Clash of Steel - Project Final Report

Team 5

Tianqi Shen
Han Lyu
Allie Bacholl
Jingyu Lui

https://github.com/AllieBacholl/face-filter

Project Final Report

University of Wisconsin-Madison

Objective

This project aims to fulfill the need of our senior design project proposal and implementation as a cumulative design of a digital system. We wish to apply our skills in computer architecture, RTL design, and novel architectural research and imagination into our project and successfully integrate them into a real world application.

Background

Clash of steel:

We chose to implement a Clash of Steel game on FPGA because while tank battle games are common in software implementations (particularly in Java/ C for embedded systems), we noticed a significant lack of hardware-based implementations using Verilog on FPGAs. This project demonstrates how classic game mechanics can be efficiently realized in hardware, offering potential advantages in performance and parallel processing compared to traditional software approaches.

Traffic sign:

We chose to focus on traffic sign detection to make a meaningful impact in two key areas: autonomous driving and accessibility for people with disabilities. For self-driving cars, accurate sign recognition is crucial for safe navigation, ensuring vehicles obey speed limits, stop signs, and other road rules. At the same time, we wanted this technology to empower individuals with disabilities—helping visually impaired pedestrians detect signs via audio cues or assisting wheelchair users with smart navigation. By improving traffic sign detection, we aim to advance both autonomous mobility and inclusive transportation, creating safer and more independent travel for everyone.

Technical description

RISC-V Processor design:

We developed our processor by using the standard RISC-V RV32I ISA, which contains all the basic operations we needed for our project.

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	slli rd, rs1, uimm	shift left logical immediate	rd = rs1 << uimm
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	srli rd, rs1, uimm	shift right logical immediate	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	srai rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	ori rd, rs1, imm	or immediate	rd = rs1 | SignExt(imm)
0010011 (19)	111	–	I	andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC
0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	[Address]$_{15:0}$ = rs2$_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	[Address]$_{31:0}$ = rs2
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2
0110011 (51)	000	0100000	R	sub rd, rs1, rs2	sub	rd = rs1 – rs2
0110011 (51)	001	0000000	R	sll rd, rs1, rs2	shift left logical	rd = rs1 << rs2$_{4:0}$
0110011 (51)	010	0000000	R	slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	srl rd, rs1, rs2	shift right logical	rd = rs1 >> rs2$_{4:0}$
0110011 (51)	101	0100000	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> rs2$_{4:0}$
0110011 (51)	110	0000000	R	or rd, rs1, rs2	or	rd = rs1 | rs2
0110011 (51)	111	0000000	R	and rd, rs1, rs2	and	rd = rs1 & rs2
0110111 (55)	–	–	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}
1100011 (99)	000	–	B	beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA
1100111 (103)	000	–	I	jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
1101111 (111)	–	–	J	jal rd, label	jump and link	PC = JTA, rd = PC + 4

Figure 1. RV32I ISA

The processor was implemented with the following five stages:

FETCH: Retrieves the next instruction in memory based on the current PC

DECODE: Generates multiple control signals to indicate to future stages what kind of operations are required for this instruction, also reads the values from the register file

EXECUTE: Contains the ALU, performs the required arithmetic operations

MEMORY: If required, writes or reads from data memory

WRITEBACK: If required, writes the value back to the register file

Convolution Neural Network Accelerator:

