Simple GPGPU

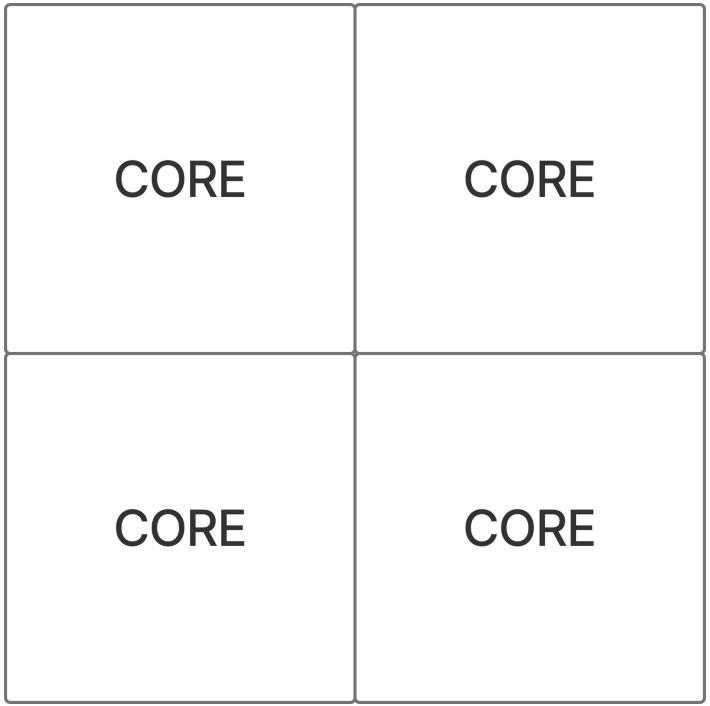


ECE 338 - Parallel Systems Architecture

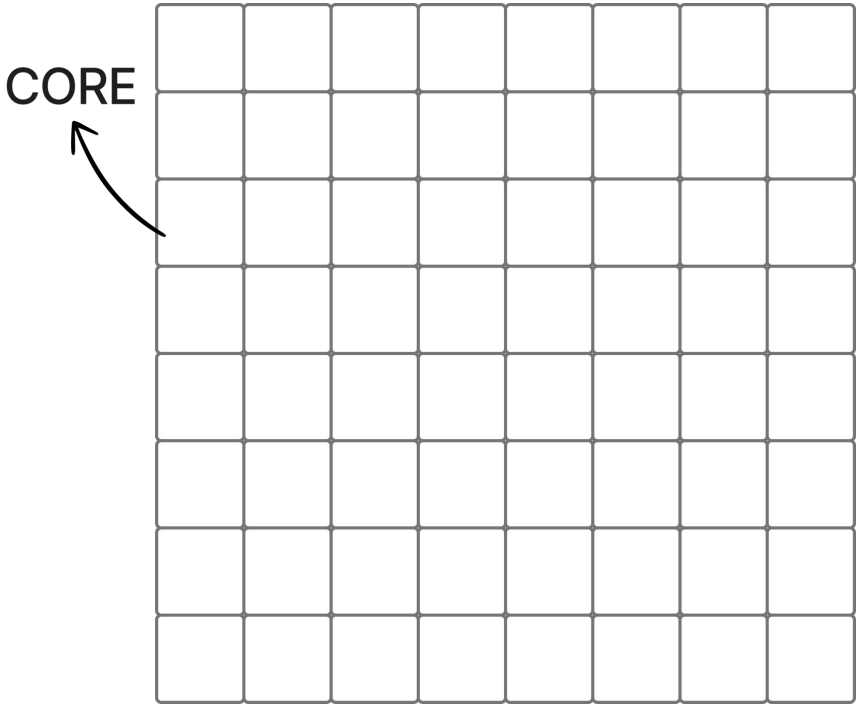
Tsiantos Dimitrios | Tsogkas Panagiotis Nikolaos | Nikas Ioannis Iason

