers can take place in a single machine clock cycle.

Some claim AHB-Lite does not "support" multi-master arbitration. Their problem is a lack of enthusiasm: motorbikes do not "support" wheelies by design, but are excellent at it.

Figure 4: AHB-Lite crossbar, module-level block diagram
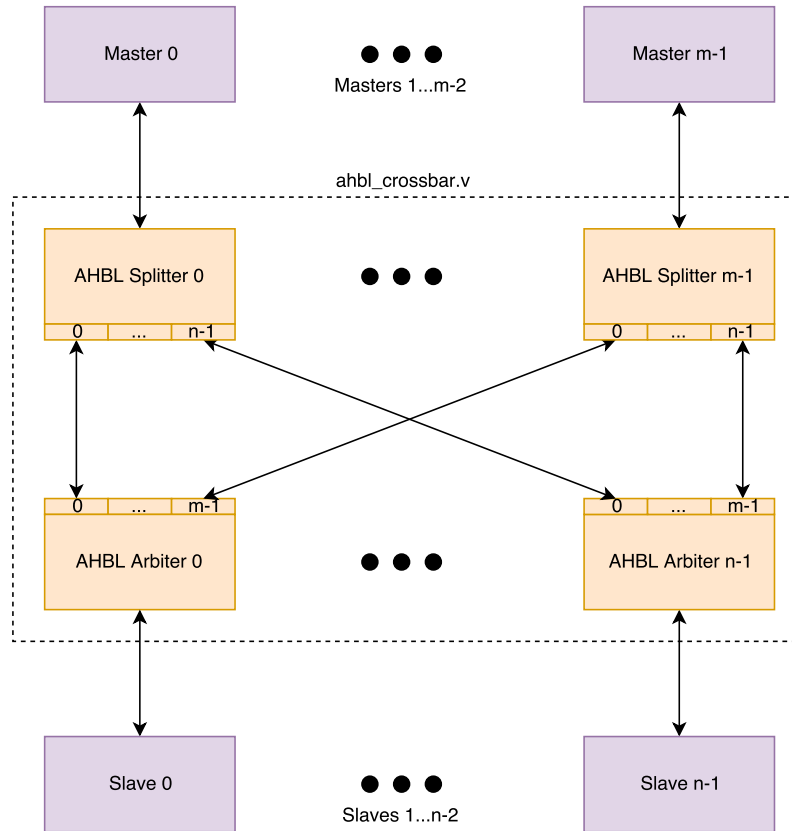
Each master is under the illusion that it is the only master in the system, but that slaves sometimes take longer to respond. During this waiting period, the slave may actually have fielded multiple transactions from higher-priority masters; this interaction is handled by the slave's AHB-Lite arbiter, and is transparent to the masters.

One of the crossbar's slave ports is attached to an AHBL-APB bridge. This bridge appears as a slave to the AHB portion of the bus fabric, and as a master to the APB portion. There are three main benefits to this scheme:

- APB is fundamentally simpler

  - This keeps peripheral gate count down
  - The peripherals on the APB bus do not need the full AHB-Lite bandwidth anyway

- Fewer AHB-Lite slaves

  - There is a nonlinear area scaling associated with adding slaves to the AHB-Lite fabric
  - This would also add extra gate delays to a fairly critical data path

- One APB master

  - AHB-Lite masters get arbitrated down to one inside the AHB-Lite crossbar. APB slaves do not care who is addressing them.
  - Different masters accessing different APB slaves will have to queue to use the bridge, even though they could theoretically proceed simultaneously
  - However, area/complexity vs performance tradeoff is more than worth it for slow peripherals
  - Multi-master APB is easy to implement, but never used in practice, due to the above tradeoff

The splitter and arbiter modules in the AHB-Lite crossbar can also be used on their own. Arbitrary multi-layer busfabric topologies should be possible with these two components.

