Explanation of Makefile (for Linux)

Note: The same command will be used for windows system. Only the toolchain is already adder to PATH. So, the directory names are not required. Required tool can be invoked using their command. Also, the cleaning of temporary file (last line) is done by `del` command in Windows system.

For the toolchain compiled using `newlib/multilib` the `riscv32-*` command can be replaced by `riscv64-*` and still can be used for compilation of 32 bit architecture (RV32I and so on). It is specified in compiler option `-mabi=ilp32 -march=rv32i` during compilation.

This Makefile is used to compile and link a firmware project for a **RISC-V 32-bit embedded system**. It automates the process of compiling C and Assembly source files, linking them with a linker script, and then generating various output formats like a binary, a hexadecimal file, and a disassembly dump.

Let's break down each part of the Makefile:

Variable Definitions

These lines define variables that make the Makefile more readable and maintainable.

- `RISCV_TOOLCHAIN_DIR = /home/kiit/riscv32/bin` : This specifies the **base directory for your RISC-V toolchain**. This is where your RISC-V specific compilers, assemblers, and other utilities are located.
- `GCC = $(RISCV_TOOLCHAIN_DIR)/riscv32-unknown-elf-gcc` : Defines the **C compiler** executable. It uses the `RISCV_TOOLCHAIN_DIR` variable to construct the full path to the `riscv32-unknown-elf-gcc` compiler, which is a GCC cross-compiler for bare-metal RISC-V 32-bit targets.

- `OBJ_DUMP = $(RISCV_TOOLCHAIN_DIR)/riscv32-unknown-elf-objdump` : Defines the **object dump utility**. This tool is used to display information about object files, including disassembly.
- `OBJ_COPY = $(RISCV_TOOLCHAIN_DIR)/riscv32-unknown-elf-objcopy` : Defines the **object copy utility**. This tool is used to copy and translate object files, often to change their format (e.g., from ELF to binary).
- `C_SRC_DIR = .` : Specifies that the **C source files are in the current directory**.
- `ASM_SRC_DIR = .` : Specifies that the **Assembly source files are in the current directory**.
- `LDS_SRC_DIR = .` : Specifies that the **linker script files are in the current directory**.
- `RESULT_DIR = .` : Specifies that all **generated output files should be placed in the current directory**.
- `MEM_SIZE = 1024` : Defines a **memory size**, likely in bytes, which is used by the `makehex.py` script.

Default Target

- `all: $(RESULT_DIR)/firmware.hex $(RESULT_DIR)/dumpfile` : This is the **default target** that gets executed when you simply run `make` without any arguments. It declares that to build "all", it needs to produce two files: `firmware.hex` (the final hexadecimal file for programming) and `dumpfile` (a disassembly of the firmware).

Build Rules

These rules define how to create the various output files from their dependencies.

- `$(RESULT_DIR)/firmware.hex: $(RESULT_DIR)/firmware.bin makehex.py`

    - `python3 makehex.py $< $(MEM_SIZE) > $@` : This rule builds the **`firmware.hex` file**.
        - It depends on `firmware.bin` and a Python script `makehex.py`.
        - `$<` is an automatic variable representing the first prerequisite (here, `$(RESULT_DIR)/firmware.bin`).
        - `$@` is an automatic variable representing the target (here, `$(RESULT_DIR)/firmware.hex`).
        - The command executes `makehex.py` with the binary file and `MEM_SIZE` as arguments, redirecting the script's output (the hex file content) into `firmware.hex`.

- `$(RESULT_DIR)/firmware.bin: $(C_SRC_DIR)/firmware.elf`
    - `$(OBJ_COPY) -O binary $< $@` : This rule creates the **raw binary image (`firmware.bin`)** from the ELF executable.
        - It depends on `firmware.elf`.
        - `$(OBJ_COPY) -O binary` tells `objcopy` to output the file in binary format.
        - This binary file is typically what's programmed directly onto the microcontroller's flash memory.