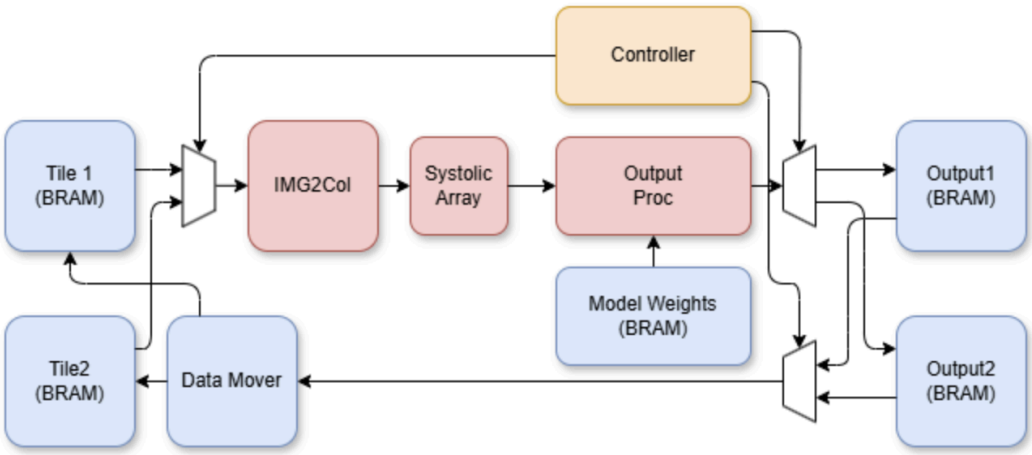

Figure 2. Convolution Neural Network Accelerator Datapath

The accelerator adopts hybrid architecture of both novel designs and traditional datapath. For CNN acceleration, systolic arrays have been considered classic and efficient for data reuse and is inspired by one of the minilab sessions. More custom modules are later on designed to fit in the need of more data reuse and efficient encoding/decoding of data and reordering algorithms such as image-to-columns.

Systolic Array:

The systolic array passes enable signals and its input data to its neighbors while receiving new data from its successors. This way, both weights and inputs can be reused allowing maximized parallelism, which is very critical for a data hungry application like CNN inference. Local FIFO buffers also give space for ramping-up and ramping-down, simplifying control logic of the controller.

Input/Output Tiling:

Due to the limited number of DSPs on FPGA, different layers of inputs or intermediate outputs need to be tiled so that each layer can be broken down into batches and run on the systolic array datapath. Ping-pong strategy is adopted so that there are no read and write conflicts and data is constantly available and clock cycles are not wasted. While one tile is feeding the systolic array, the other tile is being filled with the next batch of input data, the process is switched each time a BRAM tile is consumed. The same strategy is applied to output BRAM and the sizes of the BRAMs are carefully budgeted so that utilization on the FPGA is optimized and fits design needs. Muxes are used to control read and write ports of BRAM tiles. Besides ping-pong strategy, the Tile BRAM also utilized the data overlap of two strides in vertical direction. While the Img2Col unit reads the second and third rows of the tile, the data also fills the other tile so that the workload of data mover can be reduced.

Hardware Img2Col:

One of the highlights of our design is hardware image-to-columns. This module is not found in any open source project and is unique to our design. Considering the data access pattern of each stride during convolution, up to 66% of data is shared among neighboring FIFOs and the pattern is observed so that we can reduce the data consumption from 16 bytes per cycle to as little as 6 bytes per cycle or equivalently 18 bytes per 3 cycles. Not only is the pattern taken advantage of in our design, but it also fits into the tiling strategy perfectly as it defines one of the dimensions of the tile. We creatively gave a solution to high data consumption of AI hardware which is unprecedented in other projects, showcasing our creativity.

Output Processing Unit:

The output processing unit closely matches the relationship of outputs of the systolic array. Each column of the array represents an output channel and different rows of the same column are neighboring outputs. This creates opportunities for pooling operations to happen in a pipelined fashion. First, each column contracts by half, meaning the neighboring outputs are being selected. Then when the next batch of output is ready, two batches can then contract again to achieve 2 by 2 pooling, which is very efficient and increases throughput significantly. Each column as a channel also goes through requantization and ReLU in the pipeline, so that complex work is done by small manageable steps.

Data Mover:

The data mover unit is rather simple and straightforward. It moves the last output or first input from a targeted output BRAM to a targeted tile BRAM, the work is done during the process of the calculation of the last tile, buying time for the accelerator and implementing the Ping-Pong policy.

Controller:

This is a very critical part of the accelerator design. This manages the data flow of different modules, control signal generations and status signal interpretations and is implemented in complex FSM logic for dedication to effective control of the whole system. A lot of multiplication operations appear in the module so it is optimized to be able to allow time sharing of one DSP module among different operational needs. The bottleneck of this module is the complex control logic and address calculation for reading and writing of different RAM modules.