Currently, the RISCBoy busfabric does not support AHB-Lite bursts (TODO), and the masters do not use them.

3.1 AHB-Lite Primer

For a full understanding of the bus standard used by RISCBoy, read through ARM's AMBA 3 AHB-Lite spec. This document is mirrored in the `reference` folder in the GitHub repository, and gives a clear and comprehensive breakdown of AHB-Lite. However, the following overview should provide sufficient understanding of the standard to read through the Verilog.

Transactions take place in two phases, named the address phase and the data phase. During the address phase, the master asserts signals which control the nature of the transfer, such as the address, whether the transfer is a read or write, protection/permission information, the width of the data, and so on. During the data phase, data is asserted on either the read or write data bus (`hrdata` and `hwdata`), but never both. `hrdata` is driven by the slave, and `hwdata` by the master.

The central conceit of AHB-Lite is that these two phases are *pipelined*. Whilst the master is asserting or accepting data for an earlier transaction (currently in data phase), it concurrently asserts address and control information for a later transaction (currently in address phase). As is generally the case with pipelining, the goal is to enable higher clock frequencies with undiminished work-per-clock.

Figure 5: AHB-Lite transfers, a simple example
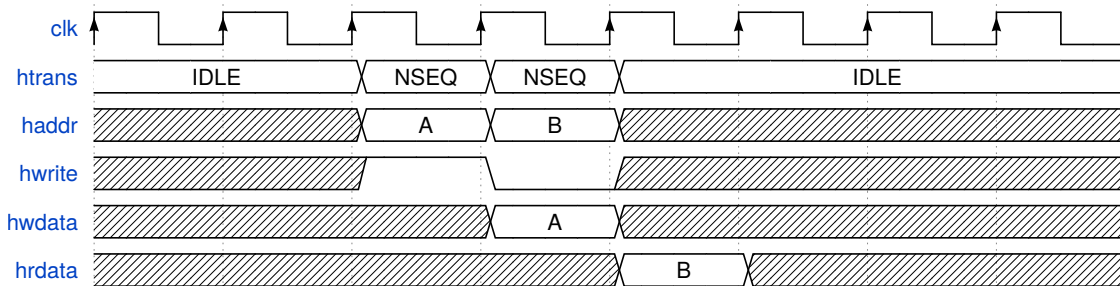
In figure 5, a master carries out two AHB-Lite transactions: a write to address A, followed by a read from address B. Only a subset of AHB-Lite signals are shown on the diagram. `htrans`, `haddr`, and `hwrite` are driven by the master, during the address phase; the other two are data phase signals. `htrans` indicates the type of transfer the master next wishes to perform, which is one of `IDLE`, `NSEQ` (non-sequential), `SEQ` and `BUSY`. The latter two are exclusive to burst transactions, which are not used in RISCBoy. (See the ARM spec if you do want details.)

Sometimes a slave is unable to service a request immediately, as shown in figure 6. `hready` is a data phase signal, which signifies the end of the current data phase.

Figure 6: AHB-Lite transfers, simple example with stalling
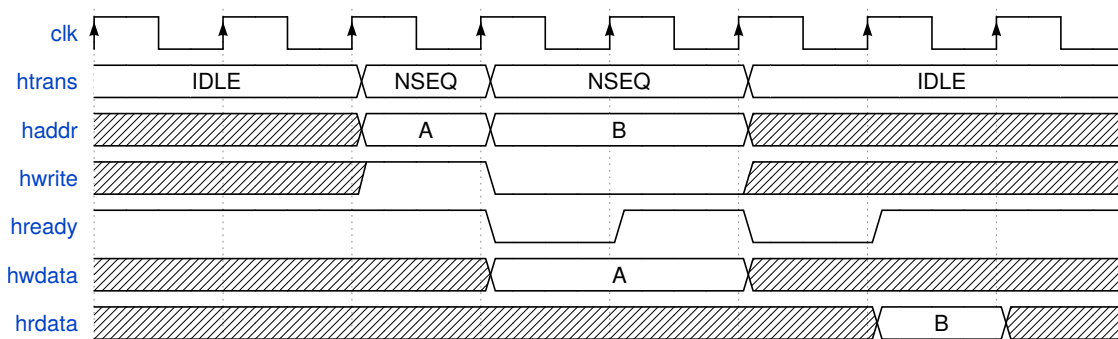
This slave needs two cycles to perform each data phase; perhaps it is an SRAM capable of running only at half the system clock speed. Therefore, `hready` is low for one cycle, and high for the second (last) cycle of each data phase. The master drives `hwdata` for the duration of A's data phase, waiting for the slave to signal completion. `hrdata`, on the other hand, is invalid until the final cycle of a read data phase.

