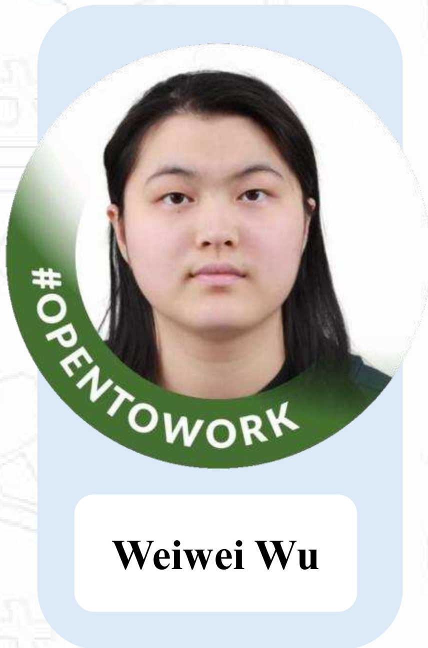
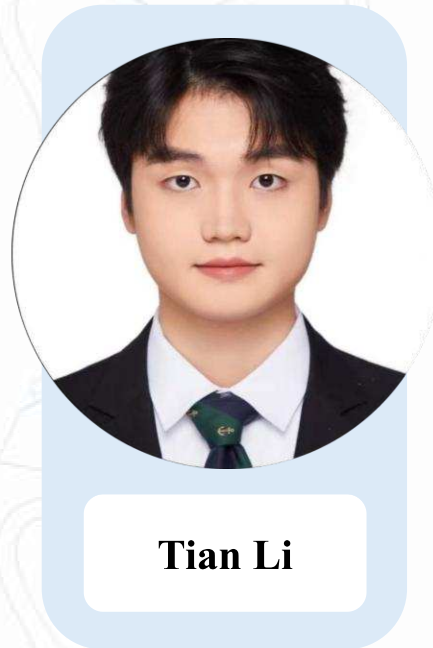
FPGA-Based LeNet-5 CNN Accelerator for Handwritten Digit Recognition

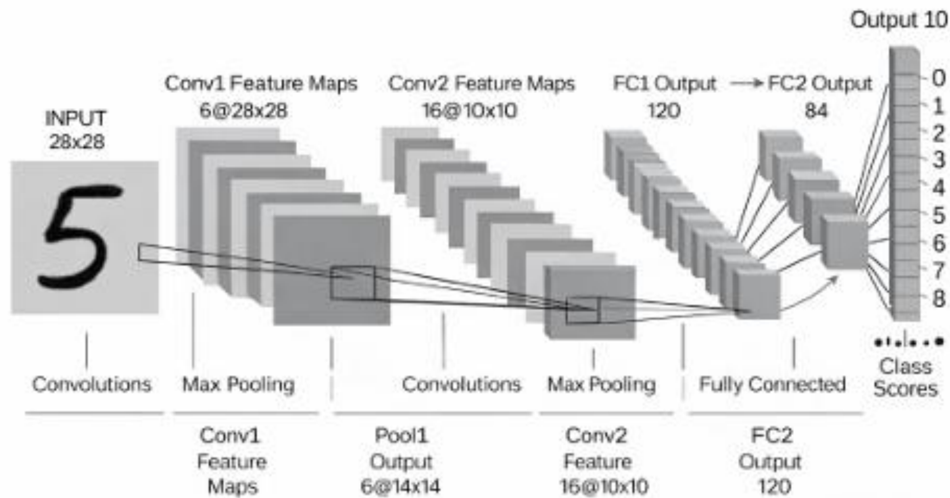
Instructor: Prof. Stephen Edwards
Teaching Assistant: Michael Lippe
Date:05/12/2026

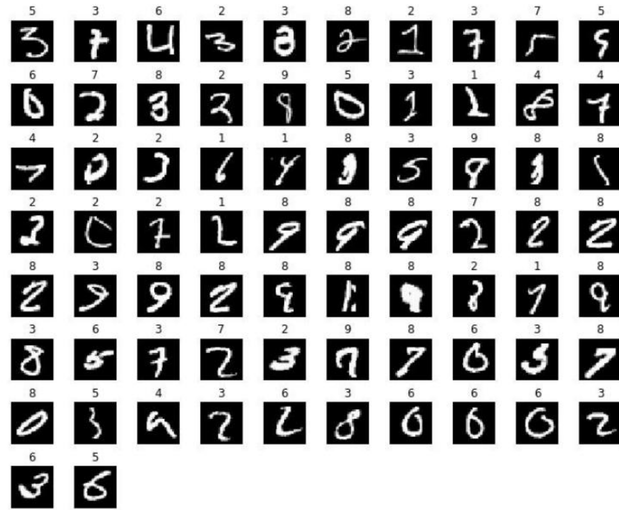
Group member

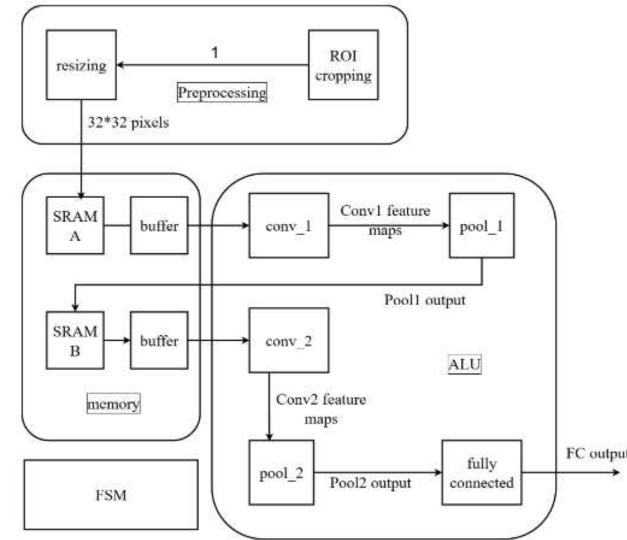




Lenet-5 based CNN architecture







LeNet-5 is one of the earliest convolutional neural networks. It leverages convolution, weight sharing, and pooling to reduce computation, making it highly suitable for image recognition and hardware acceleration.