Clash of Steel Game:

We switched to implementing a game because we found it difficult to implement everything with the convolution neural network accelerator within a week, but it still was a good learning experience for us all. We layed out every piece of game state—player and AI tank X/Y positions, bullet X/Y/direction flags, active bits, random seed, input buttons and the live counter—into contiguous 8-bit and 32-bit slots in the FPGA's external memory. Each slot is exposed via a volatile int8_t* or volatile int32_t* pointer at a fixed base address (e.g. 0x1000 for the player X, 0x1006 for the player bullet state, 0x1015–0x101A for inputs, 0x101B for lives). By packing related fields back-to-back, we minimize wasted space and ensure every

read/write is a single, aligned memory access. Boot-time initialization of these slots is done by a tiny UART loader, sidestepping the complexity of RISC-V CPU/DRAM/Peripharals bridges. The following is part of how we define our pointers in address space.

```
volatile int8_t *player_x           = (int8_t*)0x1000;

volatile int8_t *player_y           = (int8_t*)0x1001;

volatile int8_t *ai2_bullet_active   = (int8_t*)0x1014;

volatile int8_t *player_bullet_active= (int8_t*)0x100A;
```

Figure 3. Snippet of pointer definition

We firstly define that the AI player in our game is actually pseudo-AI because it is purely based on probability calculation in our game, but it acts like it has certain intelligence.

The main() routine runs one sequence of instructions so there are no deep call stacks. Therefore, the instruction count and pattern is easier for us to debug and verify. First, we check the “start” pointer to reset button (which is SW[0] in our game), all tank and bullet registers and reload the player’s eight lives. Then, we sample on the direction_register and shoot flag register to check if the player fires. Each AI tank uses a tiny linear conjugate generator (LCG) to select a new heading every other frame. At the top of each loop, we update a 32-bit rand_seed value by multiplying it by 1,103,515,245 and adding 12,345; this scrambles the bits pseudo-randomly. Then rand_seed % 5 is taken to get values 0-4. The result is 0 for ‘stay’, 1 for ‘shift up’, 2 for ‘shift down’, 3 for 'shift left' and 4 means ‘shift right’.

```
rand_seed = rand_seed * 1103515245 + 12345;

    int8_t ai1_dir = rand_seed % 5;

    int8_t ai1_dx = 0, ai1_dy = 0;

    if (ai1_dir == 1)      ai1_dy = -1;

    else if (ai1_dir == 2) ai1_dy =  1;

    else if (ai1_dir == 3) ai1_dx = -1;

    else if (ai1_dir == 4) ai1_dx =  1;
```

Figure 4. Example of pseudo-AI Direction Generation

By decoding individual integers as offsets of (dx,dy), (0,0), (0,-1), (0,+1), (-1,0), or (+1,0), each AI tank roams the battlefield unpredictably, but with a fully deterministic, RISC-V-friendly RNG. Then, we immediately write the new bullet’s position and delta into its reserved pointers. We then advance each AI tank by feeding the LCG-based 32-bit RNG to the 25% chance shooter and writing in any new AI bullet generation. With the movement triangle, we can calculate the next X/Y coordinates of each tank, sandwich it within the 0-99 play area, and force avoidance of tank-to-tank collisions so that no two tanks

share a slot. Active bullets follow suit, moving one step per frame and deactivating if they stray from the screen. Finally, we addressed interaction issues: any colliding bullets will disable each other, and bullet-tank impacts will either respawn an AI or reduce the player's life points - we'll flip a single game_over flag when the life points are zero. This flat, pointer-centered loop keeps the number of instructions fixed and allows for direct integration with bare-metal graphics drivers or schedulers. Below is our game logic diagram and a description of each possible situation.