CPU vs GPU

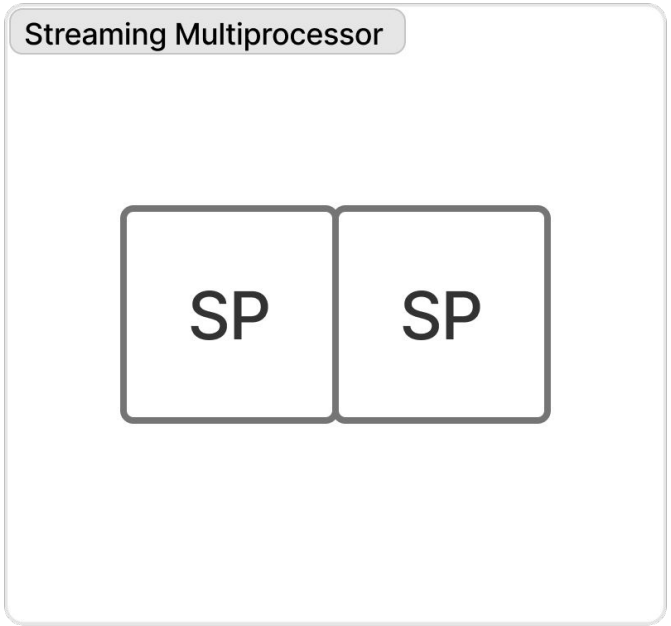
Few complex cores



Many simple cores

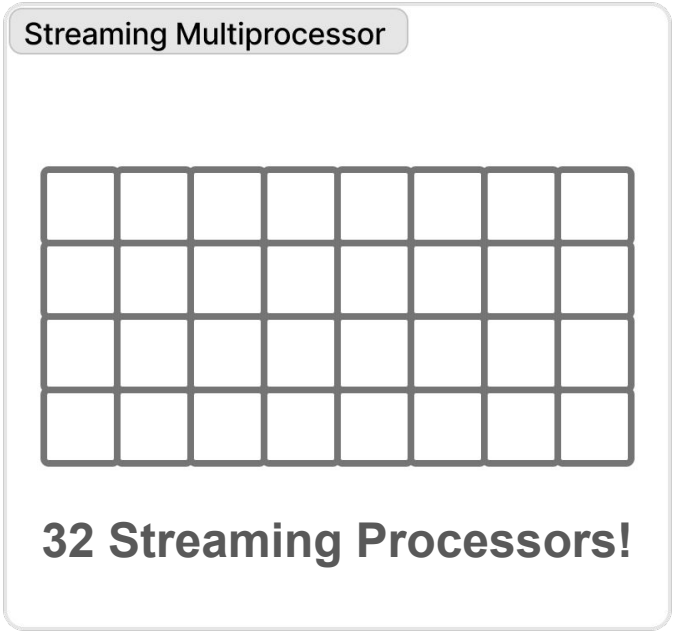


Initial Idea



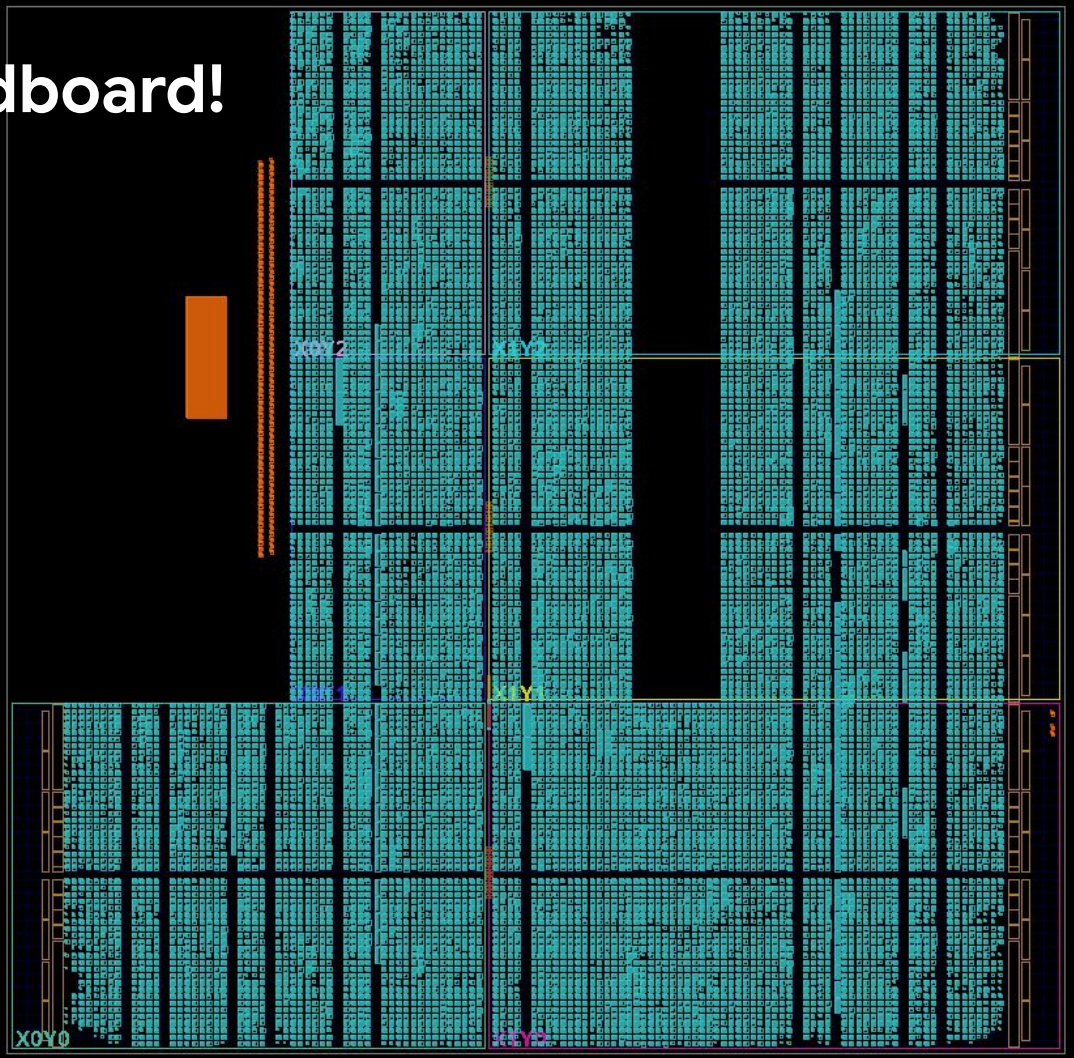
ADD	Rd = Rs + Rt
SUB	Rd = Rs - Rt
MUL	Rd = Rs * Rt
DIV	Rd = Rs / Rt
CMP	NZP = sign(Rs - Rt)
LOAD	Rd = Mem[Rs]
STORE	Mem[Rd] = Rs
BRANCH	...

Implementation

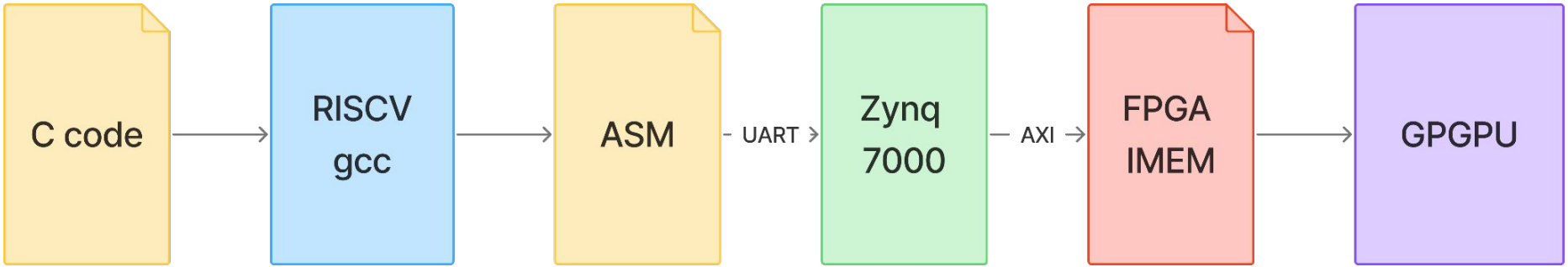


R-type	`add`, `sub`, `mul`, `and`, `or`, `xor`, `sll`, `srl`, `sra`, `slt`, `sltu`
I-type ALU	`addi`, `andi`, `ori`, `xori`, `slli`, `srli`, `srai`, `slti`, `sltiu`
I-type Load	`lw`
I-type Jump	`jalr`
S-type	`sw`
B-type	`beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu`
U-type	`lui`
J-type	`jal`
Special	`nop`

Runs on Zedboard!



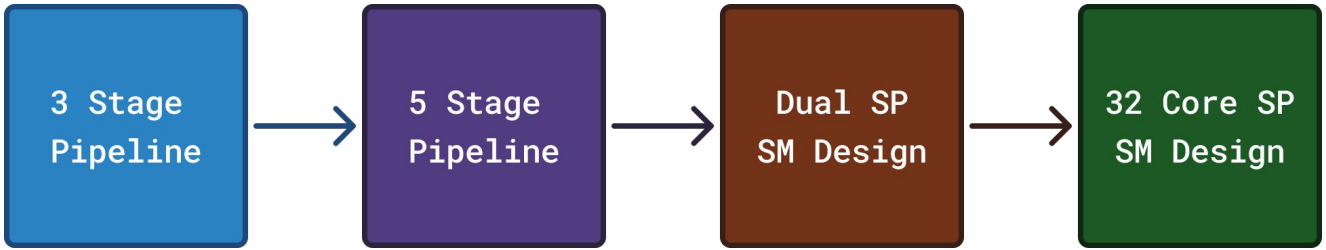
Execution Flow



Hardware Design

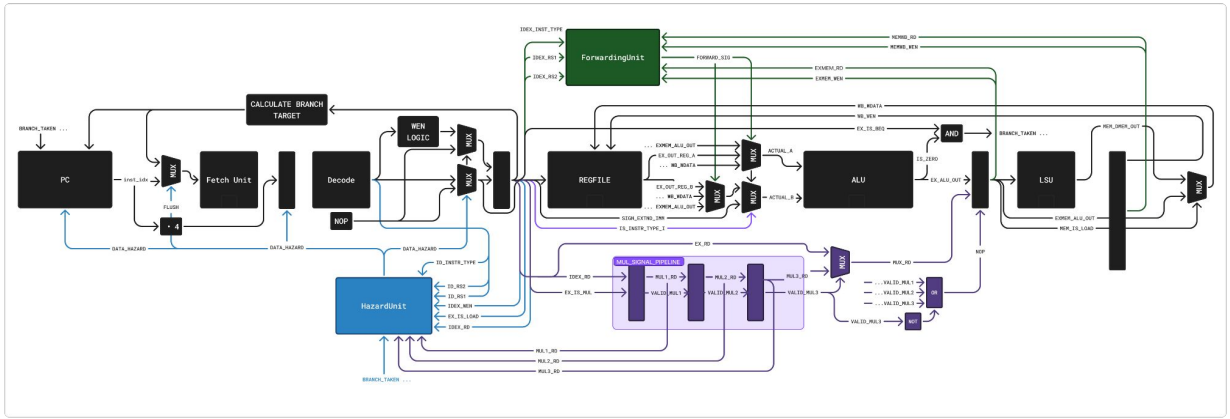
Architecture Evolution & Timeline

- **3-Stage Pipeline:** Initial scalar design, validated core **RISC-V** decoding & execution
- **5-Stage Pipeline:** Increased throughput, introduced **Hazard/Forwarding** units for data dependencies
- **Dual-Core SM:** First parallel prototype, implemented lockstep execution and shared instruction fetching
- **32-Core SM:** Final **SIMD** architecture, scaled to 32 parallel processing elements for high compute density