Note that an address phase does not complete until the preceding transfer's data phase completes. In figure 6, address B (and associated address-phase signals) continue to be driven until the A data phase completes. `IDLE` transactions do have a data phase, which always completes immediately. Consequently, `hready` idles high while the bus is in an idle state. This is why A's address phase completes immediately in the figure.

In a practical system, there are multiple slaves. Each drives a signal called `hreadyout`, to indicate that *that slave* is ready. The bus fabric tracks which slave the master is currently accessing in the data phase, and selects that slave's `hreadyout` to be the global `hready`. To see why this is necessary, think about the situation where a master is in data phase with one slave, and address phase with a different slave.

3.2 Multi-Master Operation

In a single-master busfabric, `hready` is a global signal, which causes the entire AHB-Lite state machine (masters, slaves, fabric, the lot) to advance. Where multiple masters are concerned, `hready` is more subtle; in part it is a per-master stall signal. At this point we need to be more specific about the relationship between `hreadyout` and `hready`.

Any AHB-Lite slave port (of which there is one on the master side of the splitter, n on the master side of the arbiter, and one on each slave device) has an output called `hreadyout`, which indicates the slave's readiness. Each of these ports also has an input called `hready`, which indicates that the data phase is ending for the master who is connected to this slave (which does not mean that it is in data phase with *this* slave; it may be addressing this slave while in data phase with another). `hready` is a function of `hreadyout`s and bus state. The connections between masters, splitters, arbiters and slaves are shown in figure 4.

In the single-layer crossbar on RISCBoy, each system AHB-Lite slave is the slave of an arbiter, which is the slave of several splitters, each of which is the slave of a system master. As a general rule, the busfabric must filter system slaves' `hreadyout`s up to each system master, tie `hreadyout`s across to `hready`s at the very top of the busfabric, and then distribute these `hready` signals down to the correct system slaves.

3.2.1 Multiple Masters, One Slave

The arbiters are the most complex busfabric component, so it is instructive to consider interactions between multiple masters and a single slave, which are mediated by one arbiter. There are additional complexities when we combine arbiters and splitters to build a crossbar, which are discussed in the next section.

In figure 7, two masters attempt to access a single slave simultaneously. Assume that master 0 always wins address-phase arbitration:

Figure 7: AHB-Lite transfers, two masters access one slave
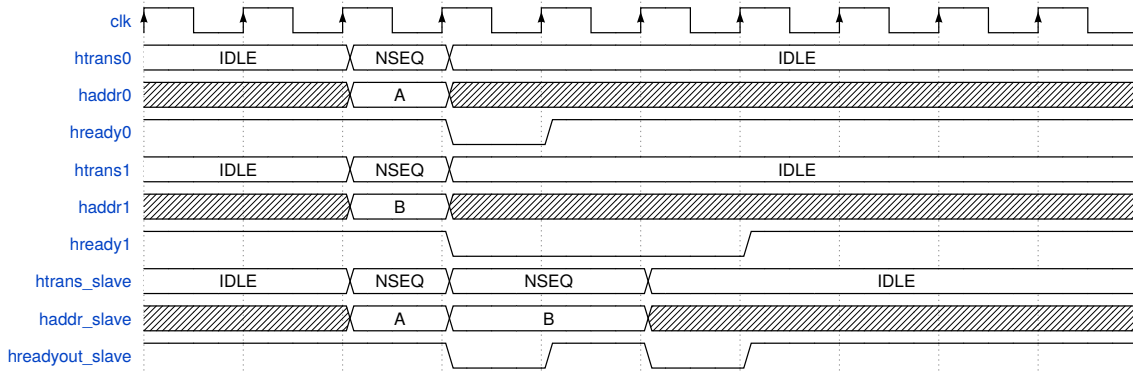
Again, we assume the slave requires 2 cycles to complete each data phase.

