# Orca: Optimized RISC-V Cryptographic Accelerator

Kosi Nwabueze
Massachusetts Institute of Technology
Cambridge, Massachusetts
kosinw@mit.edu

Haoran Wen
Massachusetts Institute of Technology
Cambridge, Massachusetts
hranwen@mit.edu

## ABSTRACT

Hardware acceleration for cryptography is widely supported in modern computer architectures. Intel first added support for Advanced Encryption Standard (AES) instructions to x86 in 2008. ARM added support for AES instructions for ARMv8 in 2011. However, instead of adding instructions to extend the ISA, we propose a different model for improving the performance of AES cryptographic routines: an external co-processor unit. We present Orca: a RISC-V microcomputer implemented on a Xilinx Spartan-7 FPGA. Orca features a pipelined RV32IM core, a co-processor providing a hardware implementation of AES-128, a text-mode video card reminiscent of 1980s IBM VGA cards, a UART programmer, and a PS/2 interface for a keyboard. By offloading cycle-intensive cryptographic routines such as AES encryption and decryption to a co-processor, we expect to observe a significant throughput increase compared to software-implemented alternatives reducing thousands of software cycles to dozens of co-processor cycles.

## 1 INTRODUCTION

At a high level, the final microcomputer design consists of the following components:

**Central Processing Unit.** At the heart of the Orca microcomputer sits a RISC-V core acting as the central processing unit. Our custom core implements the RV32IM ISA with a classic 5-stage RISC pipeline complete with pipeline hazard management, an instruction and data cache, and a cycle-by-cycle on-board debugger.

**UART Programmer.** Orca has a UART receiver that receives comamnds to control the processor and write to program RAM over a serial port. These commands include: writing a 32-bit word to a memory address, resetting the program counter and internal state of the RISC-V core, halting the execution of the core, and starting the execution of the core.

**Video Card.** Orca can display a matrix of $160 \times 45$ character cells on a 720p HDMI display. Software can manipulate each character cell individually by writing to a memory mapped buffer.

**Keyboard.** Orca supports a keyboard peripheral device which uses the PS/2 protocol. Whenever the microcomputer receives a scancode from the keyboard, the scancode is written in a memory-mapped hardware register and the software is responsible for polling a control register to check if the buffer has data in it.

**AES co-processor.** Orca features a cryptographic co-processor which implements both encryption and decryption with the AES-128 block cipher. Software can interface with this co-processor through memory-mapped registers.

**Other MMIO.** Orca also features an additional two MMIO registers: `ENTROPY` and `COUNTER`. The `ENTROPY` register provides a random number generator for the software runtime to sample from. The `COUNTER` device is a counter that increments every 50 cycles on a 50Mhz clock. Therefore, every tick on the `COUNTER` register is 1000ns. The software runtime can read this value to implement sleeping similar to `nanosleep(2)`.

Figure 1 shows how the different submodules of the microcomputer communicate with each other in a block diagram. Orca has a 32-bit address bus, which is used for ROM, RAM, and I/O. Table 1 shows the physical memory map of the entire microcomputer and its peripheral devices. In total, the microcomputer supports up to 64KiB of mixed instruction/data memory, 14KiB of video memory, and 2KiB of AES buffer memory.

**Table 1: Address-space of Orca microcomputer.**

| Start | End | Size | Description |
|---|---|---|---|
| `0x00000` | `0x10000` | `0x10000` | General-purpose program memory |
| `0x10000` | `0x10004` | `0x00004` | `COUNTER` MMIO register |
| `0x10004` | `0x10008` | `0x00004` | `ENTROPY` MMIO register |
| `0x20000` | `0x03840` | `0x23840` | Video memory |
| `0x30000` | `0x30080` | `0x00080` | Keyboard scancode buffer |
| `0x30080` | `0x30081` | `0x00001` | Keyboard control register |
| `0x40000` | `0x40400` | `0x00404` | AES input buffer |
| `0x40404` | `0x40804` | `0x00404` | AES output buffer |
| `0x4F000` | `0x4F001` | `0x00001` | AES control register |

In the following sections, we will discuss the design of each submodule and then conclude iwth an evaluation of our design.

## 2 CENTRAL PROCESSING UNIT (KOSI)

Currently, the implementation of the processor is complete. In the preliminary report, we implemented a single-cycle design of the processor core as a proof-of-concept and a baseline for the pipelined design. All of the code for the single-cycle processor is available in the `old/` folder in the attached repository.

The Orca processor is an implementation of the RV32IM standard, the 32-bit RISC-V base ISA with the standard extension for multiply and divide instructions. The processor follows a modified Harvard architecture with separate instruction and data memory; however, instruction memory can be read from. Currently, our processor is clocked at 50Mhz (half the clock rate of the board) using a Vivado template we generated. In doing so, we have a comfortable amount of worst negative slack to play with (4ns) and worst hold slack to play with (0.1ns). We implement the classic 5-stage RISC pipeline shown in $6.1910_{[6.004]}$. While designing the processor, we realized that support for interrupts would require a significant overhaul of the internal control logic for the pipeline so we instead opted with a polling approach to communicate with MMIO devices.

