FPGABoy Documentation

Luke Wren

February 15, 2018

Contents

1 What?

FPGABoy is an open source portable games console, designed from scratch. It is also...

- An open source PCB layout
- Designed with KiCAD open source PCB editor
- An open source CPU, graphics and bus architecture
- Based on the RISC-V open source instruction set
- Synthesised and taped out with iCEStorm open source FPGA toolchain

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

1.1 Logic Design

The heart of the design is a Lattice iCE40-HX8k FPGA, containing 7680 LUT4s and flipflops. The logic was designed from scratch, using synthesisable Verilog. This includes:

- RV32EC-compatible 32-bit CPU design
  - E: embedded profile (reduced register set)
  - C: compressed instruction extension, for higher code density
- 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
- AHB Lite 3.0-compatible multi-master busfabric
- Other peripherals
  - External SRAM controller
  - Display controller
  - DMA master
  - Interrupt controller
  - GPIO controller (buttons!)
  - Serial port etc.

Some attempt is made in this document to describe the operation of these hardware blocks, but if you are looking for nitty-gritty detail, the best documentation is the files ending with `.v`.

That a free synthesis tool can cram all this logic into one of the cheapest FPGAs on the market is tremendously impressive. Hopefully we will one day have a situation similar to software compilers, where free tools such as GCC are industry standards.

1.2 PCB

The board is a 4-layer stackup:

1. Signal + GND Fill
2. GND plane
3. Power planes
4. Signal + GND Fill

It is intended to be suitable for low-cost PCB prototyping services such as iTead. Board dimensions are 50mm × 50mm, which fits into the cheapest size category on iTead. For the most part, it sticks to the following minimum specifications:

- Track width 0.15mm

- Copper-copper clearance 0.15mm
- Soldermask-copper clearance 0.1mm
- Soldermask width 0.1mm
- Via drill 0.3mm
- Annular ring 0.15mm (i.e. via diameter 0.6mm)

The only exception is some 0.5mm vias underneath the BGA. Strictly this is out of specification for iTead, but they claim to have a 90 μm drill registration, so we'll see how it goes.

The iCE40-HX8k FPGA is packaged in a 256-pin 0.8mm BGA, which *can be* reflowed by a hobbyist with a hot air gun or a frying pan (best to choose a HASL finish so that contacts are pretinned). The 132-pin 0.5mm BGA has sufficient IO for our needs, but iTead does not manufacture at the tolerance required for such a fine pitch.

2 CPU Architecture

ReVive is a 32-bit processor based on the RISC V (hereafter RV) instruction set architecture. The name seemed appropriate for what is supposed to be a return to the glory days of games programming, when NULL pointers were just pointers to the start of ROM, and programmers were real programmers who wrote their *own* polygon rasterisers, dammit.

The diagram shown here is a simplified block diagram of the core. Not shown is:

- The full and specific contents of the pipeline registers
- The forwarding/bypass network which shortens data hazards to improve pipeline throughput
- Additional hardware in `ICACHE` which helps to support unaligned code fetches
- Interrupt entry/exit hardware
- The ready/valid handshakes on pipe stages which allow them to NOP later stages, and stall prior ones
- Hardware registers such as PC, and PC update logic
- Pipeline flushing signals for branch mispredict

Overall this is a fairly standard RISC-style processor pipeline. The tall blue boxes represent the registers at the boundaries of two pipe stages. The nomenclature used in the diagram is:

- `F`: fetch
- `D`: decode
- `X`: execute
- `M`: memory access (load/store)
- `W`: register writeback

The processor has a single AHB-lite busmaster; the instruction fetcher and the load/store stage need to share it, and the instruction fetch takes priority.

Branches are speculated, according to the static prediction scheme described in the RV ISA manual:

- Backward branches are predicted to be taken, on the assumption that loops will run at least twice on average.
- Forward branches are predicted not taken; the ISA manual advises that compilers should put more likely code on this path.