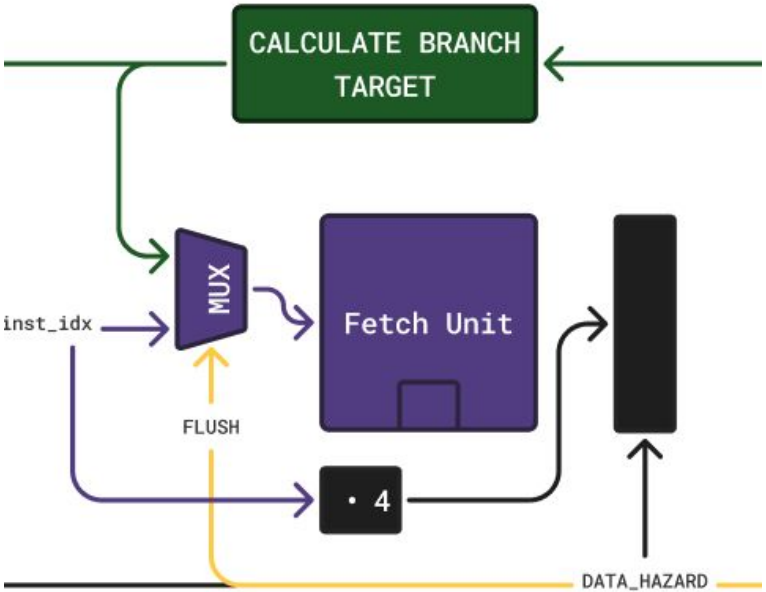
Implementation of Core Features

- **Control Flow:** Hardware support for conditional branching and unconditional jumps
- **Data Hazards & Forwarding:** Dedicated units to detect dependencies, stall safely, and forward data to minimize performance penalties
- **Multiplier (MUL) Hardware:** A custom 3-stage pipelined multiplication path designed to simulate multi-cycle DSP blocks



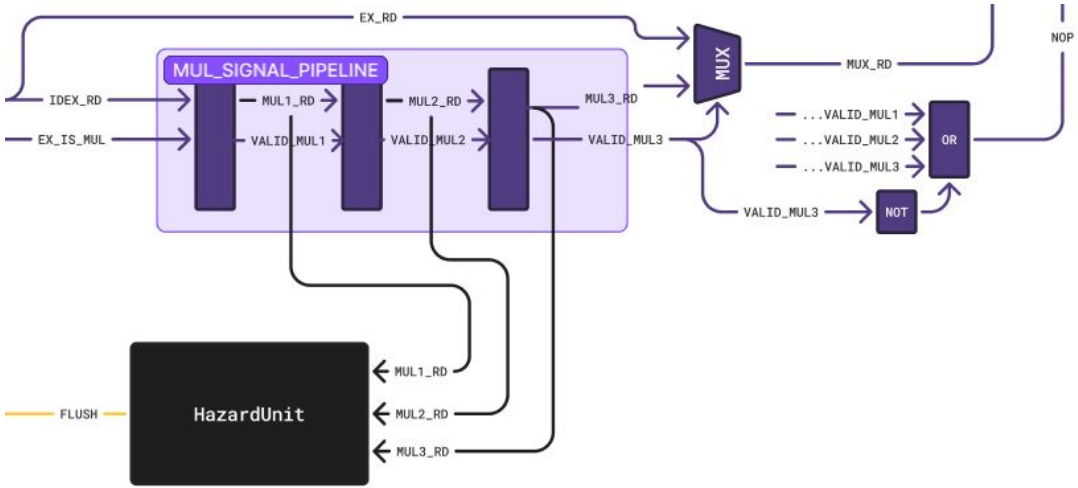
Control Flow (Branching)

- **Execution Stage Resolution:** Branch targets and conditions are evaluated in the Execute (EX) stage
- **Flush Cycles:** The target PC is forwarded to the Fetch Unit, achieving a single-cycle flush
- **Supported Instructions:** Handles conditional branches (**BEQ, BNE, BLT, BGE, BLTU, BGEU**) and jumps (**JAL, JALR**)



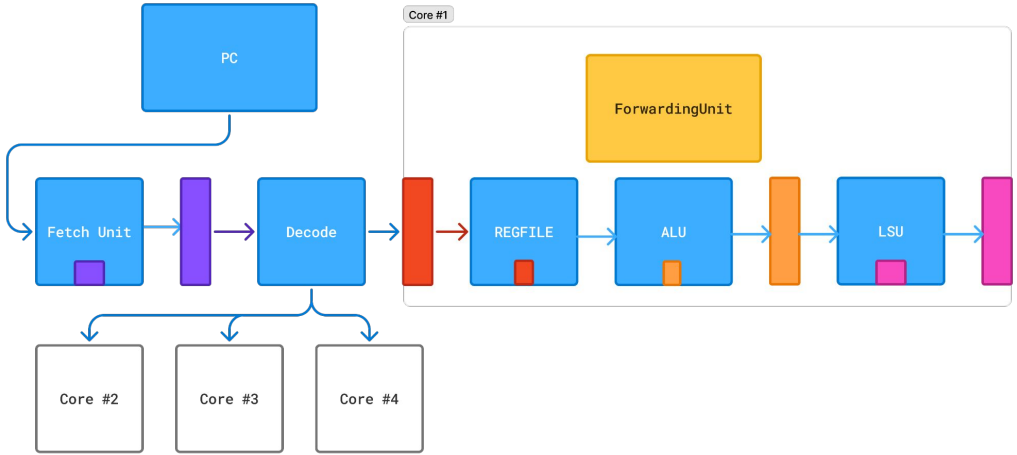
Data Hazards & Forwarding

- **Architecture:** Utilizes a conventional, **MIPS-style** approach to dependency resolution across the 5-stage pipeline.
- **Caveats (The MUL Pipeline):** Custom logic was required to handle complex hazard cases stemming from the multi-cycle DSP/Multiplication pipeline.



Transition to SIMD

- **Resource Optimization:** Copying the full 5-stage pipeline 32 times would waste immense resources fetching and decoding the same instruction
- **The Shared Front-End:** The **StreamingMultiprocessor (SM)** centrally houses the **IF** and **ID** stages
- **The Compute Unit Back-End:** The original SP was converted into a pure compute lane



SIMD Control Flow & Lockstep Execution

- **Lockstep Execution:** A single shared PC forces all 32 cores to execute the exact same instruction simultaneously.
- **Master Core Logic:** A single Core acts as the reference for control flow, its branch calculations drive the shared Program Counter.
- **No Hardware Divergence:** Cores cannot independently take different branch paths (resolved later in the software stack).

