Report for PA2

by Yuanfang Xiang (221300012)

Part I: RTFSC

Implementation

Implemented immediate number parsing and decoding of `J-type` and `B-type` instructions for `nemu`. Implemented all required instructions for all `cpu-test` programs, according to `RISCV` manual.

Exercise 1.

the process of a single instruction executed in `NEMU`:
1. In `exce_once` of `cpu`, call `isa_exce`.
2. Call `ifetch`, fetch the 4 byte instruction bit string from memory with `paddr_read` function. Then update `snpc`. 3. In `decode_exec`, initialize 2 source op-numbers and imm. Then parse instruction string with `INSPAT`. Update `dnpc` with `snpc`.
4. In `INSPAT`, identify instruction type and, call macro `BITS` to parse reg `rs1`, `rs2` and `rd` from instruction string. Then parse `imm` with `BITS` and `SEXT` according to current instruction type.
5. Call `decode_oprand` with macro define. Then run corresponding `C` code to execute operation.
6. Update `cpu.pc` (current pc) with `dnpc`.

Part II: Kernel Library Functions & Trace

Implemented more `nemu` features for debugging, including `instruction trace` (`i-ring-buf`, enables last `N` last executed instructions to be printed), `memory trace` and `function call trace`.

Reading `symtab` and `strtab` sections of `ELF` file to obtain function call records and function name. A stack is maintained to ensure correct function call and return records are recorded. **Bug to be fixed: the stack might overflow, or other bugs which might cause *segment fault*, when function call depth is too large.**

Implemented `klib` functions for `AM`, including memory and string operation (`strcpy()`, `memcpy()`, etc.), format output (`printf()`, `itoa()`, etc.). Similar to `cpu-test`, finally devised several test programs to test the functionality of `klib` under a new directory.

Part III: IO Extension

Imitating the implementation of serial port, completed `AM` and `nemu` features of `clock`, `keyboard` and `VGA`. Using APIs designed in `nemu`, implemented functions to pass I/O data to and from abstract registers of AM and nemu.

AM is finally able to run slide, typing game, demo (excepted life and bf) and snake successfully.

Part IV: Exercises

Exercise 1: how does typing game run

After compilation, link c files and convert code to machine instruction.

On the AM layer, initialize ioe and vga with calling ioe_init() and video_init() in corresponding directory according to selected ARCH.

Then enter the game main loop. Firstly update game logic, generate a new char. Next enter a new loop, wait for keyboard ioe to return a key value being hit, or NONE. When an AM game program calls io_read(AM_INPUT_KEYBOARD), am calls the handler function of corresponding virtual device register, i.e. INPUT_KEYBRD. In input.c, AM uses inw macro to read keyboard related value from mmio mapping address from memory array, put it in abstract register AM_INPUT_KEYBOARD for game to use.

Since the inw macro uses * notation to visit memory, after compilation, compiler will give nemu a memory read instruction. Therefore, nemu will call paddr_read to move the keyboard value to certain register from the mmio address of keyboard, which stored after nemu got it from SDL, in inst.c. This value is then returned by AM's inw, and passed to game.

In render, then, game calls io_write of AM_GPU_FBDRAW, follow a path similar to above (but reversed) to write RGB pixel value to mmio memory, read it in nemu and pass it to SDL lib functions of rendering to show it in the graphic window of SDL.

Exercise 2: Compilation and Link

When one of static or inline notation is removed, nemu can still compile and run test programs on AM. However when both are removed, nemu would fail to compile and trigger multiple definition error.

Reason: Since inst_fetch() is defined in a .h file, including this file in different files will create multiple symbols in elf. As a function defined in the global field, it would be labeled as GLOBAL without static inline notation. Therefore, multiple GLOBAL symbol with same name would cause multiple definition error. With inline or static inline, the implementation of this function would be expanded at where it be called, avoiding calling the instructions from its definition. With static, the symbol will be labeled as LOCAL, which is allowed to exist more than once and will be arbitrarily chosen when being called. This can be demonstrated by using readelf or objdump to read symbols and instructions in the binary file.

Exercise 3: Compilation and Link (2)

1. 36 entities of `dummy` are constructed in `riscv32-nemu-interpreter`. The result can be counted in the symbol table of `nemu`'s elf file, which is in `$NEMU_HOME/build/riscv32-nemu-interpreter`. The exact number can be counted with `grep` and `wc`.

2. Still 36. Because `debug.h` is included in `common.h` at the end of it. Therefore, every file that includes `common.h` has included `debug.h`. During symbol parsing, both them are weak symbols due to they haven't been initialized, so only one of them is chosen to compile arbitrarily, and the number doesn't change.

3. Compilation failed and triggered a `redefinition` error, because both of them are initialized. Hence they are all strong symbols, and strong symbol can be only defined once, else a `redifinition` error would be triggered.

Exercise 4: Read the F********king Makefile

1. In `hello/Makefile`, initialize the source image program to run, i.e. set variable `$NAME = hello`, `$SRCS = hello.c`. Then call `$AM_HOME/Makefile`.

2. In `$AM_HOME/Makefile`, first run a variety of sanity checks and setups. Then get work directory `am-kernels/kernels/hello` and the corresponding destination directory according to `pwd` and specified `ARCH`. Specify compilation target path of the image, set compiler and compilation flags, and link all platform- and arch-independent files in `$AM_HOME`.

3. In arch-specific configs, include configs in `$AM_HOME/scripts/riscv32-nemu.mk`. Include makefile of rv32, i.e. `$AM_HOME/scripts/isa/riscv.mk`, and makefile specified for nemu, i.e. `$AM_HOME/scripts/platform/nemu.mk` respectively in this makefile.

4. In `riscv.mk`, configure cross compiler and specified compile flags. Link platform- and arch-independent files with platform-related files (interface programs between `AM` and `nemu`, i.e. `trm`, `ioe` and `mpe`) together. Set the `image` variable to corresponding elf file (i.e. `am-kernels/kernels/build/hello-riscv32-nemu.elf`). Then bind `run` and `gdb` commands with make commands of `nemu` and pass the elf file to `nemu` as `make` args.

5. Back to `AM` makefile, specify compile rules to generate `elf` file. Compile single `.c` files to `.o`, build dependent library and link `.os` with it, form the final `elf` file.