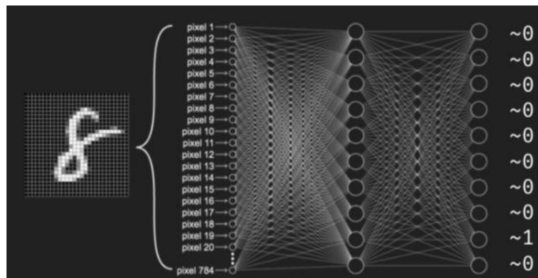
MNIST is a benchmark dataset of 28×28 grayscale handwritten digits, widely used for training and evaluating image classification models.

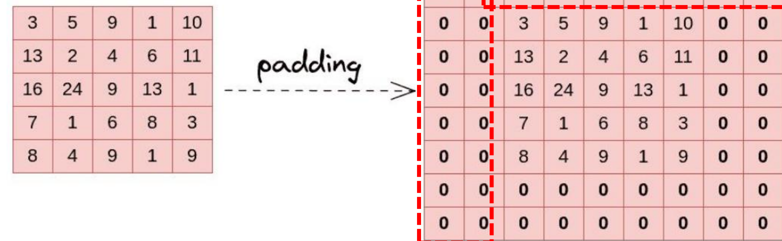
This figure shows the overall CNN accelerator **architecture**. There will be some minor refinements such as adding a quantization module later to prevent data width explosion.

32*32

28*28

output mode:





Parameter Preparation

Training and quantization

- CNN trained entirely in full FP32 precision in PyTorch
- Final trained weights and biases quantized only at export stage
- Exported as signed 8-bit .hex files for FPGA BRAM initialization

Validation Hardware simulated python script

- Find the best shifts for each layer
- Accuracy: 98.2%
- Confusion Matrix:

Software–Hardware Comparison

- FP32 software accuracy: 98.40%
- Hardware simulation accuracy: 98.20%
- Prediction mismatch rate: 1.60%

Confusion Matrix

	Digit 0	Digit 1	Digit 2	Digit 3	Digit 4	Digit 5	Digit 6	Digit 7	Digit 8	Digit 9
Digit 0	85	0	0	0	0	0	0	0	0	0
Digit 1	0	121	1	1	0	0	0	1	2	0
Digit 2	0	0	112	1	0	0	0	2	1	0
Digit 3	0	0	0	107	0	0	0	0	0	0
Digit 4	0	0	1	0	98	0	0	1	3	7
Digit 5	0	0	0	1	0	85	0	0	1	0
Digit 6	2	0	0	0	0	1	84	0	0	0
Digit 7	0	0	0	2	0	0	0	97	0	0
Digit 8	0	0	1	0	0	0	0	0	88	0
Digit 9	0	0	0	0	0	0	0	0	3	91

layer dimension

Layer Name	Input Dim	Operation	Output Dim	Weights (Stored as int8)
Input	32×32×1	Raw Pixel Data	32×32×1	0
Conv1	32×32×1	5×5 Kernel, 6 Filters	28×28×6	150 weights + 6 biases
Pool1	28×28×6	2×2 Max Pooling	14×14×6	0
Conv2	14×14×6	5×5 Kernel, 16 Filters	10×10×16	2,400 weights + 16 biases
Pool2	10×10×16	2×2 Max Pooling	5×5×16	0
Flatten	5×5×16	Unspool to 1D Vector	400×1	0
FC1	400×1	Matrix Mult (120 nodes)	120×1	48,000 weights + 120 biases
FC2	120×1	Matrix Mult (84 nodes)	84×1	10,080 weights + 84 biases
FC3 (Out)	84×1	Matrix Mult (10 nodes)	10×1	840 weights + 10 biases

Web demo & FPGA

INPUT
Drawing Pad

canvas

Stroke 22 px
Invert Off
function

Predict | Upload | Clear

Label Choose | Save

Not saved

OUTPUT
Inference Results

prediction

FPGA RESULT FROM HPS FPGA PIPELINE
5
Top-1 probability: 99.98%
Top scores: 5 (score 29399030, 99.98%) | 8 (score 12469122, 0.02%) | 9 (score 5215385, 0%)

SOFTWARE QUANTIZED RESULT FROM FPGA QUANTIZED REFERENCE
5
Top-1 probability: 99.98%
Top scores: 5 (score 29399030, 99.98%) | 8 (score 12469122, 0.02%) | 9 (score 5215385, 0%)

SOFTWARE RESULT FROM WEIGHTS BIAS 422
5
Top-1 probability: 99.98%
Top scores: 5 (score 15783814001780486, 99.98%) | 8 (score 6696598281227069, 0.02%) | 9 (score 2798222339520441, 0%)

preprocessing

PROCESSED INPUT
32x32 Preview

FPGA DEBUG
Final Logits

Showing final FPGA classification logits returned by the HPS/FPGA pipeline. FPGA core timing: 454966 cycles @ 50.00 MHz = 9.0993 ms