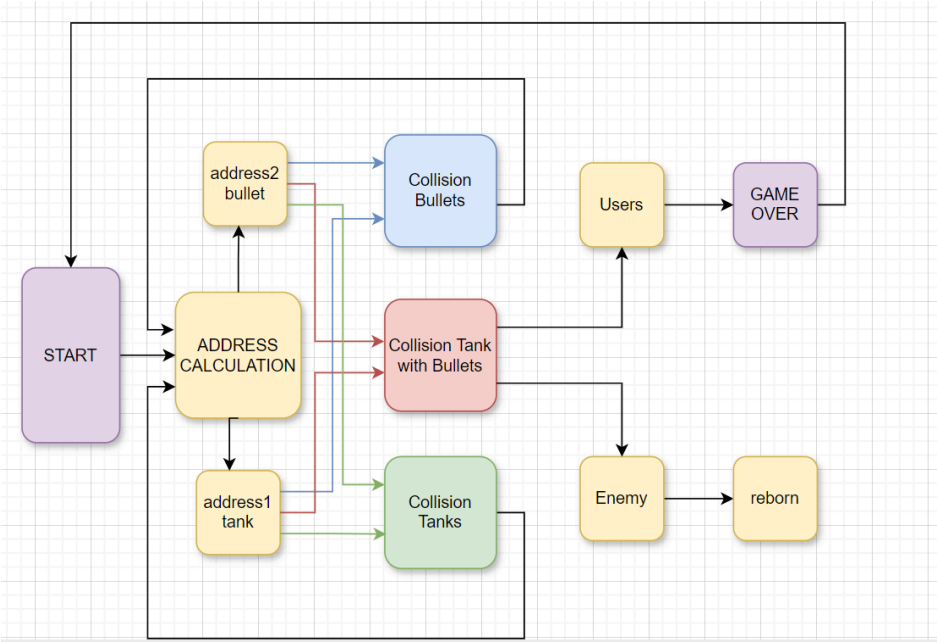

Figure 5. Game State Logic

Start – Position Calculation & State Setup

We begin each frame in START, where the tank's current screen coordinates and each bullet's coordinates in video memory are computed. Using simple C pointer arithmetic, the tank's X/Y offsets and each bullet's X/Y offsets are translated into linear memory positions. The player's remaining lives (starting at 10) are tracked and the LED indicators are updated on the FPGA display to match the number of remaining lives. With the positions and status flags updated, they are then fed into the three collision checks in parallel.

Situation A – Bullet Collision

First, in Collision Bullets, each bullet's memory position is compared against the other bullet positions. If they overlap, both bullets are cancelled out and removed from the game.

Situation B – Bullet Strikes a Tank

Here, the tank's memory position is tested against each bullet's position. On a match for the player, a life is decremented and the matching LED is turned off. Provided the player still has lives left, the tank's position is reset. Once lives hit zero, the game transitions into the END state and displays "Game Over."

Otherwise, the game returns to the START stage on the next tick. On a match for one of the enemy tanks, the enemy tank is removed from the game and respawned with invulnerability for two seconds. The player's score is increased, which is updated on the seven-segment display, if it was one of the player's bullets that killed the enemy tank.

Situation C – Tank‑to‑Tank Bumping

Assuming the player survives, we enter Collision Tanks, comparing our tank's position against each enemy tank's position. When two tanks overlap, the enemy tank is marked as destroyed, incrementing the FPGA scoreboard by one. The enemy tank's "alive" flag is cleared, and it is enqueued for a delayed rebirth. The enemy stays off-screen until its respawn timer expires, then it is initialized but invulnerable for two seconds to have a smooth respawn.

End – Loop or Game Over

If neither Situation A nor B forces zero lives, and any Situation C respawns are complete, the game loops back to START for the next frame. Once the player's lives drop to zero, the game enters the END stage: halting the position‑calc loop, rendering the final "Game Over," and leaving all LEDs dark except the "0 lives" indicator. This five-state cycle runs entirely in C on our RISC-V core, driving collisions, health, LEDs, and the FPGA scoreboard in perfect sync.

Evaluation highlights

Early on, a considerable time was spent designing and verifying the RISC-V processor which resulted in a functional processor that we were able to reuse for the Clash of Steel game. A great deal of time was also spent on designing the accelerator and optimizing it to work around the restrictions of the FPGA. With some more time, the accelerator could be fully implemented and realized.

If we were to redo this project, we would focus solely on implementing the accelerator or the processor instead of both as we originally planned. We would also implement the processor with the multiplication and floating-point extensions if we planned to run the CNN on it.