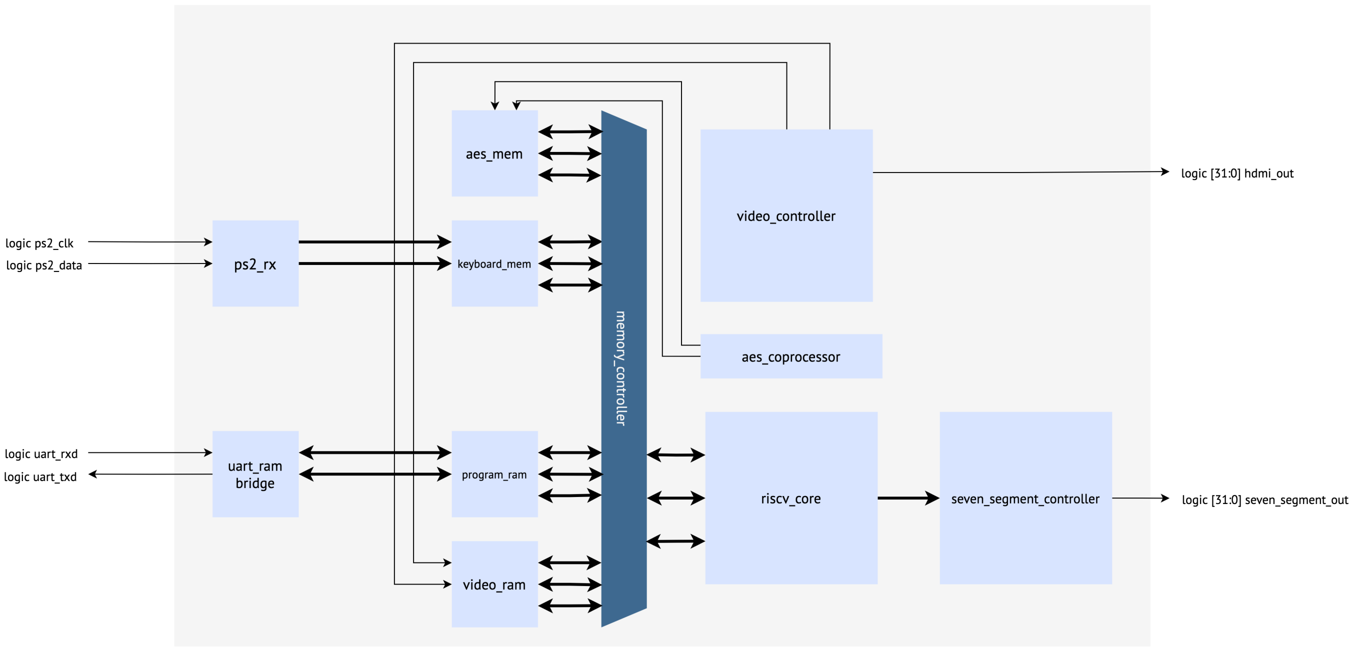


**Figure 1: A block diagram of the Orca architecture.**

Figure 2 shows a high-level overview of all the pipeline stages and control logic of the processor. The implementation of the central processing unit consists of the following SystemVerilog modules: riscv_alu.sv, riscv_constants.sv, riscv_core.sv, riscv_decode.sv, riscv_icache.sv, riscv_lsu.sv, and riscv_regfile.sv.

We designed the control logic in the processor to be versatile, so each stage can be stalled and annulled independently. This versatile approach made extensions that we wrote later on such as instruction cache misses or 32-cycle division much easier to implement. In the following subsections we discuss the implementation of each pipeline stage, the implementation of the control logic, and finally the methodology we used for debugging the processor.

## 2.1 Instruction Fetch

The instruction fetch stage matches the typical instruction fetch stage described in the classic 5-stage RISC pipeline. The relevant modules for this stage would be riscv_core.sv and riscv_icache.sv. The processor has a program counter that is incremented after every instruction fetch except after `jal`, `jalr`, or any conditional branch instruction. Our novelty comes with the implementation of our branch predictor and instruction cache.

For the sake of simplicity, our branch predictor does absolutely no prediction and assumes that jumps and branches are never taken. Then for all jump instructions and whenever a conditional branch is taken, the control logic decides to annul the past two instructions in the pipeline (namely the instructions in the decode and fetch stage).

Our instruction cache implements a two-way set assosciative cache with block size 1 (each cache line holds a single 32-bit word) and 32 sets (for up to 32 instructions in the cache). From our experience in testbenching, cache misses become very inconsequential in loops with a few number of instructions. Most of our programs have control structures that would be very cache efficient since we rely heavily on polling which are small loops.

## 2.2 Instruction Decode

The instruction decode stage takes the instruction word from the instruction cache and decodes the immediate, source registers, the program counter select, operand select for the ALU, the write-back select, the ALU function, the branch function, write enable for the register file, and data memory information. The relevant modules for this stage would be riscv_core.sv, riscv_regfile.sv, and riscv_decode.sv.

Through a combinational read port, the source operands are passed into the 32-bit register file and the values are read out and passed into the execute stage.

If one of the source operands from the decoded instruction matches the destination register of the instruction in the execute, memory, or writeback stage then the control logic will decide to bypass the offending operand with the value from a later stage. Priority for bypassing is given to the most recent stage after decode.

## 2.3 Execute

The execute stage takes the operands from the instruction decode stage and applies either an ALU function or branch function to the two operands. The relevant modules for this stage would be riscv_alu.sv and riscv_core.sv.