RISC-V compressed instructions achieve respectable code density, but it's no Thumb-2. The small instruction cache helps to tame the fetch bandwidth, so that load/stores and other system masters will still see some appreciable fraction of the bus bandwidth.

2.1 Fetch Logic

The F stage always provides D with 32 bits of fresh instruction data, in the canonical RISC-V order (halfword-wise little-endian). There are four sources for this data:

- AHB busmaster with access to system memory map
- ICACHE. A small, fully-associative cache with random eviction
- An F-local buffer containing up to 32 bits of fetched, unused instruction data
- The second halfword of the current instruction register of D, in the case that D found the last instruction to be 16-bit

2.1.1 Purpose of Fetch Buffer

Between them, ICACHE and AHB are able to provide the fetch stage with 32 bits of fresh fetch data on every cycle (assuming that, on cache miss and HREADY low, F simply stalls). The D stage will consume either 32 or 16 bits of instruction data per cycle, and the amount is determined very early in D by looking at the two LSBs of the instruction; F can see the size of the instruction currently in D. Due to the pipelined nature of AHB, the decision of whether to fetch is made one cycle ahead of F itself, which ends up being in the W stage.

This latency necessitates a small amount of buffering: on no-fetch cycles in F (the decision of whether to fetch having already been made one cycle prior), we will still need 32 bits of fetch data on hand to guarantee that we can refill CIR once D has consumed up to 32 bits of instruction.

To summarise our state transitions:

- Fetch:
  - 32-bit instruction in CIR: buffer level stays same
  - 16-bit instruction in CIR: buffer level increases by 16
- No fetch:
  - 32-bit instruction in CIR: buffer level decreases by 32
  - 16-bit instruction in CIR: buffer level decreases by 16

This suggests a simple rule: fetch if the buffer is not currently full. If we follow this rule, we guarantee that the buffer will have 0, 16 or 32 bits of data at all times.

2.1.2 Sequential Code

In sequential code, the ICACHE will be queried every cycle during W, as to whether it has the next aligned word. We are able to guarantee that all AHB and ICACHE accesses are word aligned due to the operation of the fetch buffer in F.

The query response is combinatorial, and is used to decide whether to assert an AHB access during the address phase. Successful queries will result in cached read data appearing on the next clock edge, for use by F.

F is concurrent with the AHB data phase, except for a small muxing layer at the end which assembles the instruction word from AHB data, buffered data and cached data.

Following an unsuccessful ICACHE query, F will write the AHB data into the cache, evicting one word of its present contents. *The same address must not appear more than once in the cache*, as the query logic is based on a non-priority sum of product encoder.

2.1.3 Jumps

Upon a jump or taken branch, F's halfword buffer is invalidated, and the CIR is ignored.

Aligned jumps are otherwise no different from sequential code: the ICACHE is queried with the word access, and AHB supplies the data if ICACHE does not have it.

Unaligned jumps (i.e., halfword-aligned but not word-aligned) are more painful, as we may need to perform two accesses to fetch the required data.

The logic is as such:

1. W queries the cache with the address containing the first halfword and, in parallel, unconditionally fetches the second halfword via AHB
2. - After a cache hit, both required words will be available at the end of F. We have not lost any cycles.
   - After a cache miss, W will stall stages D,X,M, and issue a second access over AHB, to fetch the first halfword

3. For fetched halfwords 0, 1, 2, 3:
    - 0 is discarded
    - 1, 2 are sent to `D`
    - 3 is stored in `F`'s halfword buffer

After this, we return to the sequential code steady-state.

2.1.4 ICACHE Writeback

`ICACHE` operates with the following timing:

1. - Fetch address is asserted to `ICACHE.raddr` on the rising edge, in `W`.
   - Data valid bit is available before the end of the cycle (cache is small, and valid check is optimised for speed)
   - Valid bit is used to determine whether to assert a transfer request on AHB
2. - `ICACHE` read data is presented to `F` on next rising edge.
   - `ICACHE` write enable, address and data are asserted by `F` during this cycle
3. - New data is visible in `ICACHE` as of beginning of this cycle.