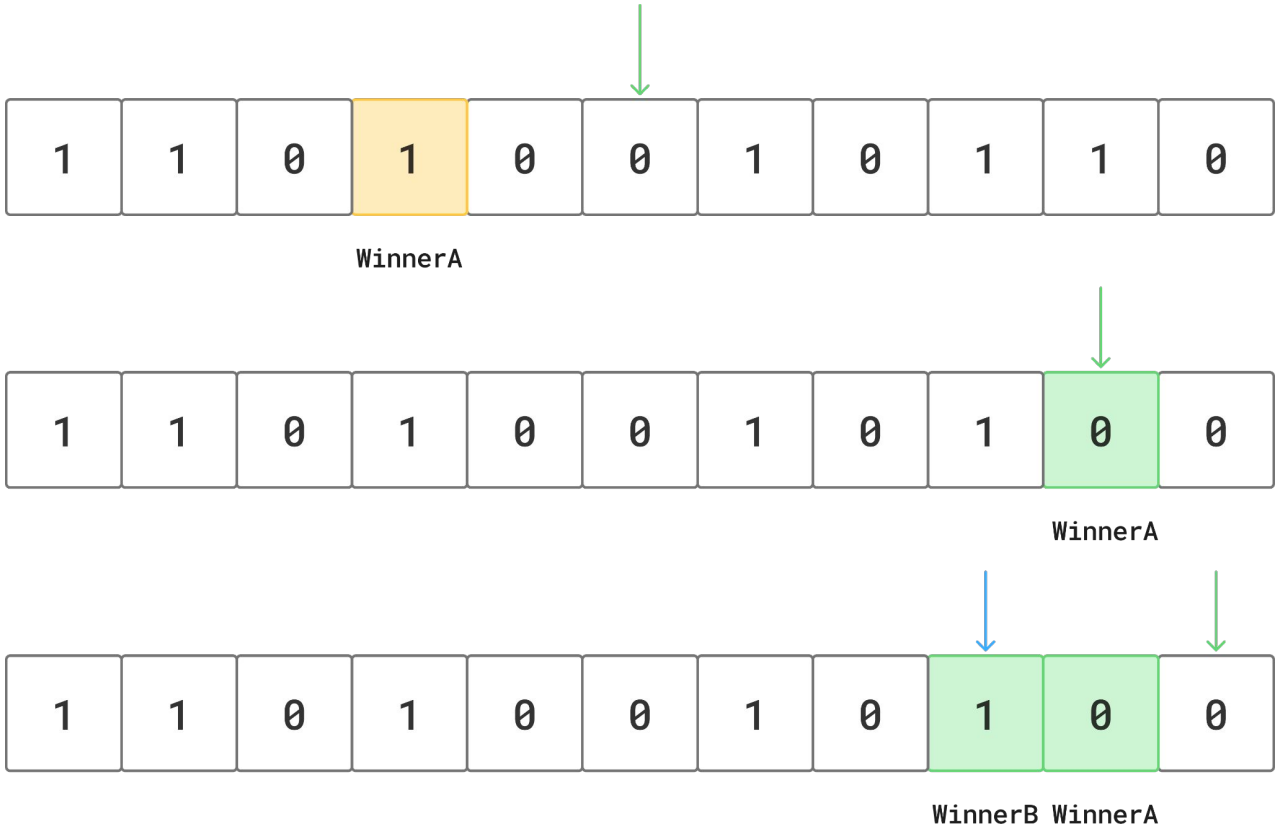
The Memory Crossbar

- **Realizing the Bottleneck:** 32 cores sharing a single memory bus creates a massive data bottleneck. (Instruction Memory is unaffected due to lockstep execution)
- **Dual Port BRAM Solution:** An Nx2 crossbar routes requests from 32 cores to a Dual-Port Block RAM, serving **two cores per cycle**
- **Bank System Solution (Future Work):** A multi-banked memory system was written to further increase ports, but was excluded from the final bitstream due to time constraints
- **Global Stall Logic:** The hardware actively accumulates crossbar **grant** signals to safely freeze the pipeline during memory contention

Crossbar Arbitration Design

- **Priority Encoding (Winner A):** The crossbar scans the 32-bit request bus from **MSB** to **LSB**. By continuously overwriting the winner index, the lowest active core **ID** ultimately wins **Port A**
- **Masking & Winner B:** The bit for **Winner A** is masked out, and the exact same scan is applied to the remaining requests to find **Port B**.
- **Multiplexing to BRAM:** The two winning indices are used as selector signals to mux their respective addresses, write-enables, and data directly into the **Dual-Port BRAM**
- **1-Cycle Delayed Routing:** Because **BRAM** reads take one clock cycle, the crossbar saves the winning core **IDs** in registers to correctly route the returning data back to them on the next cycle, and sets the request to **granted**

Crossbar Arbitration Design



Atomic Memory Operations (AMO)

- **Synchronization :** Developed alongside a multi-banked crossbar design, the AMO instruction enabled hardware level parallel synchronization.
- **Atomicity:** Bypasses the standard core pipeline to guarantee the sequence executes without data corruption or interruption from other cores.
- **Implementation:**
  - **Cycle 1 (Read & Lock): The arbiter grants access, issues a read request, locks the targeted memory bank, and latches the core's target address and addend.**
  - **Cycle 2 (Modify & Write): The crossbar hijacks the locked bank, performs the addition locally (returned data + addend), and writes the new value back to memory.**

Hardware Verification Strategy

- **Constrained-Random Verification (CRV)**: Pre-execution software validation to guarantee lockstep safety (constraints to eliminate divergence or memory corruption)
- **Random Instruction Testing (RIT):** Aggressive hardware fuzzing designed to intentionally break pipeline dependencies and stress the memory crossbar
- **The Automated Testsuite:** A custom ecosystem that compares physical Verilog simulation results against a Python-based "Golden Model" to ensure correctness

Constrained-Random Verification (CRV)

- **The Goal:** Ensure multi-core programs execute safely in lockstep without undefined hardware behavior
- **Infinite Loop Prevention:** Enforces a strict maximum cycle count (10,000 cycles) to catch stalled or infinite loops, and prevents backwards jumps
- **Divergence Checking:** Simulates branch decisions across all cores; fails immediately if cores attempt to take different paths
- **Memory Corruption Prevention:** Scans memory write addresses concurrently to ensure no two cores attempt to write to the exact same memory location in the same cycle

Random Instruction Testing (RIT)

- **Automated Fuzzing:** The **random_tester.py script** generates highly-biased **RISC-V** assembly programs to aggressively stress the hardware
- **Forced Dependencies:** Biases register selection (50% chance) to constantly force the pipeline's Forwarding and Hazard units to activate
- **"Evil Sequences":** Purposefully injects edge-case sequences (e.g., Load-Use dependencies, Store-Load bypasses, Multi-cycle DSP traps)

The Testsuite Ecosystem

run_tests.sh

Handles simulation compilation and orchestrates the execution of the entire test suite.

assembler.py

Parses custom RISC-V assembly files and translates them into machine code (.mem files) for the BRAM.

expected_generator.py

The 'Golden Model'. A Python software simulator predicting exact cycle-by-cycle states to verify hardware correctness.

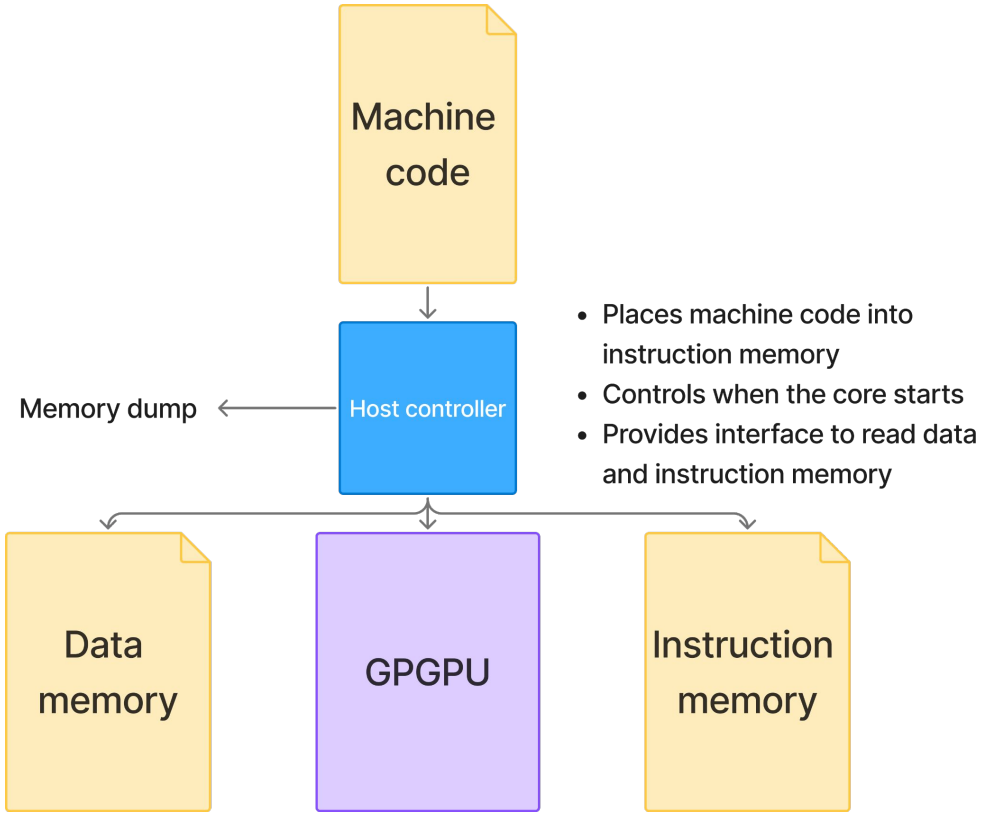
program_checker.py / random_tester.py

Program safety validator and aggressive hardware fuzzer for robust verification.

SoC & Integration

System-on-Chip

- At this stage, the core was functionally correct and verified at the RTL simulation level.
- The next step was to package it in a **SoC** that can be synthesized and deployed on the FPGA, while also allowing for **communication**.



