FPGABoy Documentation

Luke Wren

February 18, 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 (unless the fetch buffer is full), 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 fetch 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 of the word containing the first instruction halfword
   - If we hit ICACHE, we issue an AHB read to fetch the *second* word in parallel with reading out from the cache
   - If we miss, we issue an AHB read to fetch the first word from system memory.

2. On the next cycle, we have either 32 or 64 bits of fetch data available to F, depending on cache hit/miss
    - The first halfword is useless
    - If we have only 32 bits, then we put the second halfword into CIR, and assert a flag to inform D that only this halfword is valid
    - If we have 64 bits, we forward the middle two halfwords to CIR, and stash the final halfword in the fetch buffer. Normal execution resumes.
3. If CIR was half-valid:
    - D reports instruction was 16-bit: we got away with it.
    - D reports 32-bit instruction: D stalls, F sends a full CIR based on new fetch data from this cycle.

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.

2.1.5 Program Counter

ReVive does *not* use the program counter (PC) for code fetching, during sequential execution. It is used exclusively for the first fetch cycle of jumps/branches, the link value in JAL(R), and branch mispredict recovery.

Instruction fetch takes place from a consecutive series of word-aligned addresses, as described in section 2.1.3. F feeds data back to the fetch control in W to indicate whether a new fetch is needed. W therefore never requires the value of PC during normal operation. However, during non-sequential execution, it does require a PC value. Bits [31:2] are used to set the new fetch address, and bit 1 is used to determine whether the first instruction is aligned or unaligned with the fetch. The difference is outlined in section 2.1.3: essentially, on an unaligned jump, the first 16 bits of the fetch data are ignored, and the hardware scrabbles to put together a valid instruction.

The true PC lives in F. It watches the size of instructions in D, and uses this to increment by 2 or 4 each cycle. D looks at this PC value when using its jump logic: it can either look at the value currently in the PC register, or the value due to be written to it at the end of the current clock. This gives the address of the instruction currently in D, and the address of the immediately following instruction, respectively. The first is used in the calculation of jump targets; the second is the link value used in JAL(R) and mispredict recovery.

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.

2.3 Operand Bypass

ReVive follows the standard practice of providing a multiplexed operand bypass (or forwarding, depending on your predilections) network. The purpose is that register writes by a given instruction always be visible to later instructions, *before* the first instruction reaches the register writeback stage. This is shown as multiplexers on the ALU inputs in figure 1.

This removes, or at least shortens, read-after-write data hazards in the pipeline, and allows us to approach one clock-per-instruction execution rates. Without bypassing, only one instruction could be present in `D,X,M,W` at a time, giving a CPI of 4.

The following bypasses are available: (notation: pipe register → pipestage logic)

- `X/M` → `X`
- `M/W` → `X`
- `M/W` → `M`
- `M/W` → `D` (internal bypass in register file for write port → read port)

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 `rd` makes it to `W`.

These serve the following purposes:

- `rd` is needed in `W` anyway, for performing the actual writeback
- `X` looks at the two operand source registers it depends on, and then glances across at the `rd`s awaiting writes in `M` (i.e. its own output) and `W` (i.e. a load output, or an `X` output one cycle prior).
- `M` Looks at the `src` register for a store (encoded in the `rd2` specifier field), and will look at the pending register write in `W`, which will be either a load result or a prior `X` result.

This means that:

- Back-to-back ALU operations execute at 1 CPI
- Loads execute at 2 CPI if they are immediately required by the ALU. 1 CPI otherwise.
- Stores execute at 1 CPI
- In a load-store pair, the load takes only one cycle, since the `M` stage has cyclic forwarding

2.4 Pipeline Stalling and Flushing

Stalling: a stage causes prior stages to not advance their state, until this stage is ready.

Flushing: cause the in-flight instructions in some pipeline stages to be replaced with NOPs, and their results discarded.

2.5 Pipeline Operation – Worked Examples

This section is included as much for my benefit as anything else. It describes the operation of all 5 pipeline stages for a few specimen instructions. I found it helpful to print out a few copies of figure 1 to trace out instructions with pen and paper.