If we look at each master's trace, there is no indication at all that there is more than one master in the system: they present an address, and subsequently the transaction completes. Likewise, the slave neither knows nor cares that there are multiple masters: it simply carries out transactions according to the address-phase signals it sees. All of the smoke, mirrors and machinery are inside of the arbiter.

One odd feature of this trace is that, when the slave sees the address B, no master is asserting this address.

1. Initially, both masters assert `IDLE`; `IDLE` data phases complete in one cycle

2. `IDLE` data phases are concurrent with A, B address phases, so these *also* complete immediately

3. From the master 1's point of view, transaction B proceeds immediately to data phase.

4. From both the master 0's and the slave's point of view, transaction A proceeds immediately to data phase

5. Whilst the slave is in data phase for A, it is simultaneously in address phase for B

6. When A data phase completes, master 0 is signaled, and B proceeds to data phase at the slave

7. When B data phase completes, master 1 is signaled

More concisely put, the first clock cycle of a given transaction's data phase may differ between the slave and master, but the *last* cycle of that data phase is always the same clock cycle. The slave address phase will occur some time between the master address phase starting, and the slave data phase starting. These are strong enough guarantees for correct operation.

Based on this discussion, the AHB-Lite arbiters need the facility to buffer one address-phase request, per master. A buffered request will be applied before any new requests from that master, but after any higher-priority requests. There is a nonzero hardware cost to this buffering, but there are clear engineering benefits to keeping this complexity confined to the arbiters, as they are the only component in the busfabric which is explicitly "multi master".

Figure 8: AHB-Lite transfers, two masters access one slave, with low-priority back-to-back
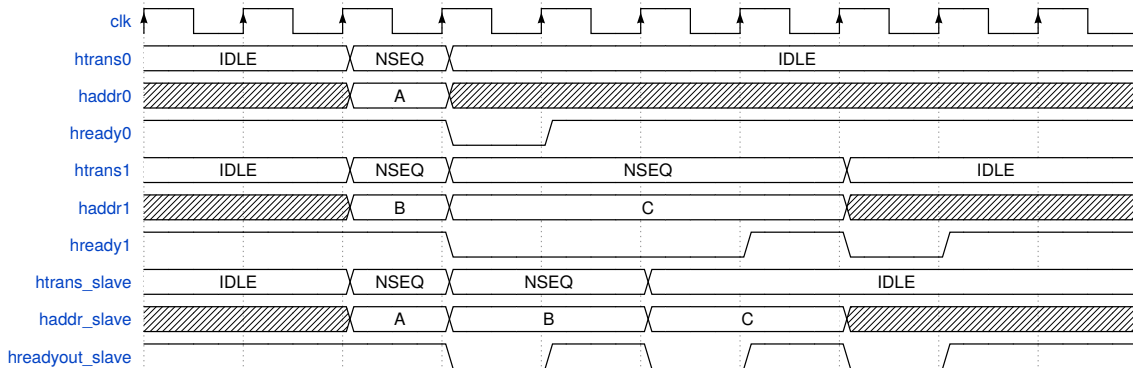
Figure 8 shows the same sequence of events as figure 7, except master 1 now performs two back-to-back transactions. Once B's slave address phase completes, the arbiter's request buffer is cleared, and the C request passes transparently through the arbiter to the slave. Again, the only indication to master 1 of any master 0 activity is increased latency.