Host Controller & Command Processor

State	Trigger	Next state
`LOADING`	Run command received	`RUNNING`
`RUNNING`	Kernel finishes	`DUMPING`
`DUMPING`	Done command received	`LOADING`

- Responsible for managing the **state** of the core
- Waits for the **command processor** to receive a command and changes the state accordingly.
- The core has 3 states.

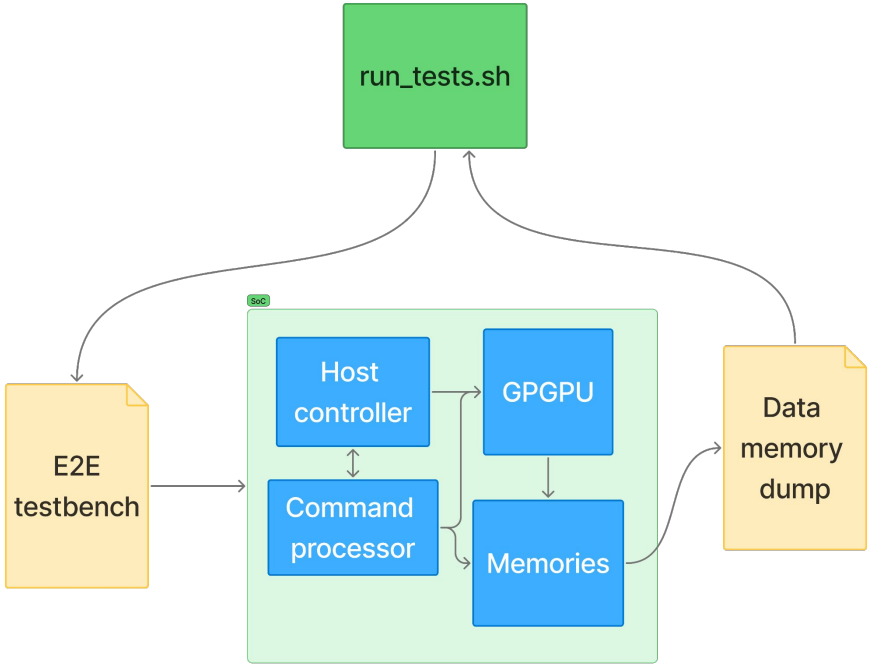
Host Command Processor

The command processor is responsible for:

- Receiving commands
- Executing the appropriate action
- Notifying the host controller if needed, based on the command.

Command	Action
`WRITE_IMEM`	Write to instruction memory
`READ_IMEM`	Read from instruction memory
`WRITE_DMEM`	Write to data memory
`READ_DMEM`	Read from data memory
`RUN`	Done writing / Start running
`DONE`	Done reading / Go to loading

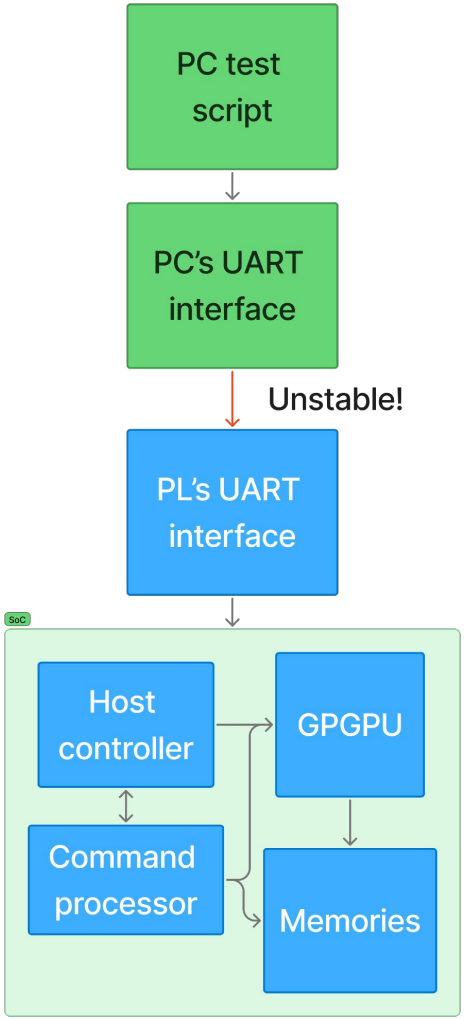
End-to-End Testing



- Existing testing suite validated modules at the **RTL level**
- It was now extended to work with the new top-level module
- Tests still work **after implementation** because they target the actual system interface.
- Provides a more **better guarantee** for correctness since it's closer to the FPGA execution

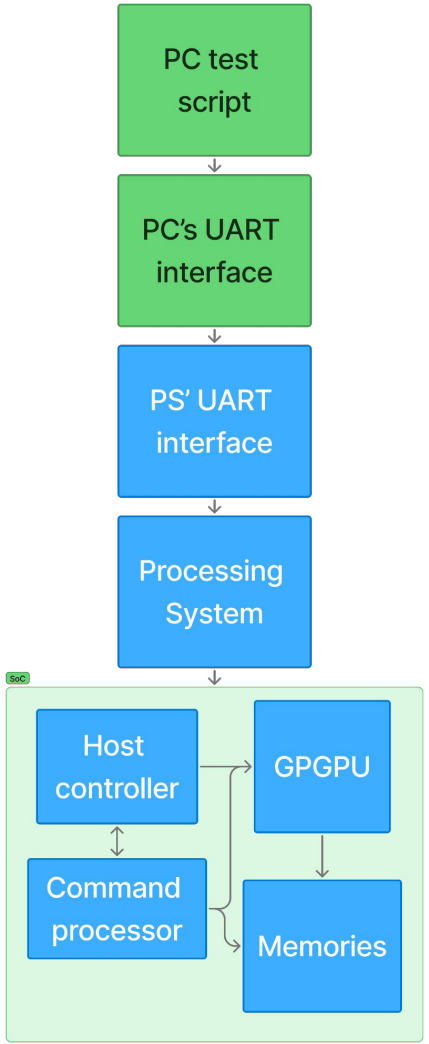
Integration - UART Attempt

- The first approach was to communicate **directly** with the Programmable Logic using a UART module.
- While simple, it proved **unstable** and made it difficult to debug reliably on the board.
- Because we needed communication with the system to be reliable we quickly **moved away** from this idea.



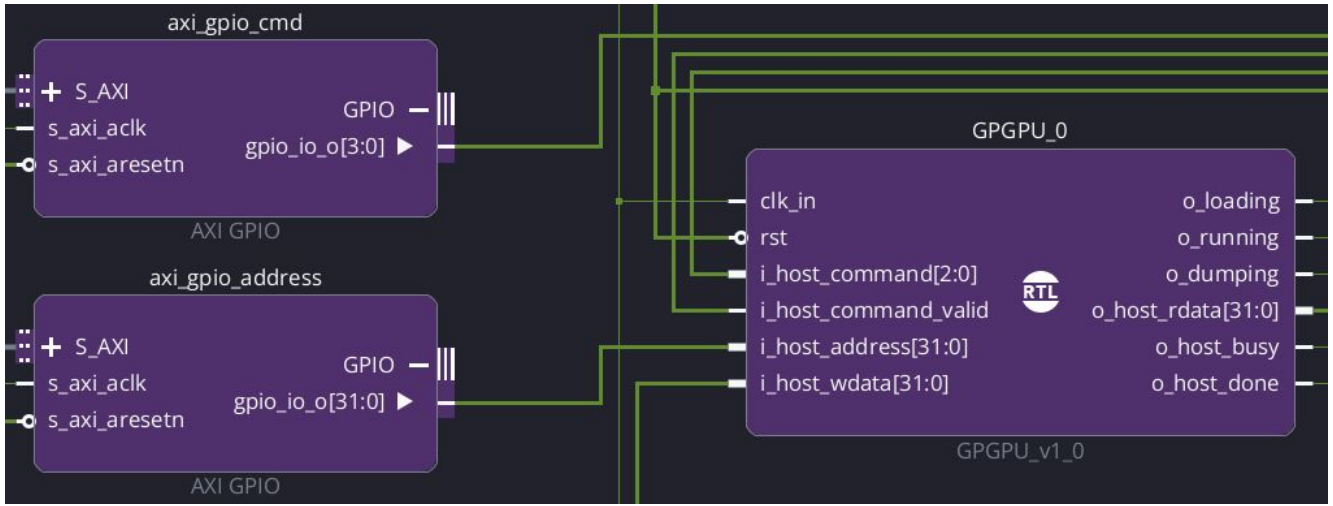
Integration - AMD Zynq™ 7000

- We moved the the communication from the PL to the Zynq Processing System.
- The PS handles UART communication from its own interface, which is much more stable.
- The core now receives commands from the PS