FINAL LOGITS

logits

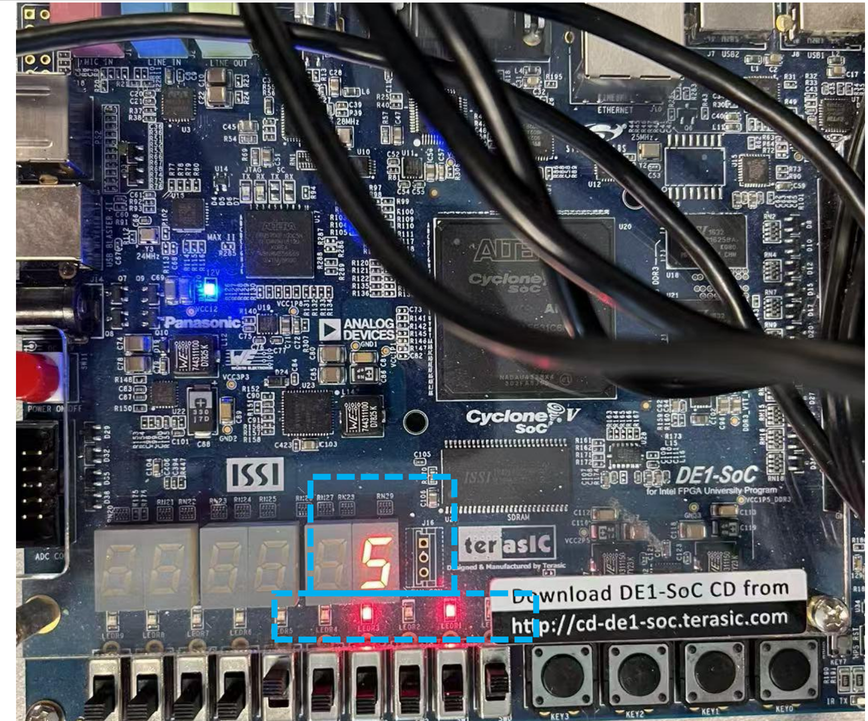
```
digit 0: -21823855
digit 1: -29093572
digit 2: -18941875
digit 3: 2706930
digit 4: -21822892
digit 5: 29399030
digit 6: -4557860
digit 7: -32246193
digit 8: 12469122
digit 9: 5215385
```

speed

PERFORMANCE
Speed Compare

CPU exact	33.68 ms
CPU quant	34.89 ms
FPGA path	12.65 ms
FPGA core	9.10 ms
Speedup	2.66x

Exact | Quant | FPGA

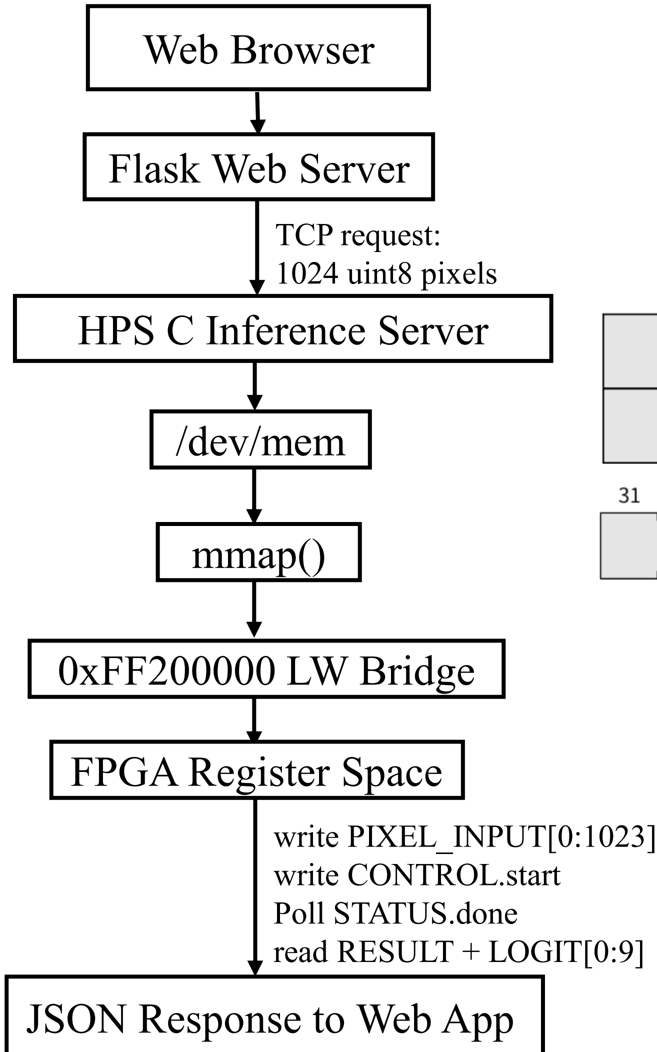


The first digital tube displays the numbers from 0 to 9

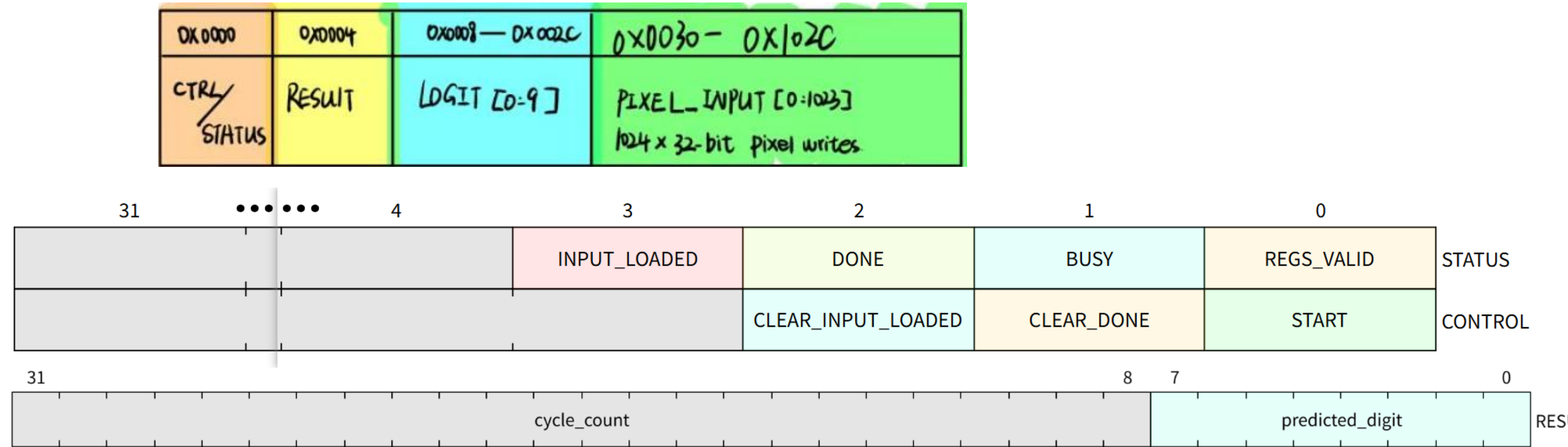
LED0 represents the digit 0, while LED1, LED2, LED3, and LED4 represent the bits of the 8421 BCD code from the least significant bit to the most significant bit, respectively.