The ALU functions correspond to the ALU operations in the RISC-V instruction set such as `add`, `xor`, `shl`, or `sra`. The branch functions correspond to branch operations in the RISC-V instruction set such as `beq` or `bgeu`. For `jal` and `jalr` instructions, the ALU also calculates the next program counter.

For the standard multiply/divide extension, we implement single-cycle efficient multiplication without any negative slack by utilizing the DSP48E1 multiply-and-add blocks on the Spartan 7 FPGAs. We implement a fixed, 32-cycle division algorithm by reusing the fixed-cycle, non-pipelined divider module discussed in lecture 9.

## 2.4 Memory

The memory stage dispatches load/store requests to the memory controller without any caching mechanism. The relevant modules for this stage would be riscv_lsu.sv and riscv_core.sv. If the current instruction does not execute a load or store instruction then it passes to the writeback stage of the pipeline without any stalls. Similarly, if the current instruction is a store then no stalling is necessary since the data will be available by the next cycle. However, if the current instruction issues a load, then the entire pipeline with the exception of writeback will stall for 2 cycles until the requested data is available.

Much of the combinational logic in the load/store unit comes from handling Xilinx byte write enable block RAM modules. Since the processor can read and write memory to the byte (8), halfword (16), and word (32) levels much of combinational logic has to shift and mask data coming in and out of the memory controller.

The design of the logic in the load/store unit also has a major shortcoming: unaligned memory accesses. According to the RISC-V specification, only unaligned instruction reads are illegal and should cause faults. However, unaligned data accesses are allowed, but the specification mentions a huge cycle penalty can be applied. Unaligned memory accesses are impossible in the Orca RISC-V processor and trying to execute one will not work. Fortunately, the RISC-V GNU C compiler can be forced to only output aligned memory accesses (bar some exceptions with variadic arguments) with the `-mstrict-align` compiler flag.

## 2.5 Writeback

The final stage in the pipeline is the writeback stage. The relevant modules for this stage would be riscv_regfile.sv and riscv_core.sv. Depending on whether the write-enable-register-file signal was set in the decode stage, the register file will optionally record the value either from the ALU or the data memory.

## 2.6 Control Logic

Our processor implements full bypassing, meaning that there is a bypass from EX to ID, MEM to ID, and WB to ID. Due to this, we have completely eliminated the need to stall for read-after-write hazards. In the following subsections, we discuss how Orca resolves each of the following hazards: load-to-use hazards, instruction cache misses, data cache misses, branch misprediction, and division stalling.

*2.6.1 Load-to-use hazards.* Load-to-use hazards occur whenever the instruction in the EX or MEM stage will execute a load instruction **and** the destination register of the offending instructions is used as a source register in the instruction decode stage. These hazards are resolved by stalling the IF and ID stages of the pipeline.

*2.6.2 Instruction cache misses.* An instruction cache miss occurs whenever the program counter asks the instruction cache for an address not in its cache. These hazards are resolved by stalling the IF stage of the pipeline.

*2.6.3 Data cache misses.* A data cache miss occurs whenever a load instruction occurs because there is no proper data cache. These hazards are resolved by stalling the IF, ID, EX, and MEM stages of the pipeline.

*2.6.4 Branch misprediction.* Control hazards occur whenever a branch or jump is executed. These are resolved by annulling the IF and ID stages of the pipeline.

*2.6.5 Division stalling.* Division stalling hazards occur whenever the ALU is executing `div`, `divu`, `mod`, or `modu`. These are resolved by stalling the IF, ID, and EX stages of the pipeline.

## 2.7 Debugging

The initial design for the pipelined processor was completed in three or four days originally; however, completely debugging the processor took another 10 days after that. To cope with the long Vivado build times we came up with the following mechanisms to make debugging easier:

*2.7.1 Halting Mode.* The processor can be placed in halting mode which allows it to advance one cycle in its pipeline by pressing `btn[3]` on the Urbana board. Designing this made it much easier to debug on actual hardware. In halting mode, you can also view the program counter of each stage, the instruction in the ID stage, and the value of any of the registers on the seven-segment display by controlling the switches on the Urbana board.

*2.7.2 UART Programmer.* Since we could not get Manta to work properly with our project, we instead wrote our own module to interpret commands based on the UART receiver code in Manta. An external computer can issue commands to the soft processor through a Python script using the `pyserial` package. The UART programmer can halt or start the processor, overwrite the instruction and data memory of the computer, and reset the registers, caches, and program counter of the processor.

## 3 VIDEO CARD (KOSI)

Currently, the implementation of the video card is complete. The video card is a text-mode display, which treats the content of the screen in terms of characters instead of individual pixels. The video card displays a $160 \times 45$ matrix of character cells on the screen. Each character cell is $8 \times 16$ pixels, so in total the video buffer covers $1280 \times 720$ pixels, utilizing 100% of the space in 720p HD video. Unlike the rest of the system, the video card runs on a 74Mhz clock domain; however, it can still communicate properly with the CPU through a block RAM which has two ports in different clock domains.