There is a different case which requires the arbiter's request buffer, shown in figure 9.

Figure 9: AHB-Lite arbiter: simultaneous request buffer writes
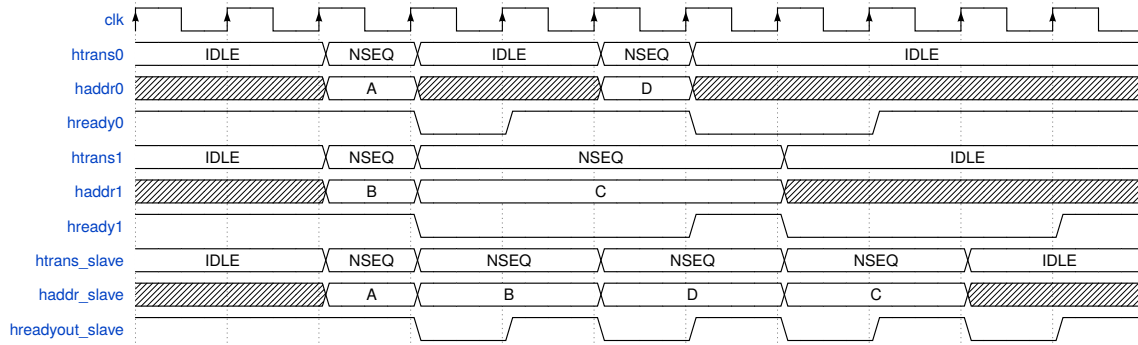
At the instant where D address phase is asserted, `hready0` is high, because master 0 previously asserted an `IDLE` transfer. However, the slave is not ready. In this case, the arbiter needs to buffer master 0's request, even though it is the highest-priority master. The buffered request is cleared once its slave address phase completes, as usual.

On the next cycle, B's data phase completes, and master 1 also considers this to be the end of the C address phase. The arbiter must write the C request into master 1's request buffer. Master 0's buffered request will continue to take priority over master 1's buffered request, until the first buffer is cleared.

There is one final case, for two masters accessing one slave, which is worth being aware of (figure 10).

Figure 10: AHB-Lite arbiter: late arrival of high priority request
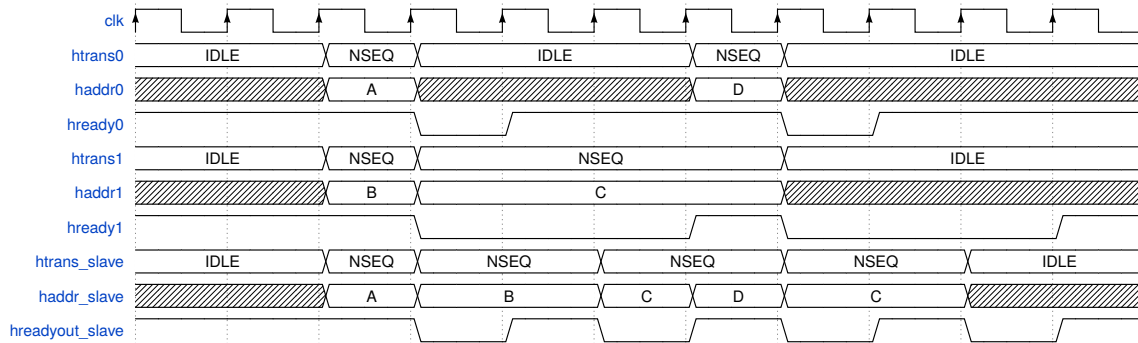
Whilst `hreadyout` is low, the C address briefly appears on the slave bus, before being replaced by the higher-priority D request. **This is a departure from the AHB-Lite standard**, which stipulates the address must be constant during this time. This is deliberate, and easily amended. Slaves are generally insensitive to address-phase request during this time (as there is no performance benefit to latching APR before `hreadyout`, due to the way the bus operates), and this avoids a priority inversion, reducing average latency for higher-priority masters. If you find something that this breaks, write me an angry email! I would be interested to see such a slave.