Software Interface

Software-Hardware Calling Flow



Memory-Mapped Register Layout



User space

0x102C - 0x0000 = 0x102C bytes

0x102C = 0x1000 + 0x2C

= 4096 + 44

= 4140 bytes

4140 / 1024 = 4.04296875 KB

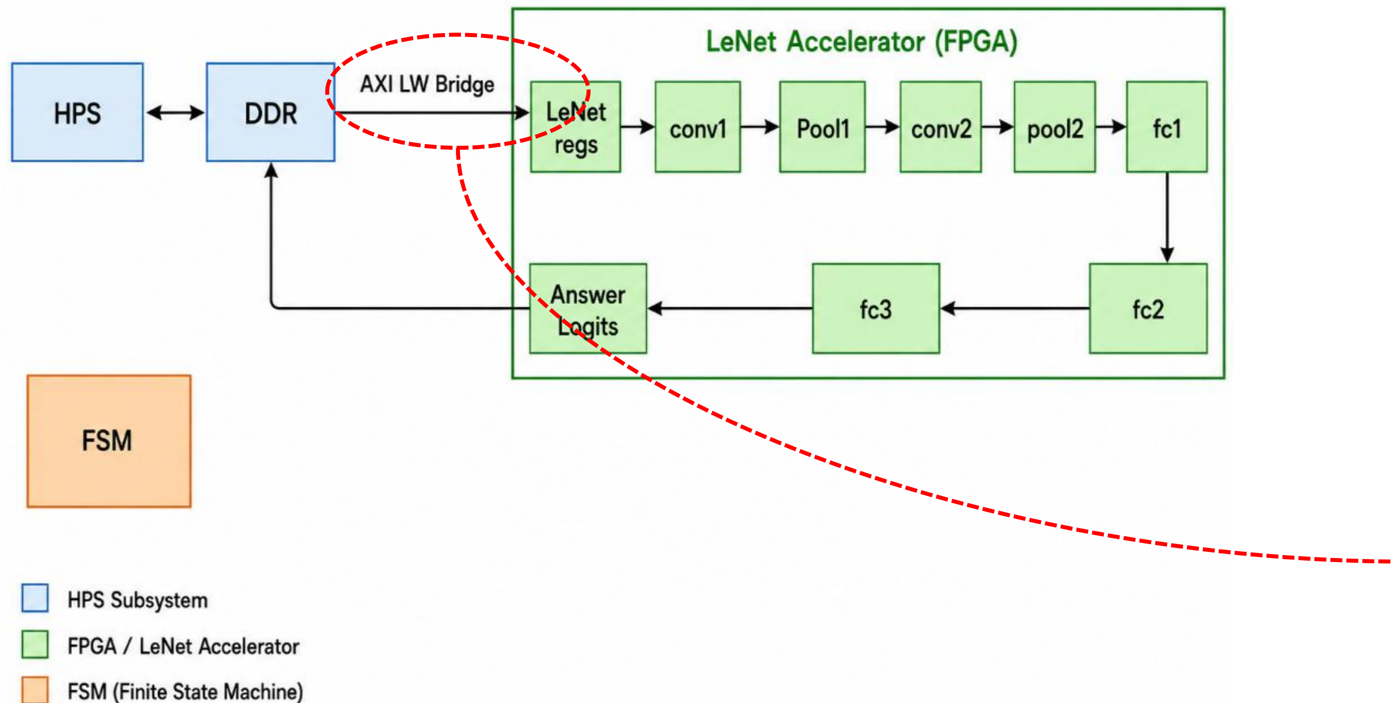
Register Map

Offset	Name	Access	Current Function
0x0000	CONTROL / STATUS	RW	Write: bit0=start, bit1=clear_done, bit2=clear_input_loaded。 Read: bit0=regs_valid, bit1=busy, bit2=done, bit3=input_loaded
0x0004	RESULT	RO	bit[3:0]=predicted_digit
0x0008	LOGIT_0	RO	Final logit score for digit 0, signed int32
0x000C	LOGIT_1	RO	Final logit score for digit 1, signed int32
0x0010	LOGIT_2	RO	Final logit score for digit 2, signed int32
0x0014	LOGIT_3	RO	Final logit score for digit 3, signed int32
0x0018	LOGIT_4	RO	Final logit score for digit 4, signed int32
0x001C	LOGIT_5	RO	Final logit score for digit 5, signed int32
0x0020	LOGIT_6	RO	Final logit score for digit 6, signed int32
0x0024	LOGIT_7	RO	Final logit score for digit 7, signed int32
0x0028	LOGIT_8	RO	Final logit score for digit 8, signed int32
0x002C	LOGIT_9	RO	Final logit score for digit 9, signed int32
0x0030 - 0x102C	PIXEL_INPUT	RW	Runtime input image window. One 32-bit word per pixel, only bits[7:0] used. Reads return the stored pixel value in bits[7:0]

RESULT stores the final digit predicted by the FPGA. Only bits [3:0] are used, representing a value from 0 to 9; the upper bits are unused.

PIXEL_INPUT[0:1023] is a memory-mapped input window used to send a 32x32 grayscale image from the HPS to the FPGA. Each pixel uses one 32-bit word, with only bits [7:0] storing the uint8 pixel value. After all 1024 pixels are written, the FPGA sets input_loaded, indicating that inference can start.