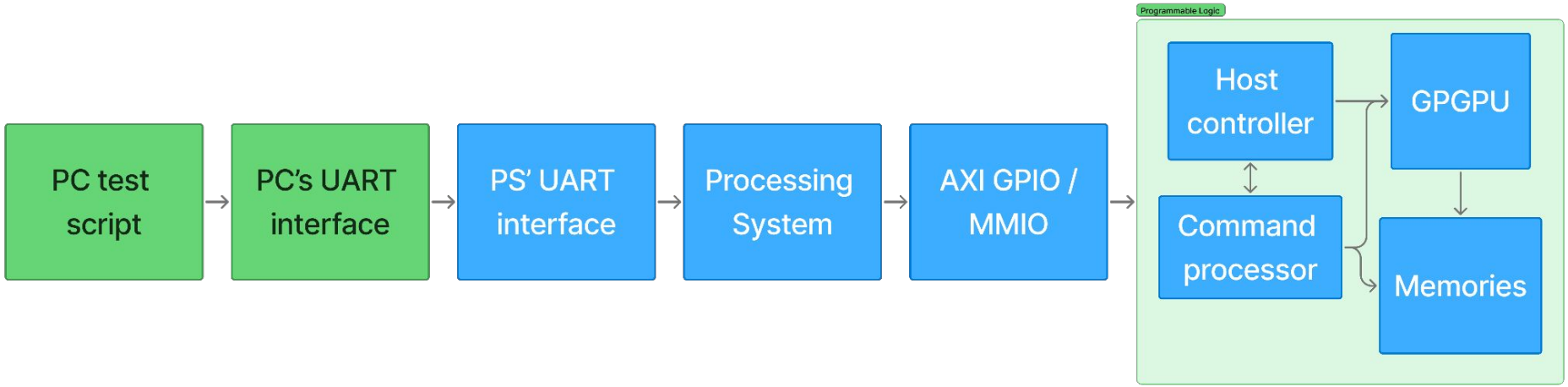


Integration - AXI General Purpose I/O

- To connect the PS with the PL we used **AXI GPIO** blocks.
- AXI GPIO exposes simple **memory-mapped** registers to the PS software.
- PS **writes** commands to those registers and **reads** data back from them.
- On the PL side, those signals are directly connected to the **host controller**.



Integration - Simplified Block Diagram



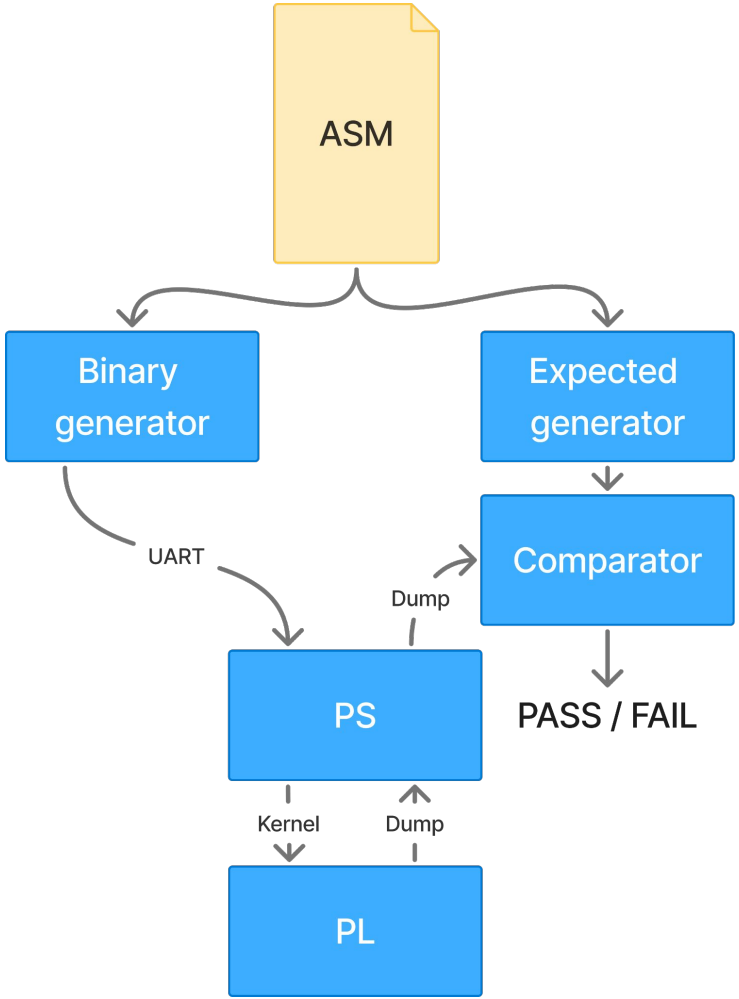
Integration - Host Software

Host command	Action
`load[i|d]mem`	Wait to receive data from UART and write them to data or instruction memory
`dump[i|d]mem`	Read data from instruction or data memory and send them over to UART
`status`	View the current status of the core
`run`	Used only when core is in `LOADING` state to start the core
`done`	Used only when core is in `DUMPING` state to go back to `LOADING`

- A lightweight host program runs on the ARM processor in the Zynq PS.
- It receives **high-level commands** from the external test script over UART.
- For each command, it performs the required AXI GPIO reads/writes to control the PL subsystem.

End-to-End Testing

- The testing suite is **extended once more** to make the existing expected-output based tests to use the actual FPGA board.
- The test script now sends commands **over UART** to the PS software.
- PS forwards the operation to the PL.
- After the execution the test script requests a **memory dump** and checks for correctness, based on the expected output.



Software Stack & Runtime

Compiler

Write assembly by hand? Doesn't scale

Solution: use RISCV GNU cross-compiler

But what about:

- Data/Instruction memory?
- Main function?
- Stack?
- Kernel arguments?
- Thread index?

Linker Script

An .ld script provided to the compiler.

Instruction & Data Memories:

```
MEMORY
{
    IMEM (rx)  : ORIGIN = 0x00000000, LENGTH = 8K
    DMEM (rw)  : ORIGIN = 0x00000000, LENGTH = 8K
}
```

Linker Script

An .ld script provided to the compiler.

Address spaces:

```
SECTIONS
{
    . = ORIGIN(IMEM);
    .text :
    {
        ...
    } > IMEM
    .rodata ALIGN(4) :
    {
        ...
    } > DMEM
...
```

Linker Script

An .ld script provided to the compiler.

Symbols:

```
__gpu_args_base = 0x00000040;
__gpu_data_base = 0x00001000;
__gpu_stack_bottom = 0x00001800;
__stack_top = 0x00002000;
```

Stack

Compiler was made for 1 core, we have 32.