The D request causes the low-priority C request to be buffered; the B data phase completes on this cycle, hence, from master 0's point of view, the C address phase does too.

3.2.2 Full Crossbar

The previous section discussed some cases where multiple masters access a single slave, and showed how the arbiter safely navigates them. There are yet more issues to consider when multiple masters and multiple slaves are involved,

which must be handled without added latency cycles, and with minimal extra gate delay.

For example, a master may be engaged in address phase with one arbiter and data phase with another arbiter simultaneously, via a splitter, and these two arbiters will not necessarily signal `hreadyout` at the same time. Consequently, a master may have a positive `hready`, filtered from its data phase arbiter, when its address phase arbiter has a negative `hreadyout`, which requires action on the arbiter's part.

There is also the issue that being in data phase with an arbiter does not mean you are genuinely in data phase with the arbitrated slave; in fact, a very simple sequence of events (all masters `IDLE` → all masters `NSEQ`) will put all masters simultaneously in data phase with the same arbiter. The arbiter behaviour described in the previous section should allow us to abstract this away, provided we can deal with the first issue safely.

Splitters will filter their slaves' `hreadyout`s based on which is currently in data phase, and present it on their own slave port. Arbiters will present their slave's `hreadyout` on any master-facing ports which are in data phase with the arbiter, and will present `hreadyout` = 1 on any idle ports.

Splitters will fan their `hready` signal out to all of their slaves; a low `hready` directed at a slave you are not engaged with is harmless.

4 Pixel Processing Unit (PPU)

WARNING: This section rambles incoherently over a piece of hardware that does not yet exist, marking a complete departure from the preceding documentation

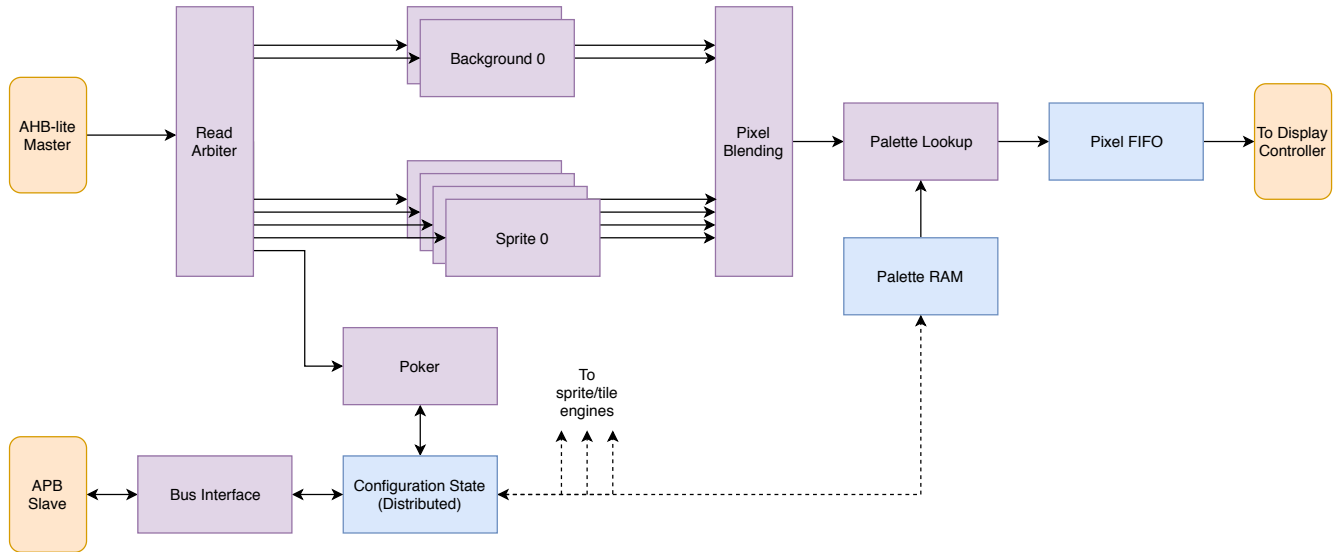Figure 11: Pixel processing unit, block-level diagram