Top-Level System Block Diagram



mapped using mmap

```
base = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED
```

```
regs = (volatile uint32_t *)base;
```

- **LW HPS-to-FPGA AXI Bridge** : Lightweight 32-bit memory-mapped interface from HPS to FPGA
- **hps_lenet_regs**: Custom register slave exposing status and pooled outputs to the HPS
- **HPS**: Runs Linux user programs and reads FPGA debug registers
- **BRAMs**: Store input image, kernels, intermediate feature maps, and pooled outputs

The custom FPGA register interface is accessed from the HPS through the lightweight HPS-to-FPGA bridge at base address 0xFF200000. In software, this address is mapped through /dev/mem, and the registers are read as memory-mapped words.

```
#define LW_BRIDGE_BASE 0xFF200000u
```

FSM

Overall CNN FSM

Current State Group	Transition Condition	Next State Group
ST_IDLE	start == 1	Conv1 states
Conv1 states	out_ch == 5, out_y == 27, out_x == 27, term_idx + 3 >= 25	Pool1 states
Pool1 states	pool_ch == 5, pool_y == 13, pool_x == 13, pool_read_idx == 3	Conv2 states
Conv2 states	out_ch == 15, out_y == 9, out_x == 9, term_idx + 3 >= 150	Pool2 states
Pool2 states	pool_ch == 15, pool_y == 4, pool_x == 4, pool_read_idx == 3	FC1 states
FC1 states	neuron_idx == 119, term_idx + 3 >= 400	FC2 states
FC2 states	neuron_idx == 83, term_idx + 3 >= 120	FC3 states
FC3 states	neuron_idx == 9, term_idx + 3 >= 84	ST_ARGMAX
ST_ARGMAX	fixed 1 cycle	ST_DONE
ST_DONE	fixed 1 cycle	ST_IDLE

The FSM does not wait for a separate read-done signal. Since the BRAM and parameter ROM are synchronous memories with a deterministic one-cycle read latency, the FSM uses fixed one-cycle transitions from READ to ACCUM. The actual conditional transitions happen in the ADD or WRITE states, where counters such as term_idx, out_x, out_y, out_ch, and neuron_idx determine whether to continue the current layer or move to the next layer.

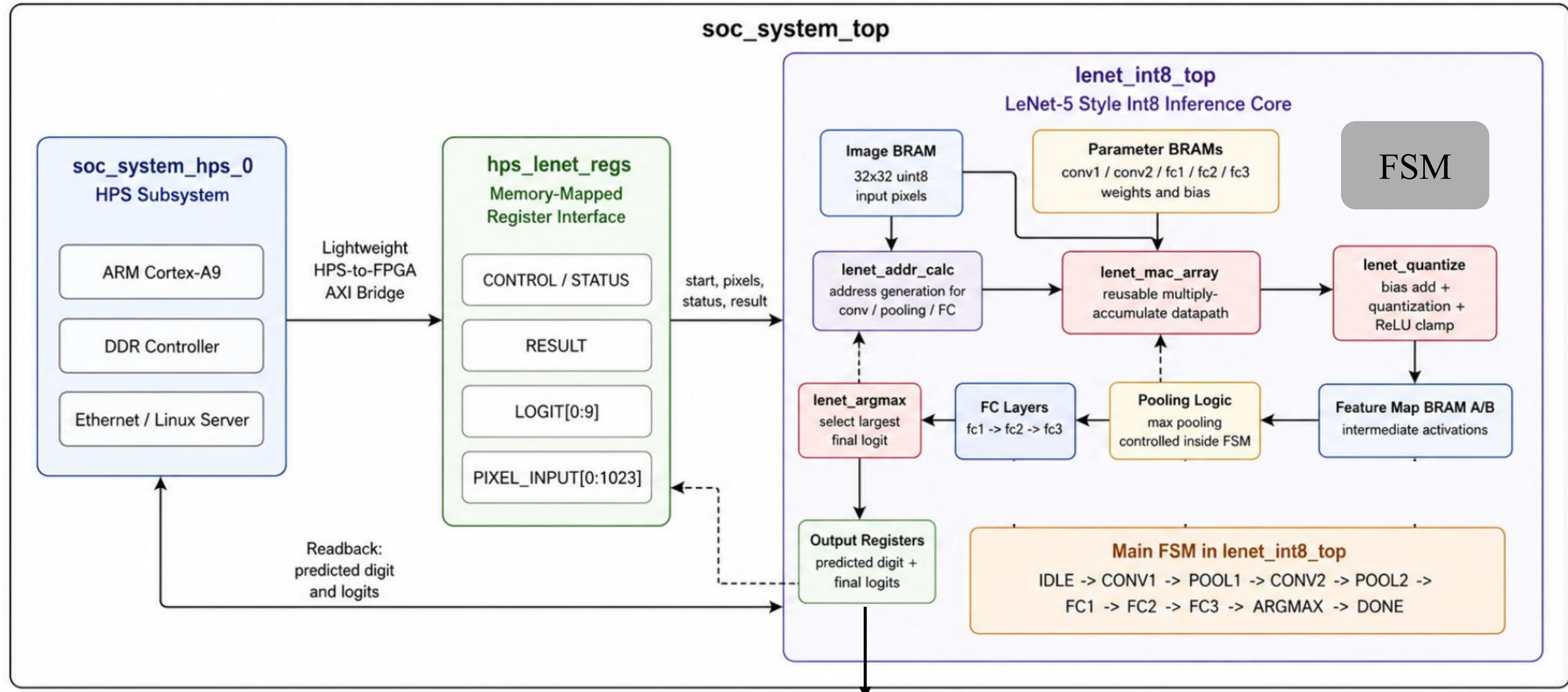
Conv / FC Micro-FSM

State	Action	Transition Condition	Next State
INIT	Clear acc_reg, set term_idx = 0, term_word_idx = 0	fixed 1 cycle	READ
READ	Send BRAM/ROM read address	fixed 1 cycle	ACCUM
ACCUM	Save mac_partial_sum int o mac_partial_sum_r	fixed 1 cycle	ADD
ADD	acc_reg += mac_partial_sum_r	term_idx + MAC_LANES < TERM_LIMIT	READ
		term_idx + MAC_LANES >= TERM_LIMIT	WRITE
WRITE	Write one output value	output counter not at layer limit	INIT
		output counter reaches layer limit	next layer