Challenges, workarounds, & lessons learned

The first difficulty we encountered was SDRAM bandwidth. We had a difficult time using the SDRAM to display full color images through VGA while transferring additional data such as model weights at the same time. We tried many different modifications to SDRAM, such as changing the parameters and the SDRAM clock frequency.

Additionally, we also wanted to preload data onto the SDRAM. We explored using the HPS side of the board to accomplish this, but ultimately decided to use UART to initialize the SDRAM since we were running out of time.

The limited SDRAM bandwidth limited the size of the model to what we could fit on chip if we also wanted decent display quality. This is why we initially switched from facial detection to traffic sign detection as the display quality requirements are not as high as a face filter and the model was much smaller. To work around the bandwidth issue, the camera video was displayed as grayscale so that only one FIFO from the SDRAM controller was needed. Additionally, the traffic sign pictures were reduced to eight bits per pixel (three red, three green, and two blue) to reduce transfer time over UART to SDRAM and to reduce the amount of SDRAM bandwidth required.

Another challenge is to set up the compiler and linker properly to fit into our memory model of the processor. We invested considerable effort into customizing the linker script (link.ld) to precisely control memory allocation. This involved defining separate memory regions and carefully mapping each section—such as .text, .data, .bss, .rodata, and custom segments—to appropriate physical addresses. We also handled variables of various types (int64_t, int32_t, int8_t, etc.) with care, ensuring correct alignment and addressability. Because the math required for the convolution neural network results in 64 bit integers, we encountered many difficulties with overflow that we attempted to work around by breaking the data down into smaller chunks but ran out of time to complete. In the future, we would try to run the convolution neural network on a processor that supports floating-point arithmetic.

A key aspect was the use of GCC's __attribute__((section("..."))) syntax to explicitly place const static variables (model parameters or lookup tables) into designated memory sections, such as DDR or ROM. This allowed us to separate read-only data from writable regions and optimize memory usage effectively. Additionally, we ensured sufficient stack space was reserved for nested function calls and local buffers, avoiding stack overflows in deeply nested computations. The result was a memory-aware application that leveraged external memory efficiently on our FPGA.

Another significant challenge was the implementation, verification, and synthesis of the accelerator. Due to its novelty and unique implementation. It is hard to estimate the LUT usage, DSP usage, as well as on-chip memory utilization. The complexity of the system and architecture makes the implementation of the controller extremely challenging. A considerable amount of time on debugging, redesigning, researching, and data access interpretation and observation is required to come up with the most resource friendly plan on a constrained FPGA board. The system takes a while for team members to familiarize themselves with and thus takes a large effort for one to effectively contribute.

Contributions of individuals

Tianqi Shen:

- Processor Implementation and Processor Verification
- Compiler and Assembler modification
- Convolution Neural Network Model Export
- ROM/SDRAM VGA Image Display
- Image Resizer Implementation
- The Full System Integration
- Poster

Han Lyu

- Project Blueprint Design and Resource Budgeting
- Design Researching and Implementation of CNN accelerator
- Processor Modification and Extension
- Compiler and Assembler modification
- Convolution Neural Network Model Training and Fine Tuning
- The Full System Integration
- Game Programming and Software Debugging

Allie Bacholl:

- Processor Execution, Memory, and Writeback stage
- Module and Python Script to Preload Images/Data onto SDRAM through UART
- Collected, Sized, and Converted to Hex all 47 Traffic Sign Images, Developed Matlab and Python Scripts to Automate
- Image Resizer Module to Convert Camera Image to Grayscale 32x32 image
- Compiled and Assembled Code with the RISC-V GNU Toolchain
- Full System Integration
- Poster
- Website

Jingyu Liu:

- Debugged the RISC-V processor and added additional essential components.
- Developed the compiler and assembler toolchain to generate and link C programs for our processor
- Authored Verilog testbenches for both the risc-v processor instructions and C applications to validate instruction correctness
- Implemented the VGA driver and rendering logic to draw moving tanks and bullets on screen
- Integrated custom IP cores with the processor, synthesized the full system on an FPGA board
- Poster