Figure 11 shows the high-level structure of the PPU. A number of independent sprite and background engines stream tile and pixel data from system memory, each producing a pixel stream which is continous in the case of the backgrounds, and generally sparse in the case of sprites. These multiple streams are blended according to transparency and layer order. Finally, any paletted pixels are looked up in the shared palette RAM, to produce full-colour pixels for the display.

The PPU makes no assumptions about display timing: for example, RISCBoy is currently equipped with a 320×240 SPI LCD, which is driven continuously, with no horizontal or vertical blanking periods, as the SPI interface is the framerate bottleneck. The PPU continuously generates pixels at some rate which is hopefully higher than the display scan rate, and its timing is decoupled from the display via the pixel FIFO (as this is the narrowest point of the PPU, so the cheapest to buffer).

A key feature is the Poker: this is a simple raster-synchronised coprocessor that allows the PPU to modify its own configuration with pixel-perfect timing, enabling advanced rendering tricks such as in-scanline sprite multiplexing. It is described in section 4.3.

4.1 Pixel Formats

Internally, the PPU uses a single native pixel format, namely ARGB 1555 (see figure 12), but to save bandwidth, the PPU can stream pixels from memory in a variety of formats, and convert internally. These range down to 1 bit per pixel, for both sprites and tiles. PPU memory accesses are **always little-endian**: for performance reasons the PPU performs the widest possible fetch, yielding multiple pixels each, which are numbered least-significant-first.

Background pixel format is configured per-background; sprite pixel format is configured globally.

For pixels smaller than one byte, the pixel order continues to be defined in a little-endian fashion, i.e. the least-significant pixel will be the first to be displayed. There is an additional constraint that all pixels be naturally aligned in memory. That is, pixel address modulo pixel size is zero.

A paletted pixel is transparent if:

- The colour index is 0, and

- Transparency is enabled for that pixel source

Figure 12: PPU pixel formats

15 14	10 9	5 4	0		
A	R	G	B		ARGB 1555, alpha = 0 when transparent

ARGB 1555, alpha = 0 when transparent

RGB 555, alpha bit is ignored (always opaque)

ARGB 1232, alpha = 0 when transparent

RGB 232, alpha bit is ignored (always opaque)