Pooling Micro-FSM

State	Action	Transition Condition	Next State
POOL_INIT	Set pool_read_idx = 0, clear pool_max_reg	fixed 1 cycle	POOL_READ
POOL_READ	Send feature-map BRAM read address	fixed 1 cycle	POOL_ACCUM
POOL_ACCUM	Update max value: pool_max_reg = max(pool_max_reg, data)	pool_read_idx < 3	POOL_READ
POOL_ACCUM	Update max value	pool_read_idx == 3	POOL_WRITE
POOL_WRITE	Write one max-pooled output	output counter not at layer limit	POOL_INIT
POOL_WRITE	Write one max-pooled output	output counter reaches layer limit	next layer

Verilog Module Organization

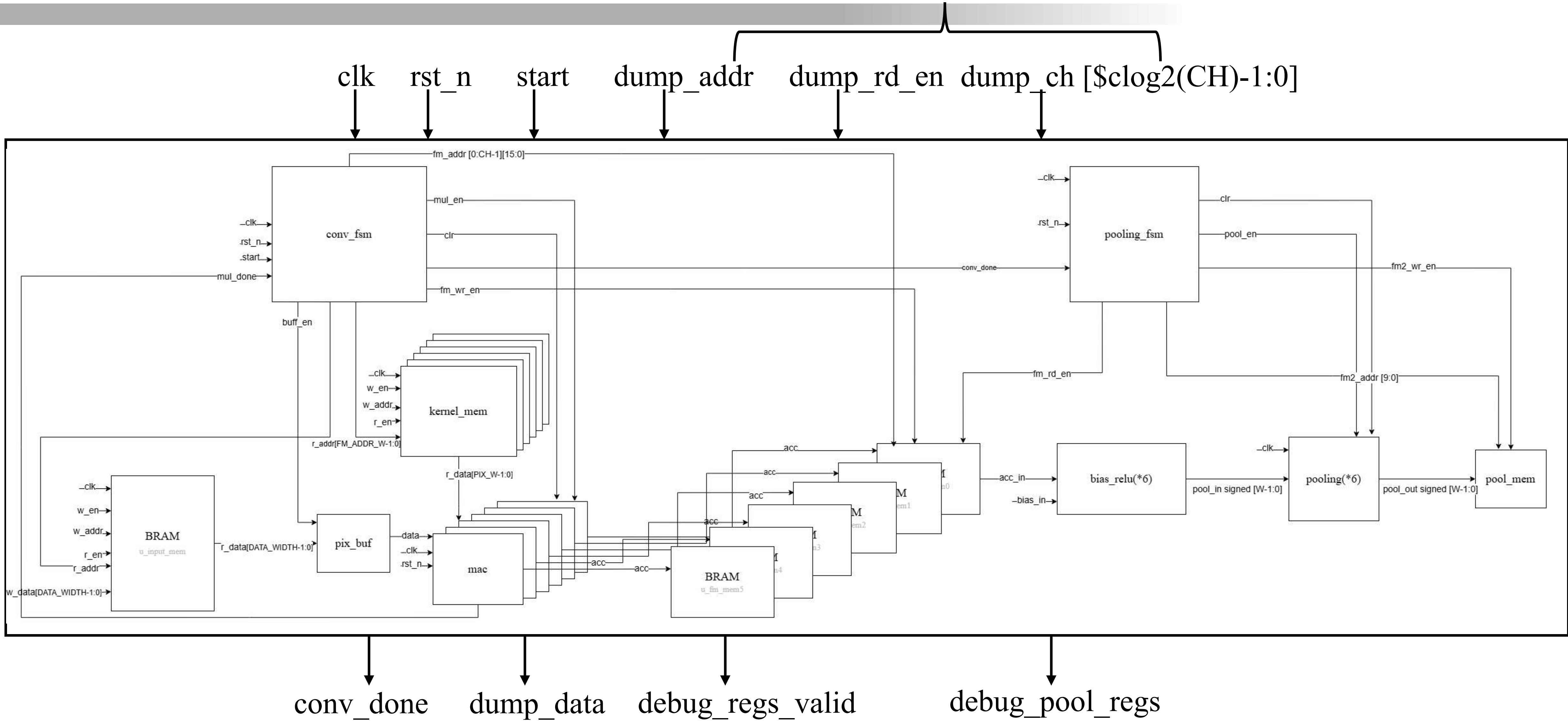


Module Interface Summary

Module	Key Inputs	Key Outputs	Function
soc_system_top	CLOCK_50, KEY, HPS AXI-LW signals	LEDR, HPS bridge connections	Top-level integration of HPS subsystem, register interface, and FPGA LeNet accelerator
soc_system_hps_0	HPS clocks/resets, AXI-LW master interface	HPS-to-FPGA lightweight bridge signals	HPS subsystem running Linux and the C inference server
hps_lenet_regs	AXI-LW read/write signals, accel_busy, accel_done, predicted_digit, regs_flat, pixel_rd_data	start_pulse, clear_done_pulse, pixel_wr_en, pixel_wr_addr, pixel_wr_data, AXI readback signals	Memory-mapped register interface exposing CONTROL/STATUS, RESULT, LOGIT[0:9], and PIXEL_INPUT[0:1023] to HPS
lenet_int8_top	clk, rst_n, start, pixel_wr_en, pixel_wr_addr, pixel_wr_data, pixel_dbg_rd_en, pixel_dbg_rd_addr	busy, done, predicted_digit, logits_flat, pixel_dbg_rd_data	Main LeNet-5 style int8 inference core with FSM control, convolution, pooling, FC layers, and output generation
BRAM	clk, w_en, w_addr, r_en, r_addr, w_data	r_data	Generic synchronous memory used for input image, feature maps, and parameter storage
lenet_addr_calc	out_ch, out_y, out_x, pool_ch, pool_y, pool_x, term_idx, pool_read_idx	conv1_src_addr, conv1_dst_addr, pool1_src_addr, pool1_dst_addr, conv2_src_addr, conv2_dst_addr, pool2_src_addr, pool2_dst_addr	Generates addresses for convolution and pooling memory accesses
lenet_mac_array	activations_flat, weights_flat, valid_mask	partial_sum	Reusable multiply-accumulate datapath for convolution and fully connected layers
lenet_quantize	value_in, shift_right	value_out	Applies fixed-point right shift, saturation, and ReLU-style clamp to activation values
lenet_argmax	logits_flat	digit	Selects the largest final logit and outputs the predicted digit