2.5.1 ADD

Assembly: `add rd, rs1, rs2`
Pseudocode: `rd` ← `rd1` + `rd2`

1. `F`
   - Instruction can be either 16 or 32 bit
2. `D`
   - Current values of source registers are read out from register file
   - An internal bypass in the register file routes in the result of the instruction currently in `W`, if said instruction's `rd` matches our `rs1,rd2`, and the instruction generates a regfile write.
3. `X`
   - Check the `rd` of `X/M` against our `rs1, rs2`. If there is a match:
     - If `M` instruction is an ALU op: bypass it in
     - If `M` instruction is a load: stall until it completes
   - Otherwise, check `rd` in `M/W`. If there is a match, bypass this result in.
4. `M`
   - `add` is a no-op in `M`
   - Our result is present in the pipe register, as well as our `rd`, and the fact that we are an ALU operation. Earlier pipe stages will forward our result as necessary
5. `W`
   - Our result is pending write to the register file, and will be visible as of the next cycle
   - `D` is able to bypass our result in due to internal regfile bypass. `X,M` can also bypass our result using the normal bypass network
6. Retirement
   - The result has been written to the register file. The instruction has no further side effects.

2.5.2 JAL

Pseudocode: `rd` ← `PC + 2/4; PC` ← `PC + imm`
(`JAL` has both 32-bit and 16-bit variants, so the value of the link address will vary.)

1. `F`
   - Whilst the jump is being fetched in `F`, the following instruction is entering the AHB address phase in `W`.
   - This following fetch will be wasted, but we have no way of knowing this until we start to decode the jump instruction.
2. `D`
   - This is a J-format instruction. It contains 20 bits of immediate data, which are sign-extended and effectively left-shifted by 1 by the immediate mux structure.
   - `pc_next` is passed on to `dx_linkval`. ALU + muxes are configured to compute the sum of this value and `x0`.
   - Immediate value is passed on as normal.
   - `F` is stalled
3. `X`
   - ALU passes link address to `X` result.
   - Jump target adder computes `PC + imm`.
   - Fetch update logic in `W` is flagged that `X` has a valid jump target. This is forwarded on to the fetch logic.
   - `D` is stalled.

4. M

    - New instruction is in F. Two penalty cycles were incurred by jump.
    - The instruction is finished, but the link value continues on down the pipeline toward W. It is also available to the bypass network.

5. W

    - Link address is written to register file.
    - Target instruction is now in decode, and may use the link value via regfile bypass.

2.5.3 JALR

Pseudocode: `rd ← PC + 2/4; PC ← rs1 + imm`

1. F

    - Whilst the jump is being fetched in F, the following instruction is entering the AHB address phase in W.
    - This following fetch will be wasted, but we have no way of knowing this until we start to decode the jump instruction.

2. D

    - This is an I-format instruction. It contains 12 bits of immediate data, which are sign-extended but not shifted.
    - `pc_next` is passed on to `dx_linkval`. ALU + muxes are configured to compute the sum of this value and `x0`.
    - Immediate value is passed on as normal.
    - F is stalled

3. X

    - ALU passes link address to X result.
    - Jump target adder computes `rs1 + imm`.
    - Fetch update logic in W is flagged that X has a valid jump target. This is forwarded on to the fetch logic.
    - D is stalled.

4. M

    - New instruction is in F. Two penalty cycles were incurred by jump.
    - The instruction is finished, but the link value continues on down the pipeline toward W. It is also available to the bypass network.

5. W

    - Link address is written to register file.
    - Target instruction is now in decode, and may use the link value via regfile bypass.

2.5.4 LW and friends

Pseudocode: `rd ← mem[rs1 + imm]`
Two cases: aligned and unaligned. Aligned:

1. F

    - Nothing special to see here.

2. D

    - This is an I-format instruction. It contains 12 bits of immediate data, which are sign-extended but not shifted.

3. X

    - ALU computes `rs1 + imm`. This is the load address. AHB address phase starts immediately if master is not contested (timing should be okay...)