The encoder logic used in the cache lookup does not give correct output if a tag appears multiple times in the cache. It is therefore vital that the same piece of data is not written more than once. In light of this, there is an issue with the timing above: if we were to query the same address in `ICACHE` on two consecutive cycles, the second cycle will not yet see the results of the writeback, so will also trigger an AHB fetch. On the third cycle, with the results of the first fetch already in the cache, `F` would also be writing the same data again, triggered by the second instruction fetch.

One resolution is for `F` to store the last address written to `ICACHE`, and gate writes to this address. This is okay, but adds another 30 flops to `F`, and the routing will already be quite congested. A better answer is to guarantee that `W` will not fetch from the same address on consecutive cycles, since this would itself be an inefficiency.

2.2 Jumps and Branches

Due to the pipelined nature of AHB, we are unable to jump or to take branches in fewer than 2 cycles:

- Cycle 0: AHB address phase to fetch jump/branch instruction
- Cycle 1: AHB data phase for fetch of jump/branch. Next instruction is in address phase concurrently.
- Cycle 2: Jump/branch instruction is now available to `D`, and can be (quickly) used to control the new address phase.
- The immediately following instruction is already in data phase

This gives the following cycle costs, assuming full speculation:

- Jump: 2 cycles
- Predicted, non-taken branch: 1 cycle
- Predicted, taken branch: 2 cycles (same as jump)
- Branch mispredict: 4 cycles

The branch prediction scheme is static: take backward branches, and do not take forward branches.