Figure 3 demonstrates the high-level design of the video card. We reuse the `video_sig_gen.sv` module we wrote in earlier labs to generate a video signal at around a 74MHz clock rate. Using the horizontal count and vertical count signals, the video hardware performs a look-up in software-controlled video memory to determine what character to render at that position on the screen. Then, using the `font_brom.sv` module, the hardware figures out the pattern to draw for a particular character by looking up its dot matrix pattern

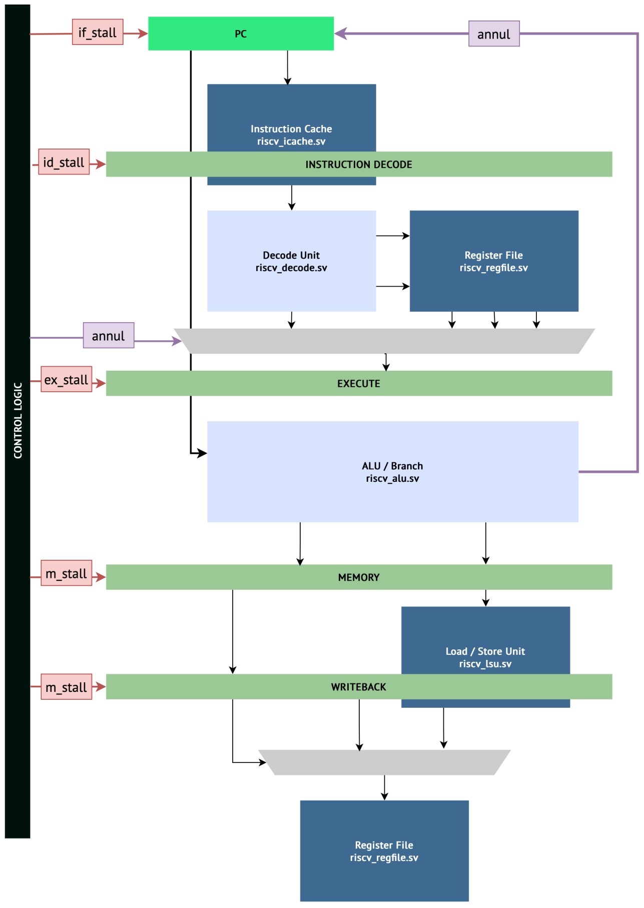


**Figure 2: A block diagram of the Orca RISC-V processor.**

in memory and storing its font. Finally, the `attribute_brom.sv` module determines the background color (one of 8 colors based on the original Microsoft VGA Palette), the foreground color (one of 16 colors based on the same palette), and whether or not that character cell is blinking.

For any character cell location $(x, y)$, user software can configure what character is displayed by writing the ASCII value (technically any character from IBM code page 437 works) of the character to memory address $0x3000 + 2 \times (160 \times y + x)$. To manipulate the background and foreground color, write to address $0x3000 + 2 \times (160 \times y + x) + 1$.

The character font and color palette are converted to a synthesizable `.mem` file by a custom Python script. During synthesis, Vivado flashes the read-only block memories with the appropriate `.mem` files.

## 4 KEYBOARD (KOSI + HAO)

Currently, the implementation of the keyboard is complete. Keyboard peripheral support is implemented by the `ps2_rx.sv` module. The keyboard interfaces with a PS/2 connector which is connected to the Urbana Board through a PS/2-to-PMOD breakout board. Figure 4 shows the simple finite state machine which implements the receiver logic for the keyboard.

Scancodes are the fundamental message send by the peripheral device. Each scancode is a serial frame of 10 bits:

- 1 start bit (always 0)
- 8 data bits
- 1 parity bit (odd parity)
- 1 stop bit (always 1)

The PS/2 protocol outputs one scancode whenever a key is pressed and then outputs the same scancode preceded by the byte `0xF0` when the key is released.

The receiver module writes scancodes it receives from the keyboard into module `ps2_bram.sv`, which implements a hardware ring buffer. After the scancode has been written to memory, the receiver module notifies the CPU through a hardware interrupt. Software can then read from the ring buffer and update appropriate status registers to notify the PS/2 hardware which characters have been read.

The keyboard peripheral is integrated into the micro-computer through a KEYBOARD_CTRL_REGISTER at address $0x3\_0080$ and scancode buffer that starts at address $0x3\_0000$.

The KEYBOARD_CTRL_REGISTER is 8 bits with the following information.

- KEYBOARD_CTRL_REGISTER[0] - Scancodes available flag (HIGH if scancodes available in scancodes buffer, LOW otherwise)
- KEYBOARD_CTRL_REGISTER[7:1] - Counter for the number of scancodes available in the scancodes buffer

When there is a key press, the hardwire writes the scancode to the scancode buffer and sets the KEYBOARD_CTRL_REGISTER[0] bit HIGH. The software pulls the control register and if there is data available, loops through the buffer up the scancode counter (KEYBOARD_CTRL_REGISTER[7:1]) to get the scancodes and writes 0x0 to the control register to reset it.

## 5 AES CO-PROCESSOR (HAO)

### 5.1 Hardware Breakdown

Orca's AES co-processor implements a hardware accelerator for the 128-bit ECB (Electronic Code Book) mode of the AES algorithm. This means that the AES co-processor uses a 128-bit secret key and 128-bit block cipher to perform the algorithm. Below is the breakdown of the hardware modules that comprise the AES co-processor. The following are the core hardware modules that perform the AES algorithm for encryption and decryption.

- aes_core.sv test
- aes_encryption.sv
- aes_decryption.sv
- aes_sbox.sv
- aes_inv_sbox.sv
- aes_key_memory.sv
- aes_key_schedule.sv
- aes_key_memory.sv

These are the high-level controller that connects the AES to the main processor.

