ARMv8 AArch64 Emulator Checkpoint Report

Team 54

June 6, 2025

1 Group Organisation and Workflow

Our group used a structured workflow and leveraged Git with dedicated `feat/` branches and frequent stub implementations to maintain code quality and compilation. Commit messages followed an imperative format with contributor shortcodes. Every new feature created a GitLab merge request which was reviewed and approved by a separate team member to uphold coding standards.

1.1 Division of Labour for Emulator (Part I)

The emulator development was systematically divided into four core components, ensuring balanced responsibilities across the team:

- **Jai – Core System & I/O:**
  Established the emulator's core infrastructure by defining `ARMState` (registers, PC, PSTATE, memory), initialising its state, implementing the binary file loader, and managing the main execution loop, including command-line parsing, halt instruction, and final output.
- **Richard – Instruction Decoding & Bitwise Utilities:**
  Led instruction decoding by developing bit extraction utilities, defining C structures (`instruction_types.h`) for decoded instructions, implementing `decode_instruction`, and creating helpers for `PSTATE` flag updates and register ID mapping.
- **Prasanna – Data Processing Instructions Execution:**
  Implemented the execution logic for all data processing instructions (arithmetic, wide moves, logical, multiplies), developed underlying bitwise shift functions, and integrated `PSTATE` flag updates.
- **Zayan – Memory & Control Flow Instructions Execution:**
  Implemented memory access and control flow instructions, including single data transfers (`ldr`, `str`) with all addressing modes and load/store size handling, and developed all branch instructions for PC modification, offset calculations, and conditional execution.

2 Group Progress and Communication

Our group is working well together thus far, showing strong individual performance as well as good mutual support. We have elected Jai as our group leader, to manage the workflow and decide how the project will be split between members of the group. Initially, we split the emulator task into 4 roughly equal components and developed group standards for git commit messages, branch names, and comment styles. Communicating through a Slack channel, we organise frequent meetings (approximately every other day) to ensure that each team member is continuing to stay on track with their assigned work. A key practice for us has been to pair up for merge commits so that we can peer review eachother's code for quality and catch any potential issues.

For later parts of the project, we can see our current communication standards continuing majorly as they are. We feel that in future tasks (the assembler and extension) there will be more interdependencies on eachother's code, so integration tasks will pose a greater challenge, yet we feel confident that our established communication standards and peer review process should suffice.

3 Emulator Structure and Reusability

Emulator Architecture and Reusability for Assembler

Our emulator is structured around a standard Fetch-Decode-Execute cycle, operating on a shared `ARMState` struct that represents the machine's current state. This struct includes an array of 64-bit general-purpose registers (`registers[31]`), the program counter (`pc`), a `PSTATE` struct holding the NZCV flags as bools, and a 2MB byte-addressable memory array.

The fetch phase reads 32-bit A64 instructions from memory using the current PC. The decode phase identifies the instruction category using bits 25–28 and extracts relevant fields using bitwise operations, with the help of reusable utility functions (e.g., `get_bits()`). A decoded instruction is represented by a `DecodedInstruction` struct which includes the instruction type as an enum `type`, a 32-bit `raw_instruction` and members for the various instruction-specific fields (e.g., `dp_imm_imm12`). The execute phase performs the instruction logic using helper modules for arithmetic, memory access, or branching, updating the `ARMState` accordingly and incrementing the PC.

Several components of the emulator will be reusable in the assembler, especially the instruction decoding logic and its bit extraction and field parsing logic, which can be reverse engineered to help us generate binary encodings from assembly syntax. Additionally, functions that map register names to IDs and handle 32-bit versus 64-bit register size will be useful when parsing operands.

4 Challenges and Mitigation

Building the emulator presented both technical and collaborative hurdles. Technically, translating ARMv8 AArch64's instruction set into C demanded close attention to detail, especially with bitwise operations. Getting complex instruction behaviors right, like 'ASR' and 'ROR' shifts - with their nuances of signed bit propagation and 32/64-bit operand widths—or accurately calculating `PSTATE` flags (`N`, `Z`, `C`, `V`), was particularly challenging. Our approach involved breaking down these complexities into smaller functions, extensive manual cross-referencing with the ARM Architectural Reference Manual, and iterative refinement.

From a collaborative standpoint, integrating code from four parallel development streams could have led to significant dependencies and merge conflicts. We mitigated this by establishing clear interfaces in shared header files (`arm_state.h`, `instruction_types.h`) early on, which proved essential for minimizing integration friction. Our disciplined Git workflow, emphasizing frequent 'git pull –rebase' operations, helped maintain a linear history and manage divergences. When push failures indicated remote history conflicts, we quickly learned to diagnose these and employ 'git push –force-with-lease' after a successful rebase.