Block diagram of conv1_top

The "read address selection signal" used when reading the pooling result from HPS/registers



RTL Simulation and Testbench Verification

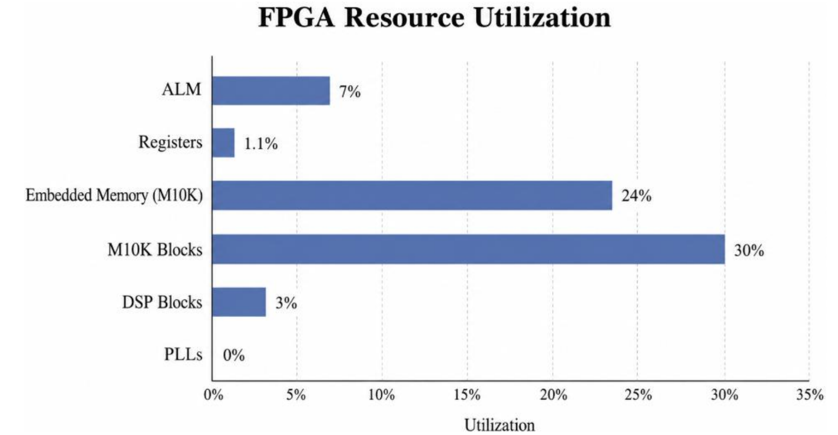
Signal	Meaning	What to Observe
start	Starts one inference run	One-clock pulse after input pixels are loaded
busy	CNN core is running	Stays high during convolution, pooling, and FC layers
done	Inference is finished	Goes high when prediction and logits are valid
cycle_count	Number of cycles used by one inference	Final value is 454,966 cycles

Block RAM usage

RAM Block	Local BRAM Address Range	HPS / LW Bridge Address Range	Data Width	Used Depth	Allocated Depth	Allocated Bits	Function
Image BRAM	0x000 - 0x3FF	LW offset 0x0030 - 0x102C; HPS physical 0xFF200030 - 0xFF20102C	8-bit	1024 each	1024 each	8,192 each / 24,576 total	32x32 input pixels, replicated for 3-lane parallel reads
Feature Map BRAM A	0x0000 - 0x1FFF	FPGA internal only	16-bit	up to 4704 each	8192 each	131,072 each / 393,216 total	Intermediate activation buffer A
Feature Map BRAM B	0x0000 - 0x1FFF	FPGA internal only	16-bit	up to 4704 each	8192 each	131,072 each / 393,216 total	Intermediate activation buffer B
Conv1 Param ROM	0x00 - 0x3B	FPGA internal only, initialized from hex file	24-bit	60 packed words	60	1,440	Conv1 weights + bias, 156 logical 8-bit params packed by 3
Conv2 Param ROM	0x000 - 0x32F	FPGA internal only, initialized from hex file	24-bit	816 packed words	816	19,584	Conv2 weights + bias, 2416 logical 8-bit params packed by 3
FC1 Param ROM	0x0000 - 0x3F47	FPGA internal only, initialized from hex file	24-bit	16200 packed words	16200	388,800	FC1 weights + bias, 48120 logical 8-bit params packed by 3
FC2 Param ROM	0x000 - 0xD73	FPGA internal only, initialized from hex file	24-bit	3444 packed words	3444	82,656	FC2 weights + bias, 10164 logical 8-bit params packed by 3
FC3 / Output Param ROM	0x000 - 0x121	FPGA internal only, initialized from hex file	24-bit	290 packed words	290	6,960	Output layer weights + bias, 850 logical 8-bit params packed by 3

FPGA Resource Utilization

Resource Type	Available Resources	Used Resources	Utilization
Logic Elements (LE)	85,000	~6,300 equivalent	~7%
Adaptive Logic Modules (ALM)	32,070	2,390	7%
Registers	128,300	1,429	1.1%
Embedded Memory (M10K)	3,970 Kb	946 Kb	24%
M10K Blocks	~397	119	30%
MLAB	480	0	0%
Variable-Precision DSP Blocks	87	3	3%
18x18 Multipliers	174	6 equivalent	3%
FPGA PLLs	6	0	0%
HPS PLLs	3	HPS subsystem only	N/A



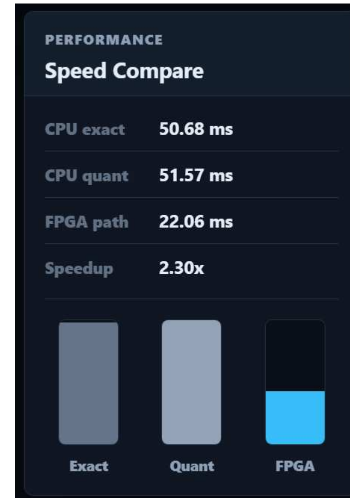
- So, we can use more resources to make a tradeoff between speed and hardware consumption

Type	num	Function
Image BRAM	x3	load pixel x3 (MAC_LANES=3)
Feature Map A	x3	load fm
Feature Map B	x3	load fm
Parameter PACKED_ROM	x5	load bias of Conv1,2. FC1,2,3 weight/ bias.