- aes_co-processor.sv

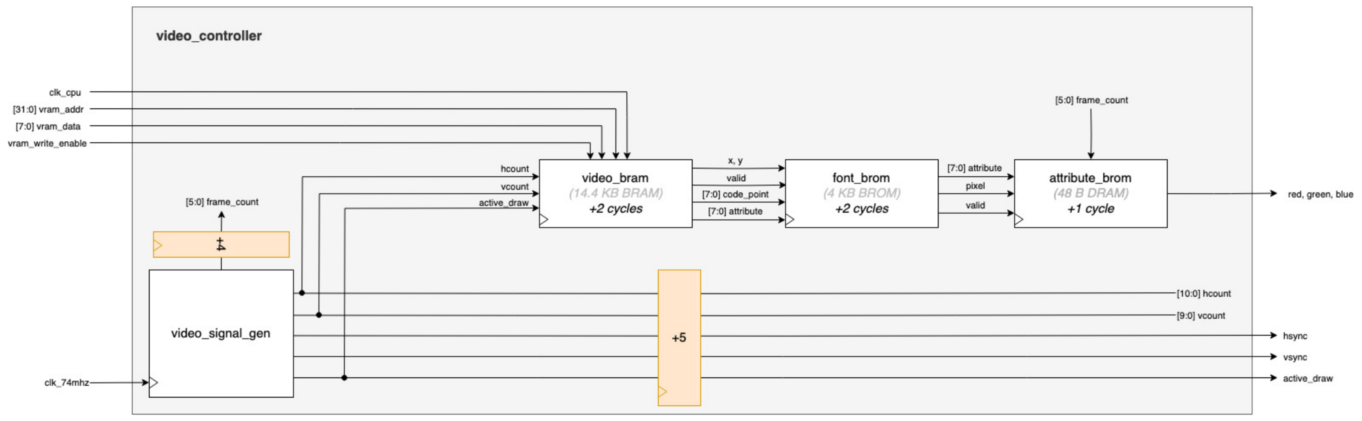


**Figure 3: A block diagram of the video pipeline.**

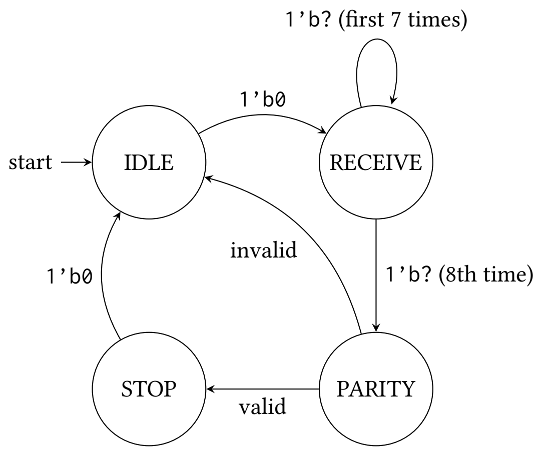


**Figure 4: Receiver state machine for the PS/2 protocol.**

- aes_mem.sv

The integration works as follows. There are two lines of buffer: one for input text for AES to process, and another one for output text after AES is done processing. The input buffer goes from address $0x4\_0000$ to $0x4\_0404$ and output buffer goes from address $0x4\_0404$ to $0x4\_0804$. There is an MMIO address at $0x4\_F000$ that is a control register with the following information.

- AES_CTRL_REGISTER[0] - encryption flag
- AES_CTRL_REGISTER[1] - decryption flag
- AES_CTRL_REGISTER[2] - AES output data available flag (HIGH is processed data is in the output buffer, LOW otherwise)
- AES_CTRL_REGISTER[3] - AES status (HIGH for processing, LOW otherwise)
- AES_CTRL_REGISTER[7:4] - AES stage counter

AES stages are as follows.

(0) RD_DWORD_1
(1) RD_DWORD_2
(2) RD_DWORD_3
(3) RD_DWOR_4
(4) START_AES
(5) WAIT_FOR_AES_RESULT
(6) WB_DWORD_1
(7) WB_DWORD_2
(8) WB_DWORD_3
(9) WB_DWORD_4

The AES co-processor will read 4 words and perform the AES algorithm on it before putting the result into the output buffer. This will keep going until it hits a terminating data, which is hardwired to be $0xDEADBEEF$. Once all the input buffer data is processed, hardware writes to the AES_CTRL_REGISTER[2] a value of 1 to signal that the AES processing is complete and data is ready in the output buffer.

The software can then read the control register at address $0x4\_F000$ and poll to check until AES_CTRL_REGISTER[2] is HIGH before reading the AES output buffer data.

The software can also initialize the start AES by writing to address $0x4\_F000$ a value of $b001$ for encryption or $b010$ for decryption. The user will have to put data in the AES input buffer beforehand.

aes_core.sv is the controller that integrates the encryption, decryption, and key schedule.

## 5.2 AES Algorithm Overview - Encryption

*5.2.1 Key Schedule Algorithm.* Upon initiating the AES core, the AES core first expands the inputted secret key for 10 rounds, using the key schedule algorithm (KSA). The KSA generates 10 unique keys on top of the original input key for each round of the total 10 rounds of encryption and decryption. The 11 unique keys are stored via the aes_key_memory.sv module. After the key has been expanded, the AES then starts the encryption or decryption process. Both encryption and decryption have 10 rounds, each round has 4 stages of repeated logic. At the end of the 10 rounds of performing each of the 4 stages, the respective cipher text/plain text will be ready.