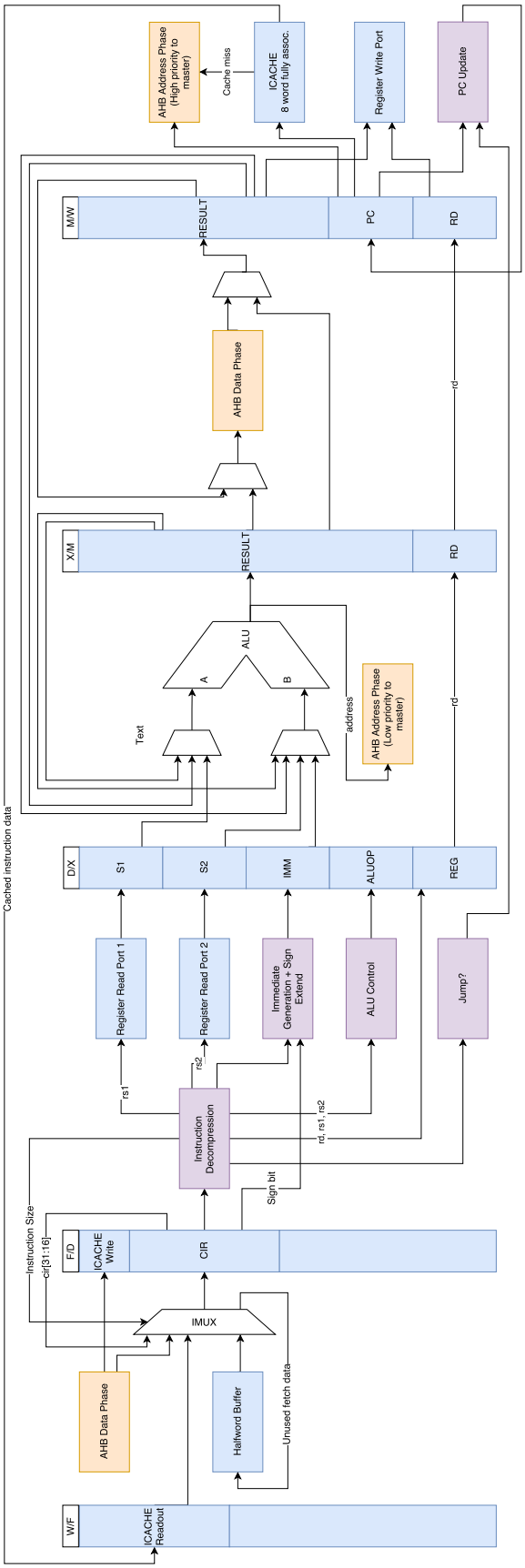
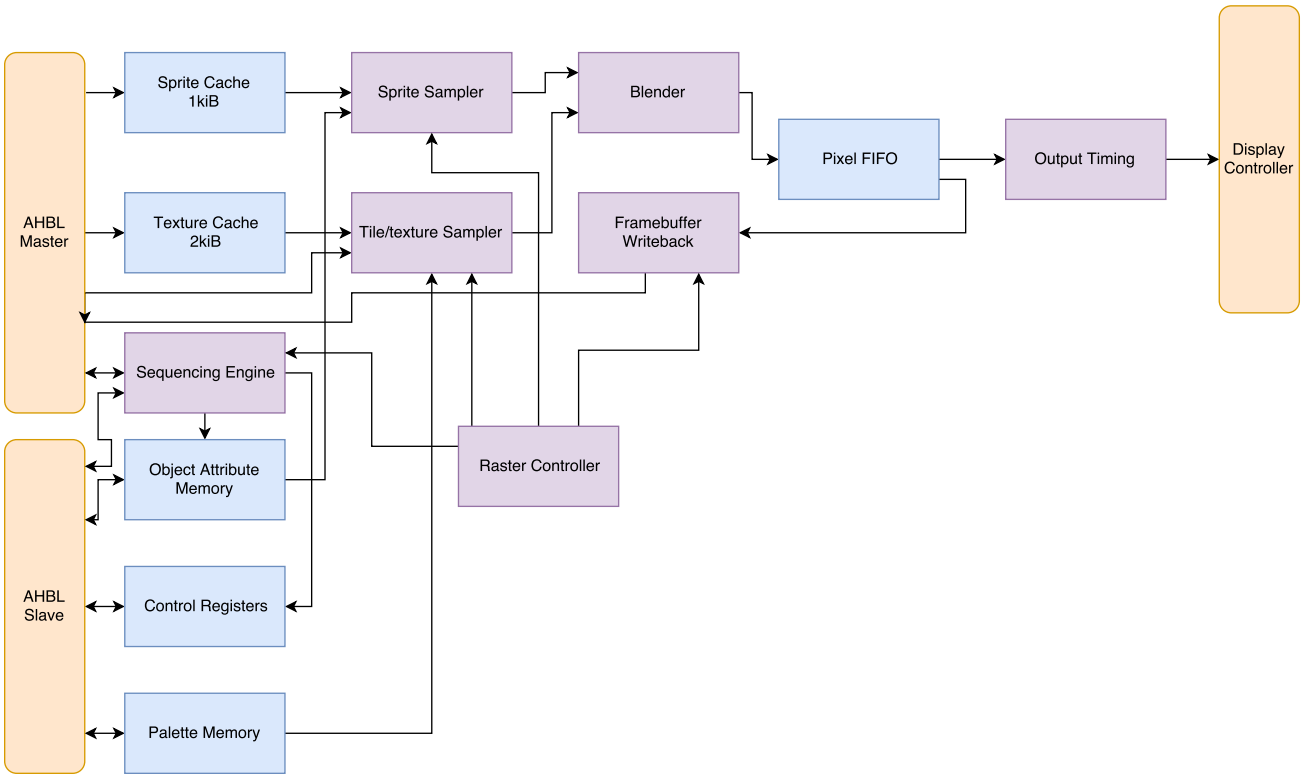


Figure 1: ReVive architectural block diagram

3 Graphics Pipeline



4 Bus Fabric and Memory Subsystem

AHB-lite single layer full crossbar, 32-bit datapath, no burst support. Low-bandwidth peripherals attached with an AHB-APB bridge + splitter. Written from scratch, so no guarantees that it works.

External 0.5MiB asynchronous SRAM with 16-bit data bus, 10 ns access times