P8: an index into a table of 256 colours

P4: an index into a table of 16 colours

P2: an index into a table of 4 colours

P1: an index into a table of 2 colours
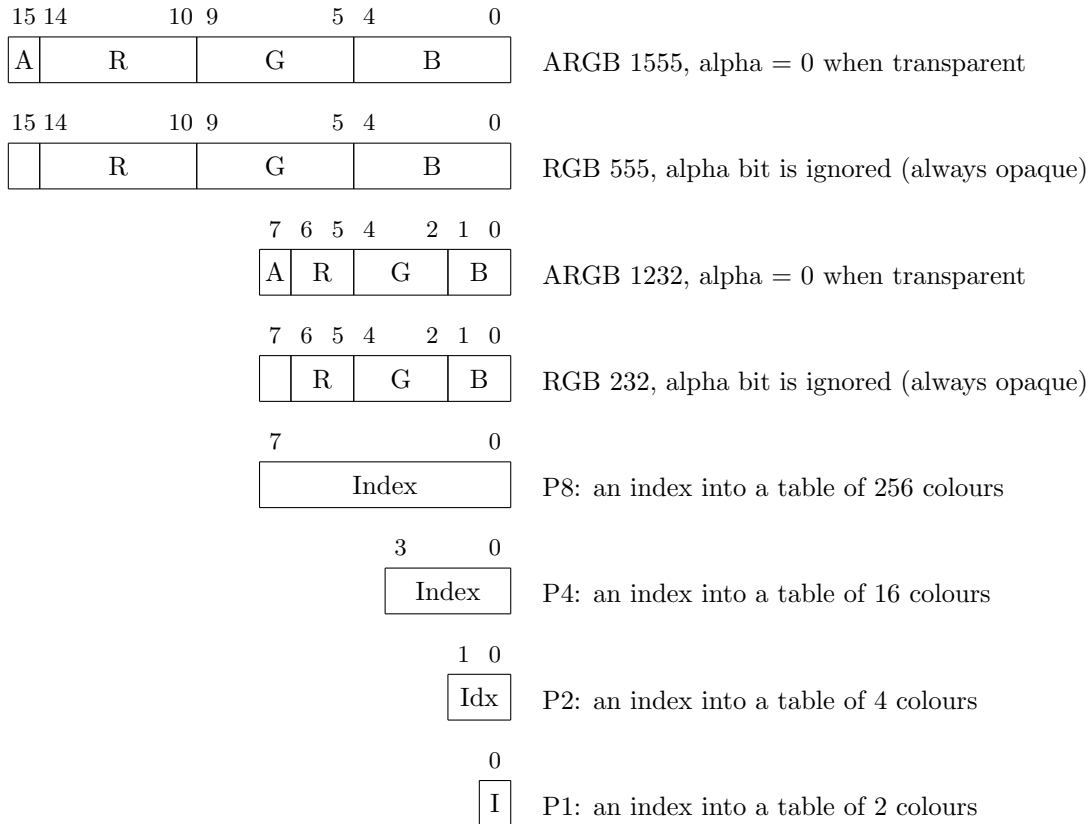
So at e.g. 1 bit per pixel, it is possible to have either 2 colours, or 1 colour plus transparency. It is important that the transparency of a pixel is known as early as possible: this enables the PPU to perform layer composition before palette lookup, so only one palette lookup per *screen* pixel is required.

4.2   Layers

Each background, and each sprite, can be individually assigned a layer number in the range $0...n-1$, where n is the layer count (configurable). For each blended pixel, the blender observes non-transparent pixels from each source (sprite or background), and applies the following rules:

- If there are no input pixels, output the display background colour.

- Else output the pixel with the lowest layer number

- If there are multiple pixels on the same layer:
    - Sprites beat backgrounds
    - Lower-numbered sprites beat higher-numbered
    - Lower-numbered backgrounds beat higher-numbered

4.3   Poker

The Poker is a simple raster-synchronised coprocessor, inspired by the Amiga Copper. It allows the PPU to manipulate its own controls, with pixel-perfect timing. There are no restrictions on when this takes place, but partially-drawn tiles may complete with the old configuration, and the new configuration will take effect on the next sprite/tile (TODO).

17

This enables acrobatic feats such as multiplexing a single sprite back-to-back down a scanline. However, Poker execution does have a performance cost, as Poker instructions are fetched through the same memory interface as pixel data. Overuse may cause missed pixels.

Poker instructions reside in main system memory. Instructions are documented below. The instruction set is designed to be simple to decode and simple to generate dynamically, not to have high code density!

31	24 23	12 11	0
wait	x	y	

Suspend execution, resuming immediately before pixel (x, y)

31	24	15	0
poke		addr	

31			0
data			

Write data to addr (in the PPU's configuration address space)

31	24 23	16 15	8 7	0
sprite		start	end	

31				0
addr				

Batch-update sprites from start to end with memory block starting at addr.

31	24
jump	

31	0
addr	

Set Poker program counter to addr