*5.2.2 Sub Bytes.* For encryption, each round consists of the stages sub bytes, shift rows, mix columns, and add round keys. In subbytes, each byte of the block is substituted via a look-up table of the Rijndael S-box. The s-box is implemented via combinational logic that takes in the input block and connects the appropriate s-box value based on the byte bit address.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \rightarrow \begin{bmatrix} s_{a11} & s_{a12} & s_{a13} & s_{a14} \\ s_{a22} & s_{a23} & s_{a24} & s_{a21} \\ s_{a33} & s_{a34} & s_{a31} & s_{a32} \\ s_{a44} & s_{a41} & s_{a42} & s_{a43} \end{bmatrix}$$

In the above transformation, $s_{a11}$ is the substitute value of $a_{11}$ after the s-box lookup table.

*5.2.3 Shift Rows.* In the shift rows stage, each row of the block is shifted by some amount to the left. No change is done to the first row. A left shift of 1 byte is done to the second row. A left shift to f 2 bytes is done to the third row. Lastly, a left shift of 3 bytes to the left is done in the 4th row.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \rightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{22} & a_{23} & a_{24} & a_{21} \\ a_{33} & a_{34} & a_{31} & a_{32} \\ a_{44} & a_{41} & a_{42} & a_{43} \end{bmatrix}$$

*5.2.4 Mix Columns.* In the mix columns stage, each one of the four columns is modulo multiplied in Rijndael's Galois Field by a given matrix. Below is the matrix used.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

We perform the modulo multiplication with each column.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ a_{41} \end{bmatrix} = \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \end{bmatrix}$$

$$\ldots$$

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

*5.2.5 Add Round Key.* Lastly, in the add round key stage, each entry is xor-ed with the respective key from the at a given round from the AES key memory. For example, at round 5, the block of data would be xor-ed with the round 5 key from key memory.

$$\begin{bmatrix} rk_{n11} & rk_{n12} & rk_{n13} & rk_{n14} \\ rk_{n22} & rk_{n23} & rk_{n24} & rk_{n21} \\ rk_{n33} & rk_{n34} & rk_{n31} & rk_{n32} \\ rk_{n44} & rk_{n41} & rk_{n42} & rk_{n43} \end{bmatrix} \oplus \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

## 5.3 Decryption

Decryption is a similar process as encryption except it is now the reverse of the stages done in encryption. So the stages are now inverse add round key, inverse mix columns, inverse shift rows, and inverse sub bytes.

*5.3.1 Inverse Add Round Key.* Decryption adds the round key in reverse order. What this means is that in the first round of decryption, it is going to add the 10th key from the key memory. In the second round of decryption, it's going to add the 9th key, and so on.

$$\begin{bmatrix} rk_{n11} & rk_{n12} & rk_{n13} & rk_{n14} \\ rk_{n22} & rk_{n23} & rk_{n24} & rk_{n21} \\ rk_{n33} & rk_{n34} & rk_{n31} & rk_{n32} \\ rk_{n44} & rk_{n41} & rk_{n42} & rk_{n43} \end{bmatrix} \oplus \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

$rk_{nentry}$ represents the n-th round key entry from the key memory.

*5.3.2 Inverse Mix Columns.* Next is the inverse mix columns. It is still the same concept as mix columns from encryption, but Rijndael's Galois Field is now the following.

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix}$$

*5.3.3 Inverse Shift Rows.* Next is the inverse shift rows operation. For inverse shift rows, each row is shifted right rather than left accordingly. The first row remains unchanged. The second row is shifted right 3 times. The third row is shifted right 2 times, and the last row is shifted right 1 time.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \rightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{22} & a_{23} & a_{24} & a_{21} \\ a_{33} & a_{34} & a_{31} & a_{32} \\ a_{44} & a_{41} & a_{42} & a_{43} \end{bmatrix}$$

*5.3.4 Inverse Sub Bytes.* Lastly, each byte of the data with the Inverse Rijndael S-box.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \rightarrow \begin{bmatrix} is_{a11} & is_{a12} & is_{a13} & is_{a14} \\ is_{a22} & is_{a23} & is_{a24} & is_{a21} \\ is_{a33} & is_{a34} & is_{a31} & is_{a32} \\ is_{a44} & is_{a41} & is_{a42} & is_{a43} \end{bmatrix}$$

where $is_{a11}$ is the inverse s box substituted value for $a_{11}$.

## 6 SOFTWARE (KOSI)

Currently, we have written a wide variety of software demos for the Orca microcomputer including a small runtime library. We use the 32-bit version of the official RISC-V GNU toolchain to build our software using `gcc`, `objdump`, `objcopy`, and `ld`. To evaluate the correctness of the Orca RISC-V processor we wrote the following software demos:

(1) `helloworld.S` - A simple, hello world program.
(2) `aestest.c` - Encryption and decryption using the AES coprocessor.
(3) `conway.c` - Implementation of Conway's Game of Life.
(4) `divide.c` - Testing integer divison.
(5) `kbdtest.c` - Testing keyboard peripheral device.
(6) `mmio.c` - Testing the ENTROPY and COUNTER mmio devices.