- `$(RESULT_DIR)/dumpfile: $(C_SRC_DIR)/firmware.elf`
    - `$(OBJ_DUMP) -d $^ > $@` : This rule generates a **disassembly dump (`dumpfile`)** of the ELF executable.
        - It depends on `firmware.elf`.
        - `$^` is an automatic variable representing all prerequisites (here, `$(C_SRC_DIR)/firmware.elf`).
        - `$(OBJ_DUMP) -d` disassembles the executable code sections. The output is redirected to `dumpfile`. This is very useful for debugging and

understanding the compiled code.

- `$(C_SRC_DIR)/firmware.elf: $(C_SRC_DIR)/main.o $(ASM_SRC_DIR)/start.o sections.lds`
    - `$(GCC) -Og -mabi=ilp32 -march=rv32i -ffreestanding -nostdlib -o $@ -Wl,--build-id=none,-Bstatic,-T,sections.lds,-Map,$(RESULT_DIR)/firmware.map,--strip-debug $(ASM_SRC_DIR)/start.o $(C_SRC_DIR)/main.o -lgcc` : This is the **linking rule** that creates the **final ELF executable (`firmware.elf`)**.
        - It depends on `main.o` (compiled C code), `start.o` (compiled assembly startup code), and `sections.lds` (the linker script).
        - `-Og` : Optimization level for debugging.
        - `-mabi=ilp32` : Specifies the Application Binary Interface (ABI) as “ILP32” (int, long, pointer are 32-bit). This is standard for RV32.
        - `-march=rv32i` : Specifies the RISC-V instruction set architecture as RV32I (Integer base instruction set).
        - `-ffreestanding` : Indicates a “freestanding” environment, meaning no operating system is assumed (common for embedded systems).
        - `-nostdlib` : Prevents linking against the standard C library. You’re building a bare-metal application.
        - `-o $@` : Specifies the output file name as the target (`firmware.elf`).
        - `-Wl,...` : Passes options to the linker.
            - `--build-id=none` : Disables the generation of a build ID.
            - `-Bstatic` : Links static libraries only.
            - `-T,sections.lds` : Tells the linker to use `sections.lds` as the **linker script**. This script defines memory regions and how different sections of your code and data are placed in memory.

        - `-Map,$(RESULT_DIR)/firmware.map` : Generates a **map file (`firmware.map`)** which shows the memory layout of your program.
        - `--strip-debug` : Removes debugging symbols from the final executable.
    - `$(ASM_SRC_DIR)/start.o $(C_SRC_DIR)/main.o` : The object files to be linked.
    - `-lgcc` : Links against `libgcc`, which provides GCC's runtime support routines (e.g., for integer division or floating-point operations if used).
- `$(C_SRC_DIR)/main.o: $(ASM_SRC_DIR)/main.c`
    - `$(GCC) -c -Iinclude/ -mabi=ilp32 -march=rv32i -Og --std=c99 -ffreestanding -nostdlib -o $@ $<` : This rule compiles the **C source file (`main.c`)** into an object file (`main.o`).
        - It depends on `main.c`.
        - `-c` : Compiles the source file but does not link.
        - `-Iinclude/` : Adds the `include/` directory to the list of paths where the compiler searches for header files.
        - `--std=c99` : Compiles the C code adhering to the C99 standard.
        - Other flags (`-mabi=ilp32`, `-march=rv32i`, `-Og`, `-ffreestanding`, `-nostdlib`) are similar to those used in the linking step, ensuring consistent compilation settings.
- `$(ASM_SRC_DIR)/start.o: $(ASM_SRC_DIR)/start.S`
    - `$(GCC) -c -mabi=ilp32 -march=rv32i -o $@ $<` : This rule compiles the **assembly source file (`start.S`)** into an object file (`start.o`).
        - It depends on `start.S`.
        - This typically contains the startup code for the embedded system, like initializing the stack pointer and jumping to the main function.

Clean Target

- `clean:`
    - `rm *.o dumpfile *.elf *.hex *.bin *.map` : This rule defines the **`clean` target**. When you run `make clean` , it removes all intermediate and final build products, helping to ensure a fresh build.