All SPs will try to access the same address on the stack:

```
addi sp,sp,-32
sw x8,28(sp)
sw x9,24(sp)
sw x18,20(sp)
sw x19,16(sp)
sw x20,12(sp)
sw x21,8(sp)
sw x22,4(sp)
```

Stack

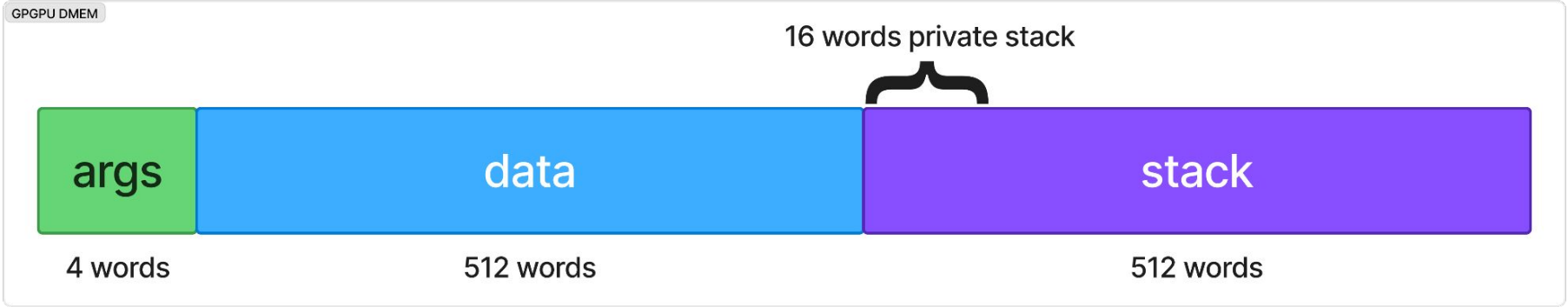
```
#define GPGPU_START(kernel_fn)
__attribute__((naked, noreturn, section(".text.start"))) void _start(void) {
    __asm__ volatile(
        "mv x5, x31\n"
        "slli x6, x5, 6\n"
        "lui sp, %hi(__stack_top)\n"
        "addi sp, sp, %lo(__stack_top)\n"
        "sub sp, sp, x6\n"
        "jal x1, " #kernel_fn "\n"
        "1:\n"
        "jal x0, 1b\n"
    );
    __builtin_unreachable();
}
```

offset sp by **16 words** (64B)

tid	Stack Base
0	0x2000
1	0x2000 - 64 = 0x1fc0
2	0x2000 - 128 = 0x1f80
…	…

Memory

Name / Symbol	Base Address	End Address	Size Bytes	Size Words
__gpu_args_base → __gpu_args_end	0x00000040	0x00000050	16 B	4 words
__gpu_data_base → __gpu_data_limit	0x00001000	0x00001800	2048 B	512 words
__gpu_stack_bottom → __stack_top	0x00001800	0x00002000	2048 B	512 words



Thread ID

```
static inline __attribute__((always_inline))
unsigned int gpgpu_thread_id(void)
{
    unsigned int tid;
    __asm__ volatile("mv %0, x31" : "=r"(tid));
    return tid;
}
```

Keep value of x31 to **tid variable**

C Code

```
void kernel_main(void) {
    unsigned int tid = gpgpu_thread_id();

    volatile int *base = (volatile int *)(uintptr_t)GPGPU_ARGS[0];
    volatile int *data = base;
    volatile int *diff = base + (CORES + 1);

    // Calculate differences
    diff[tid] = data[tid + 1] - data[tid];

    return;
}
GPGPU_START(kernel_main)
```