4. `M`
    - AHB data phase takes place. Loaded word appears in `M/W` result register at end of cycle.
5. `W`
    - Loaded data is written to register file.

Unaligned: Only `X` and `M` are different.

1. `X`
    - ALU computes `rs1 + imm`.
    - First cycle: AHB address phase to `(rs1 + imm) & ~0x3`. `F, D` stalled.
    - Second cycle: AHB address phase to prev address + 4. `F, D` resume.
2. `M`
    - Second cycle: store AHB data phase result in a word-sized local buffer.
    - Third cycle: use muxes to build full load result from buffer, second data phase.

`SW` is even worse. Due to AHB alignment requirements, a word-store may have to be implemented as a byte store, halfword store and byte store.

`LH` will require either one halfword fetch, one word fetch, or two byte fetches depending on alignment.

`LB` can always be completed with a single fetch.

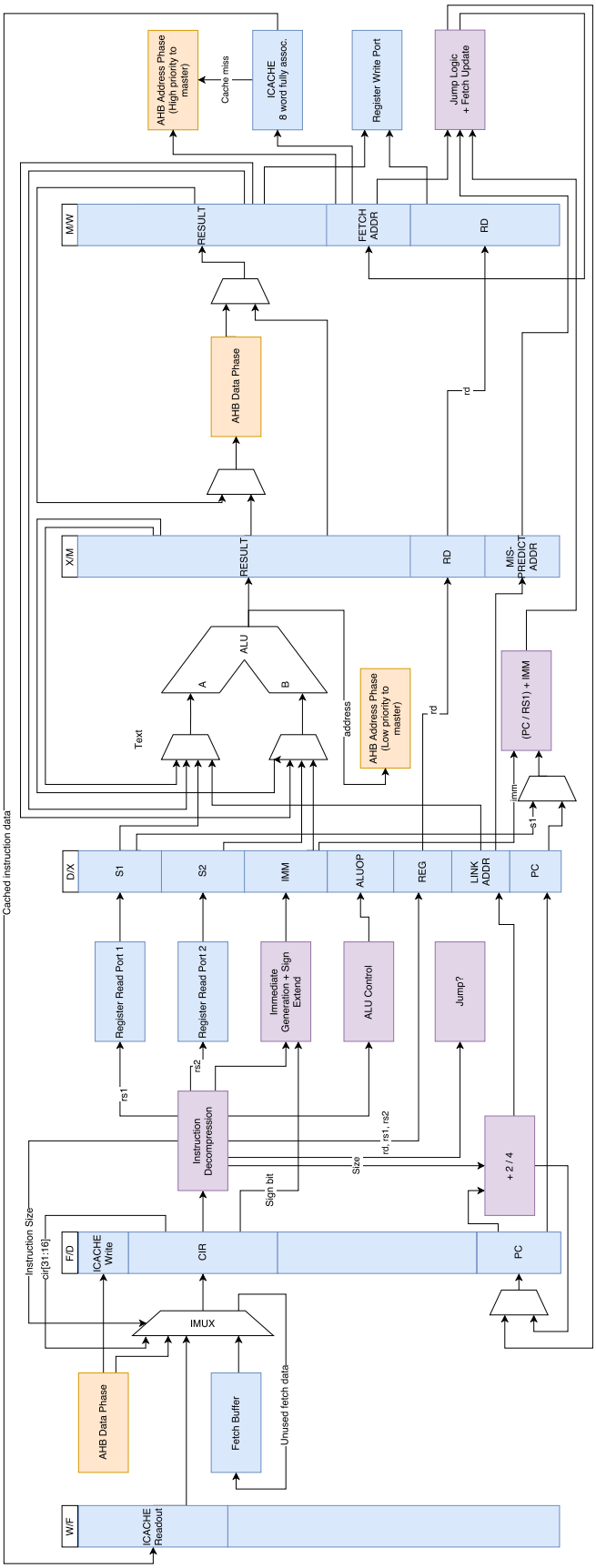
2.6 Interrupts

Interrupts and traps are implemented using the mispredict recovery hardware. They are precise and immediate; however, no state saving is carried out by the processor. The handler is responsible for saving any registers it uses on the stack, and restoring the state of the stack afterwards. The RISC-V ISA makes it possible for our handler to completely restore the processor state before exiting, but does not provide any mechanism to perform a return without first smashing the link register on handler entry.