(7) `monitor.c` - Simple program which monitors a memory region.
(8) `multiply.c` - Testing integer mulitplication.
(9) `quicksort.c` - Implementation and test of quicksort algorithm.
(10) `videotest.c` - Testing out video device.
(11) `xorshift.c` - Implementing random number generator xorshift32 algorithm.

Figure 5 and Figure 6 show examples of these programs executing on our video hardware.

## 7 EVALUATION

We evaluate our design based on testbench results and Vivado reports. We will consider our design successful by the following metrics: (1) maximizing the processor cycles-per-instruction time (CPI), (2) showing a considerable cycle improvement through a hardware AES co-processor, and (3) by looking at utilization and timing reports from Vivado.

### 7.1 Processor Performance

After the implementation of a simple two-set associative instruction cache, we saw that pipeline utilization of the processor shot way up and the cycle latency of executing loops went way down. Also, the throughput of the processor went way up since the pipeline can have a maximum of five instructions in the pipeline due to the reduced number of stalls. Before, the pipeline could only have two instructions in flight since every instruction fetch would cause a 3-cycle stall.

### 7.2 AES co-processor

For most of the testing of AES processing (encryption or decryption), it took around 200-300 cycles before data from the input buffer was processed and placed in the output buffer.

### 7.3 Utilization Reports

According to our `vivado.log` file, we achieved a worst negative slack of 4.106 nanoseconds and worst holding slack of 0.044 nanoseconds. According to the `post_synth_util.rpt`, we used 53.33% of all our block RAM tiles (which means we could have allocated more towards program memory). In addition, we used 9.17% of DSPs to implement multiplication across the AES co-processor and the RISC-V processor. Finally we used three clocking domains, 100MHz, 50MHz, and 74MHz which used 40% of all our MMCME modules.

## 8 CONCLUSION

This was a very challenging project because of many different factors but it was a great learning experience overall. The core of the project was mainly split between the pipe-lined RISC-V processor, AES co-processor, and peripherals.

The pipe-lined RISC-V processor is written out by Kosi, from the pipeline design to hazard handling to branch prediction. He wrote the logic for the processor that powers the microcomputer. Kosi was also responsible for writing the following peripherals: a video controller for displaying text to the screen, a keyboard for handling PS/2 protocol, and manta based UART protocol for code upload. Hao wrote the logic for the AES co-processor as well as the integration of the AES co-processor and keyboard to the main processor. The open-sourced code base for Orca is available at https://github.com/kosinw/orca.

The final product of the Orca processor was a well-rounded processor that supports the optimized running of many software programs, as well as included an onboard hardware accelerator for AES encryption and decryption for security. While there were certainly lots of challenges in coming up with the design and development of the system, it was a great learning opportunity and experience in the end.