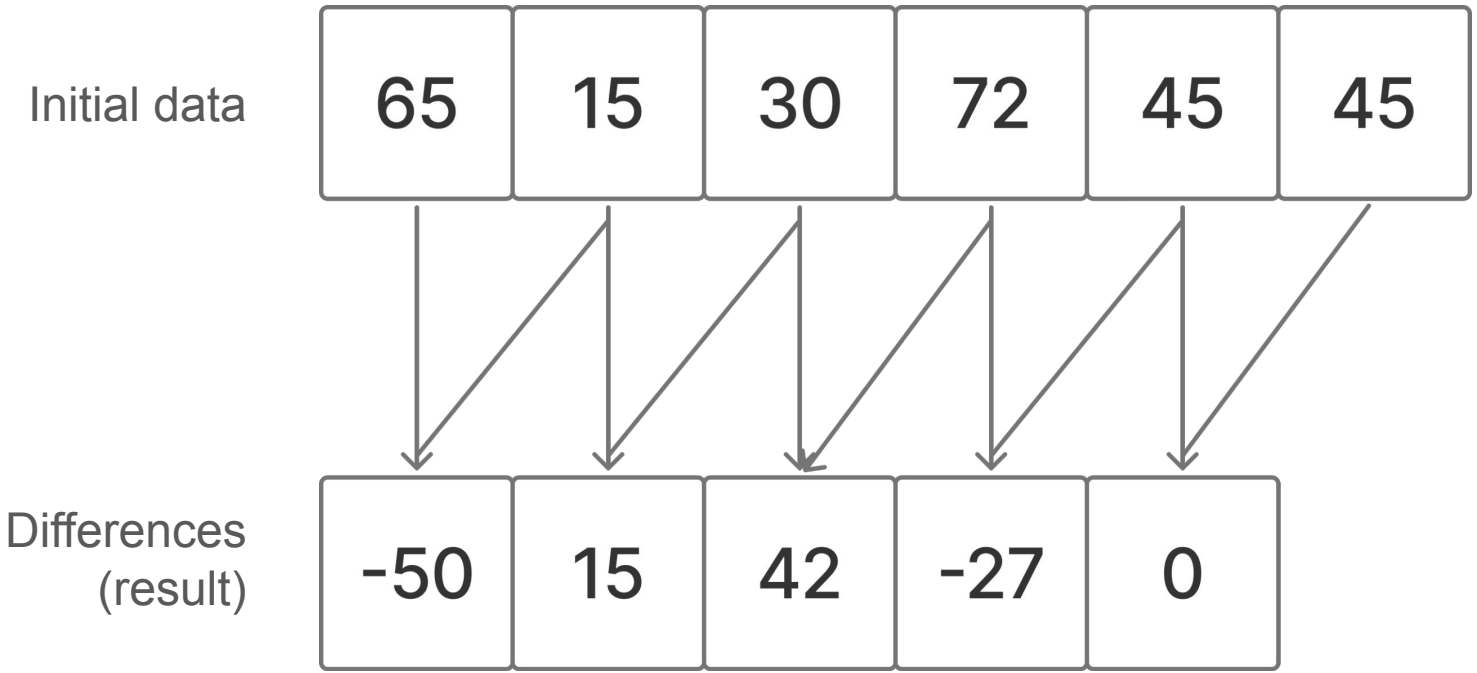
get the thread index

write array to given address

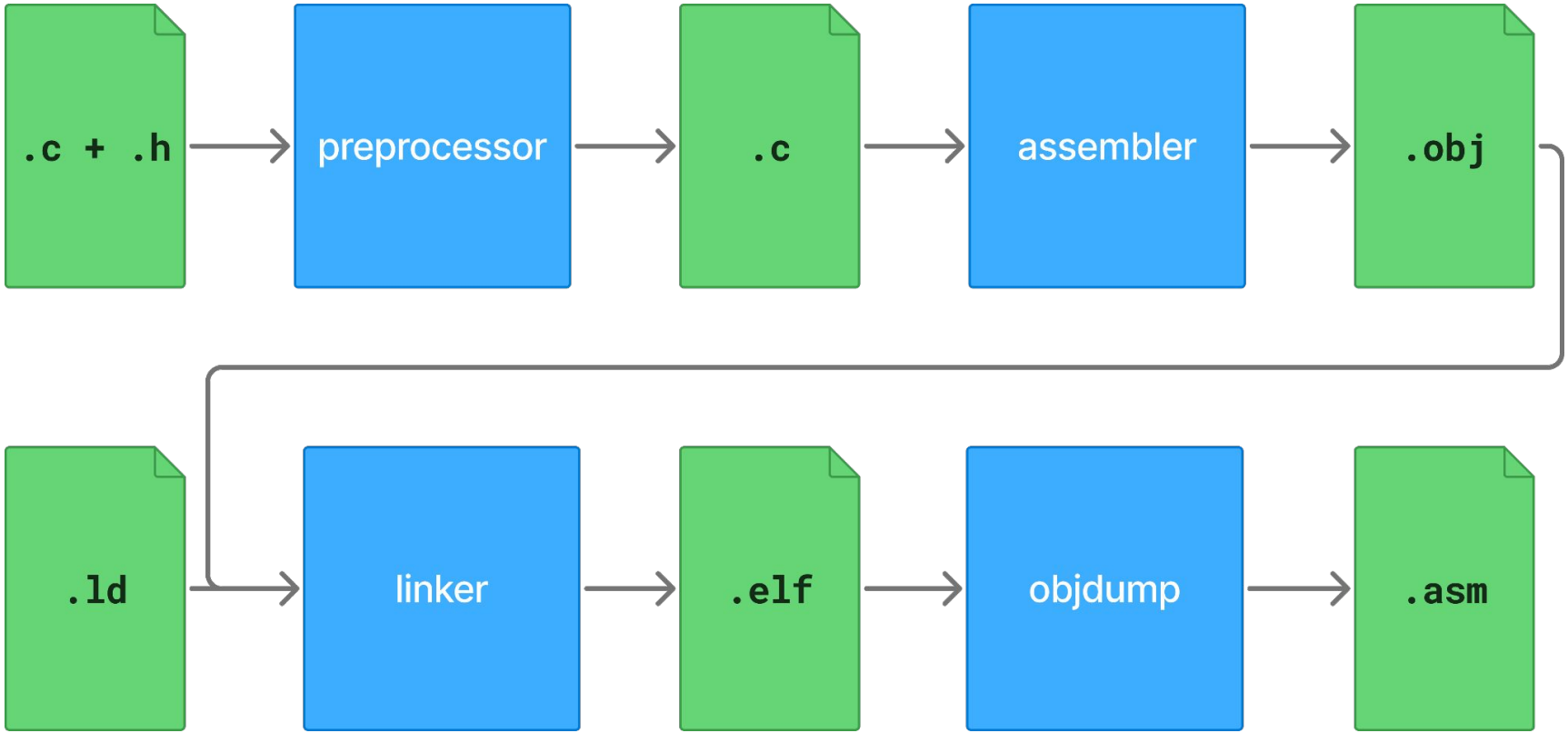
call kernel



C code - Adjacent differences



Workflow



Results

Results - Performance

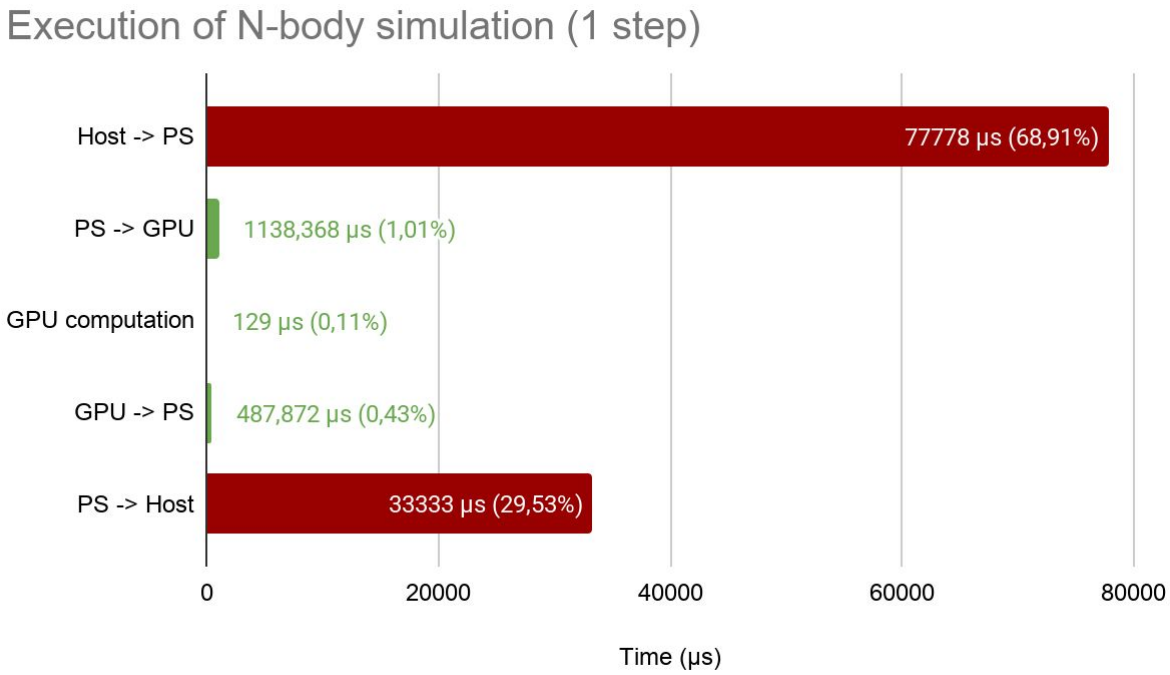
$$Performance_{peak} = \#\text{SPs} \times \text{IPC} \times f = 32 \times 1 \times 38\texttt{MHz} \approx 1.23 \texttt{ GINTOPS}$$

N-body simulation comparison		
	GPGPU	**Arm Cortex-A9**
Clock frequency	~38.5MHz	~667MHz
Bodies	32	32
Execution time	~129µs	~308µs
Speed up	**~2.38x**	1x

Results - Memory Bandwidth

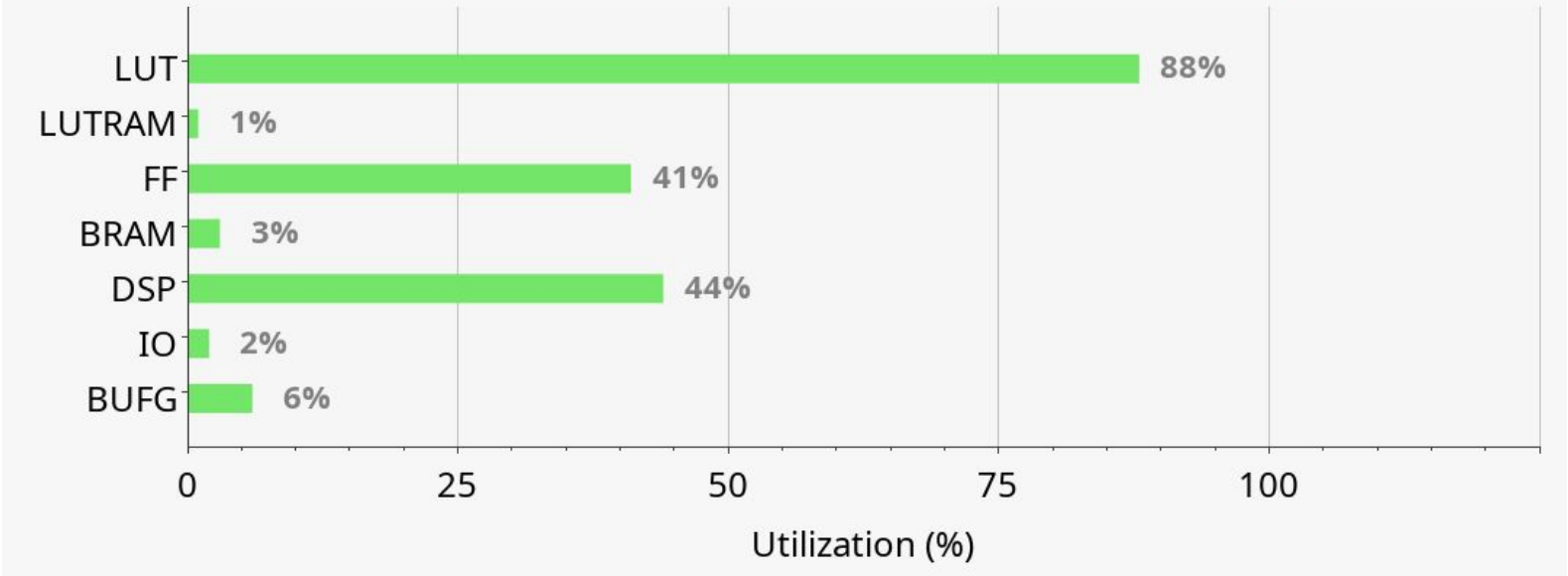
GPGPU memory bandwidth	
Latency	1 cycle
Data width	32 bits
Throughput	2 ports * 4 bytes * 38MHz = 307MB/s

Results - Bottlenecks



- The main **bottleneck** are the transfers **from** and **to** the Zynq-7000 processing system.
- These transfers are over **UART** using a 115200 baud rate or effectively a **11.52KB/s bandwidth**.

Results - Utilization



Future Work

- Floating-point & division
- Warp scheduling
- Warp divergence
- More streaming multiprocessors
- Dynamic memory allocation
- Package the design into and AXI-Lite peripheral
- Integration with a real OS for better communication
- Use larger FPGA that supports PCIe

Questions?



ECE 338 - Parallel Systems Architecture

Tsiantos Dimitrios | Tsogkas Panagiotis Nikolaos | Nikas Ioannis Iason