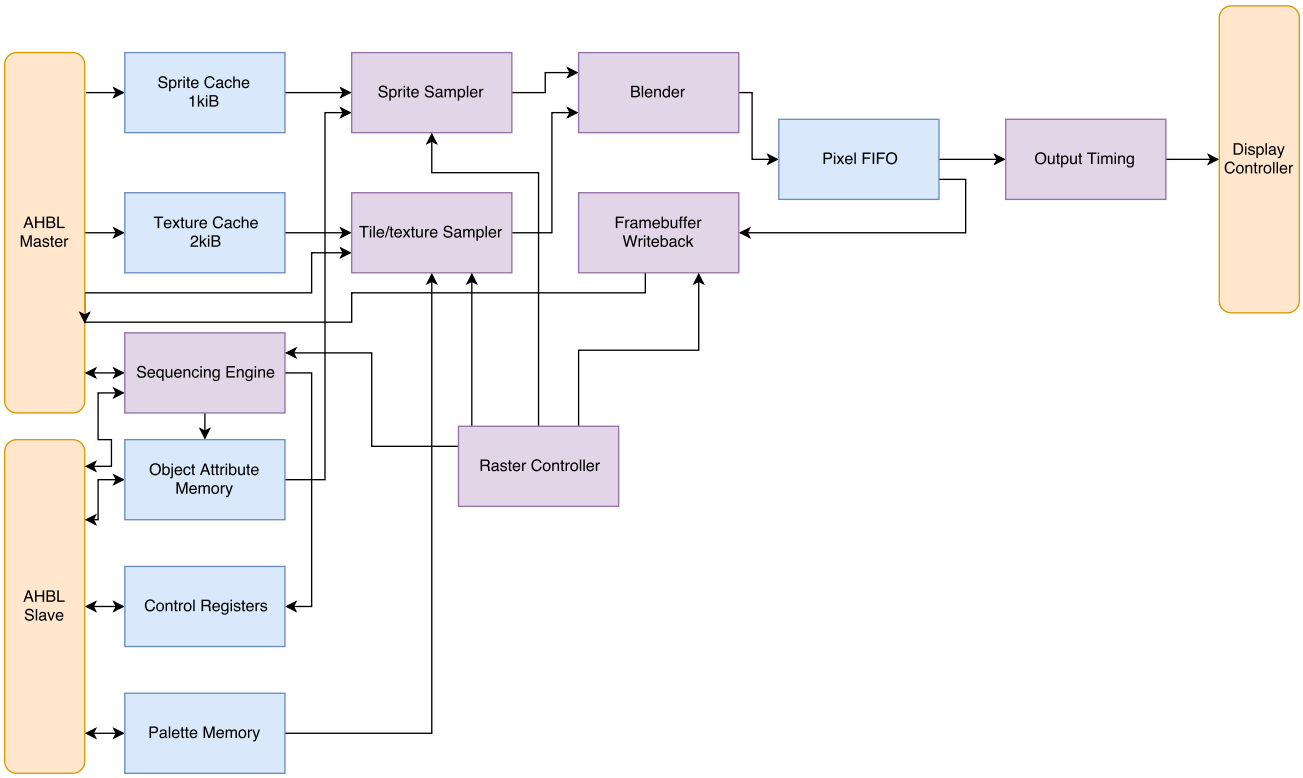
The interrupt return mechanism is implemented with a small amount of ISA abuse: the hardware stashes the link address in an invisible architectural register during handler entry. The handler calls the magic instruction `JAL x0, 0` to jump back to this stashed `PC` value and therefore exit the interrupt. Hopefully not too much is lost by shadowing this instruction (jump to self without link).

This assumes that the programmer always maintains a valid stack pointer in `x2`, with enough space to save the registers. In practice this is not such an onerous task, and other processors such as Cortex-M0 make liberal use of stack operations on interrupt entry/exit.

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