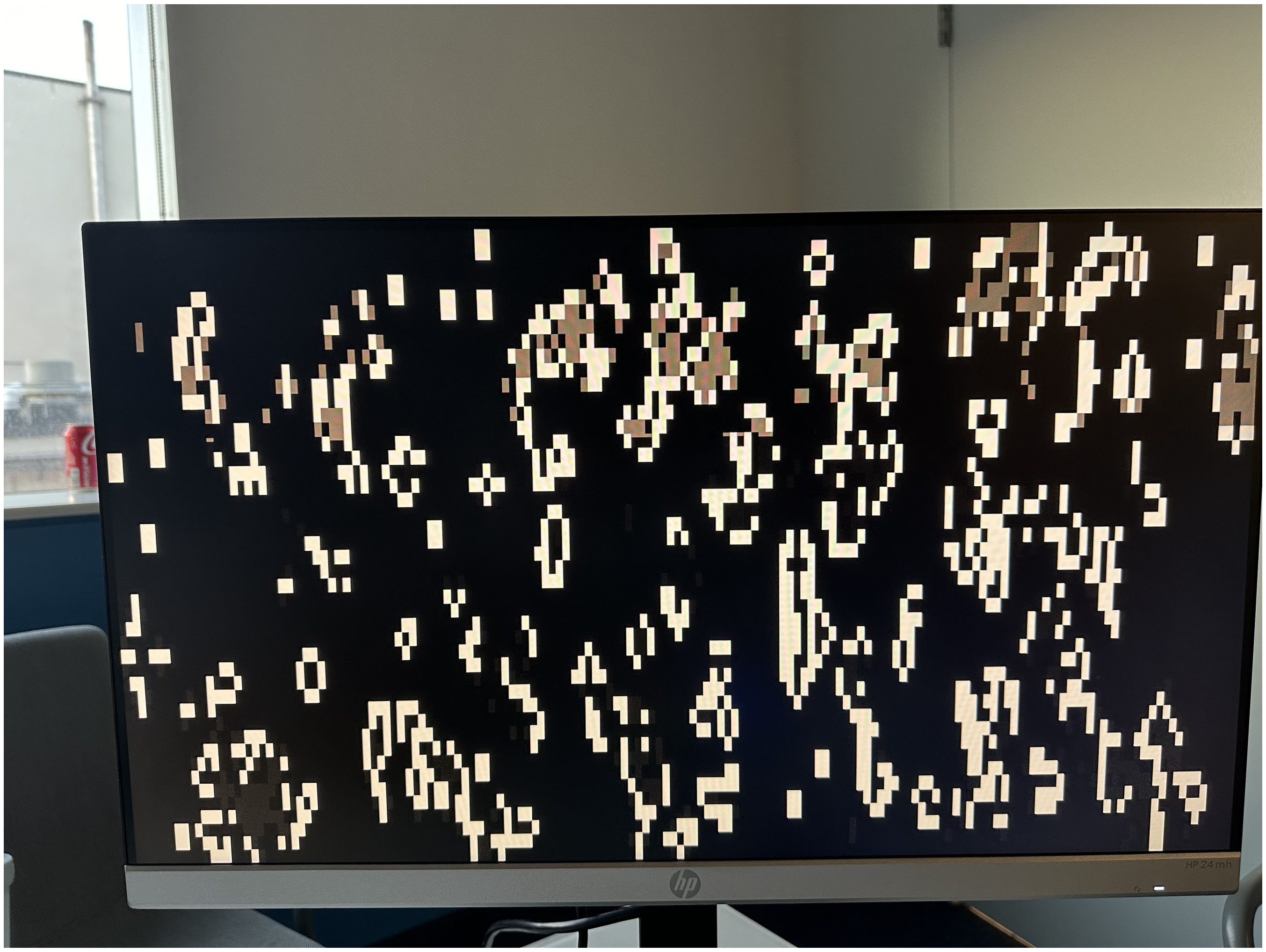
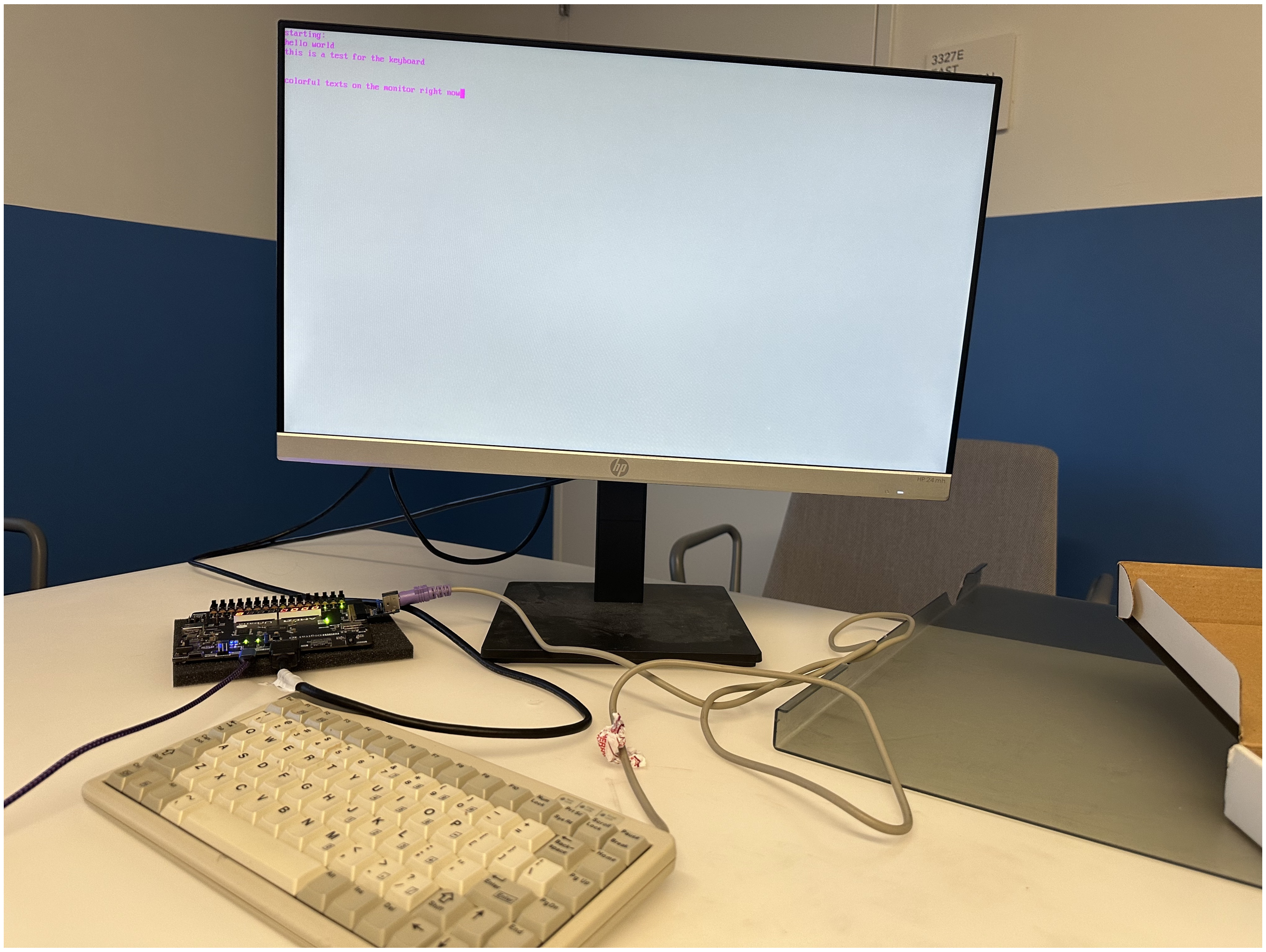


**Figure 5: conway.c**



```
starting:
hello world
this is a test for the keyboard


colorful texts on the monitor right now
```

**Figure 6: kbdtest.c**