Ping-Pong structure

Speed - Cycle

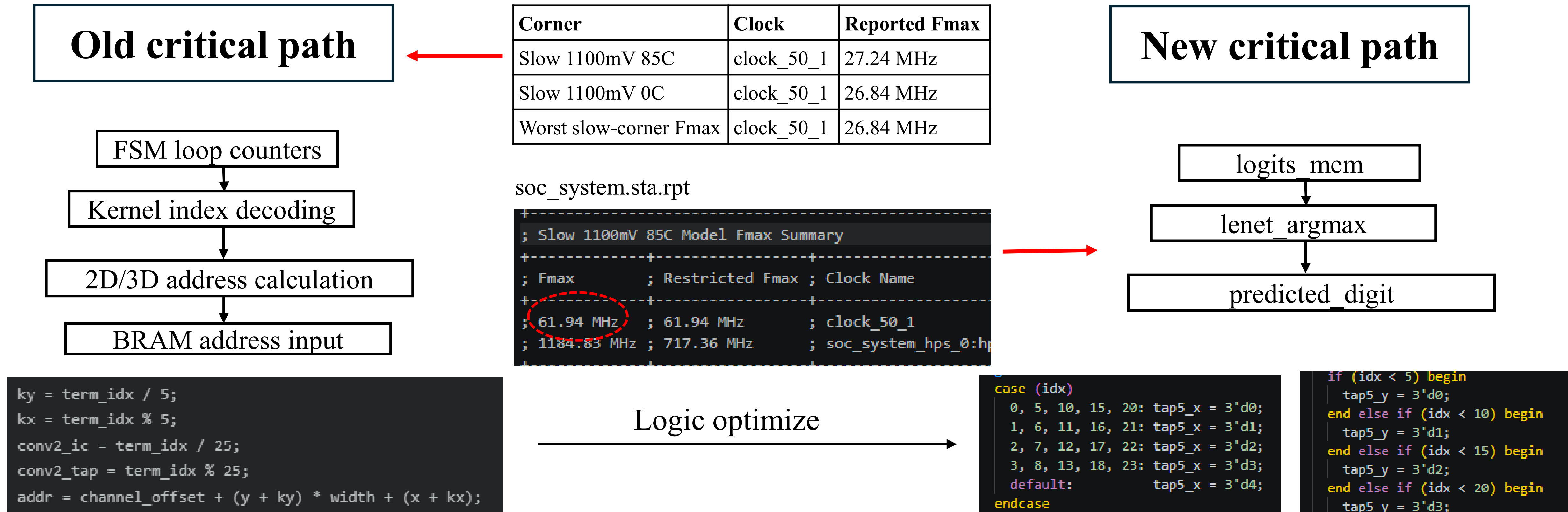
Stage	Output Count	Each output need/cycle	Cycles
Start	1	1	1
Conv1	6 * 28 * 28 = 4704	1 + ceil(25/3)*3 + 1 = 29	136,416
Pool1	6 * 14 * 14 = 1176	1 + 4*2 + 1 = 10	11,760
Conv2	16 * 10 * 10 = 1600	1 + ceil(150/3)*3 + 1 = 152	243,200
Pool2	16 * 5 * 5 = 400	10	4,000
FC1	120	1 + ceil(400/3)*3 + 1 = 404	48,480
FC2	84	1 + ceil(120/3)*3 + 1 = 122	10,248
FC3	10	1 + ceil(84/3)*3 + 1 = 86	860



Speedup
= CPU exact time / FPGA path time
= 50.68 ms / 22.06 ms
= 2.30x

Optimization	Idea	Effect	Cost
Add parallel MAC units	Compute multiple multiply-accumulate terms per cycle	Fewer cycles per output	More DSP / ALM
Parallelize output channels	Compute multiple feature maps at the same time	Faster Conv1/Conv2	More BRAM read ports / DSP
Unroll FC layers partially	Compute several FC products in parallel	Faster FC1/FC2	More DSP
Pipeline layer execution	Overlap read, MAC, write stages	Better throughput	More control complexity
Batch multiple pixels per write	Pack 4 uint8 pixels into one 32-bit word	Faster HPS input transfer	More unpacking logic

Speed - Max Frequency



- CNN FSM must generate BRAM addresses every cycle.
- Direct division/modulo for kernel decoding can create long combinational logic.
- This path can limit Fmax because BRAM address must be ready before the next clock edge.

- Implemented with range comparisons instead of division
- Reduces combinational complexity
- Helps optimize the BRAM address-generation path

End-to-End Data Flow

Software VS. hardware

HTML canvas
PC
base64 string
(JSON)
Browser side
Flask (app.py)
hps-client.py (LZ)
USE_HPS_INFER=1
HPS
hps_infer_server.c
listen port 9090
Software ref (no quantize)
accept a tcp from py client
parse_pixel_values()
write into FPGA
① mmap FPGA reg
Linux chose the virtual adress
map size
② write 1024 pixels into FPGA pixel reg window
HPS hardware converts this into an AXI write transaction
/dev/mem + mmap()
AXI signal
hps_lenet_regs.sv
1. Recieve write addr, data (both)
2. Judge if the addr is pixel window
prediction
confidence → FPGA send raw logits. Flask use softmax-like method change it to percentage
matches → Top 3 scores
hps_result.get("debug")
What is AXI doing in one inference
front end
app.js
app.py
get response: chunk = sock.recv(16384)
hps_client.py | pars JSON
JSON
response → TCP
write_response
TCP socket handle to py client
server.c
read result
read logits
result_word
high 16 bits: cycle
low 4 bits: predicted result
HPS call read_fpga_outputs() to get results
regs_flat[(word_index-2)*32+:32]
Reg map
logits
soc_system_top.sv
connection of instantiation of lenet_int8_top.v
CNN core: lenet_int8_top.sv
image BRAM instantiation
write the last pixel, pull up input_loaded
server.c server write control reg
clean old "done" signal
begin a new inference
server.c read status reg after writing "start" signal
hps_lenet_regs.sv seen back status of FSM
done_latched
FSM get "start" signal, initialize to "S_IDLE" state
soc_system_top.sv
FC3: 10 logits
ST_ARGMAX
done
READ — from input BRAM / fm BRAM / ROM
multiply — MAC (x3 in parallel)
ADD — accumulate partial sum
write — add bias, quantize, write to BRAM