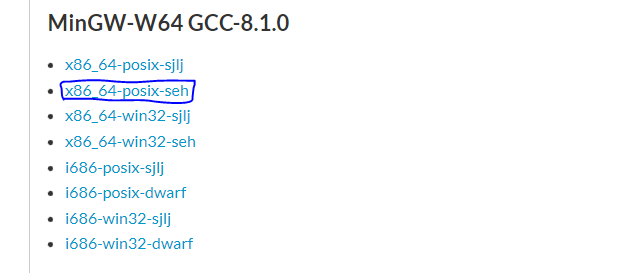


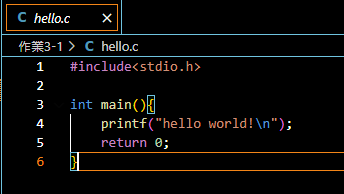
我原本用的是32位元的MinGW，題目說要能編譯出x86\_64指令及架構的程式，所以我找到了MinGW-w64這組工具鏈

<https://sourceforge.net/projects/mingw-w64/files/>

Posix是支持多線程的意思，seh是異常處理機制，網路上說sjlj效能較差開銷大，所以選擇seh



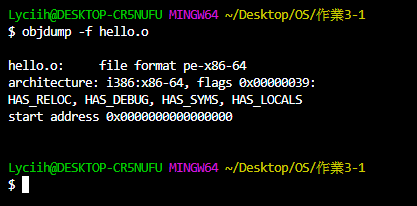
接著編寫一個hello.c



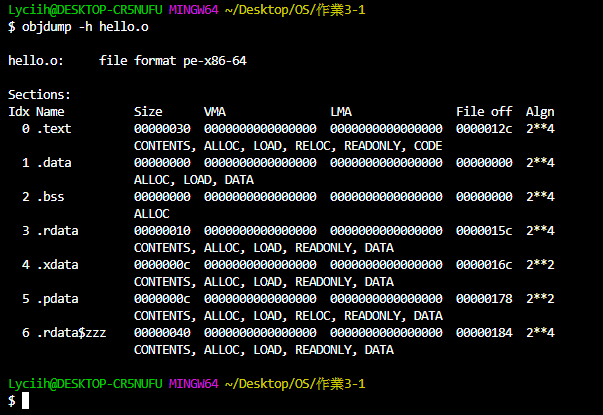
進行編譯



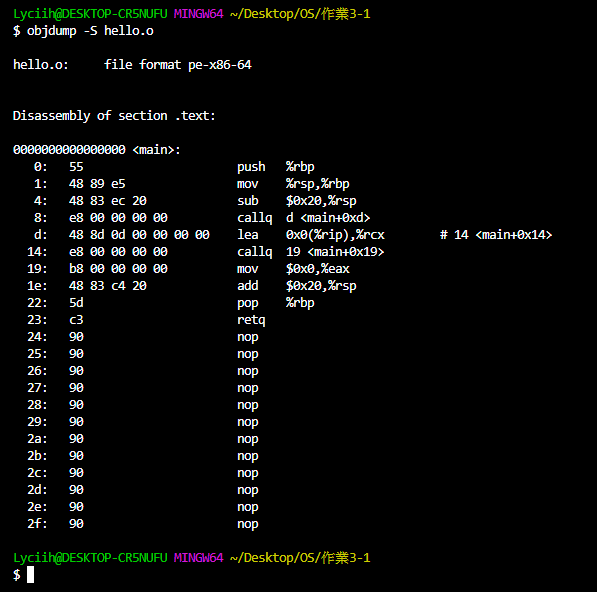
查看hello.o的檔頭訊息，使用objdump -f hello.o



查看hello.o的 Section header table，使用objdump -h hello.o



對hello.o進行反組譯，使用objdump -S hello.o



問題來了，竟然沒有出現對應的c code!!!

上網查詢，才知道要在編譯時加入 -g的參數，這樣才會把c code的資訊一起編譯進去

使用 gcc –c –g hello.c 重新編譯一次，成功

一張含有 文字 的圖片

自動產生的描述

一張含有 文字 的圖片

自動產生的描述

先建立檔案並編譯，再查看section header

一張含有 文字 的圖片

自動產生的描述

為了弄懂這題，首先要先了解組合語言中section到底是什麼，於是我查出每個片段名的意思

<https://learn.microsoft.com/en-us/windows/win32/debug/pe-format?redirectedfrom=MSDN#section-flags>

在微軟的網頁找到了PE fromat，裡面有介紹每個片段的表格

一張含有 桌 的圖片

自動產生的描述一張含有 文字 的圖片

自動產生的描述一張含有 文字 的圖片

自動產生的描述一張含有 文字 的圖片

自動產生的描述

.text段 : 用來保存實際的程式碼的地方

.data段 : 已經初始化的資料

.bss段 : 存放宣吿變數的地方(還沒有初始值)

.rdata段 : 已初始化的唯讀資料

.xdata段 : 儲存異常訊息的地方(自由格式)

.pdata段 : 儲存異常訊息的地方

剩下的目前還沒找到

所以我猜測:

global\_init在.data段 、global\_const在.rdata段、static\_var在.data段、static\_var\_uninit在.bss段

auto\_var在.data段、"hello world!\n"在.data段

一張含有 文字 的圖片

自動產生的描述

為了驗證我的想法，我使用以下指令



global\_init和static\_var都在.data段沒錯

一張含有 文字 的圖片

自動產生的描述

static\_var\_uninit在.bss段沒錯

一張含有 文字 的圖片

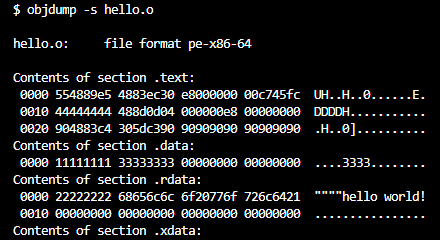
自動產生的描述

global\_const在.rdata段沒錯

一張含有 文字 的圖片

自動產生的描述

用了另一個指令參數 -s(小寫) hello world在.rdata段



但是卻找不到auto\_var

於是用objdump -S(大寫) hello.o 再看一次程式碼一張含有 文字 的圖片

自動產生的描述

發現 int auto\_var = 0x44444444 這段竟然直接放在程式碼段裡

查詢到維基百科 .bss和.data所存的變數都是全域變數

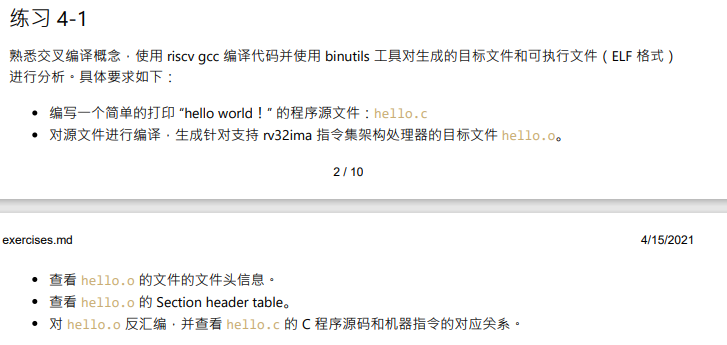


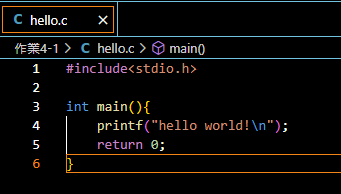


我的推論

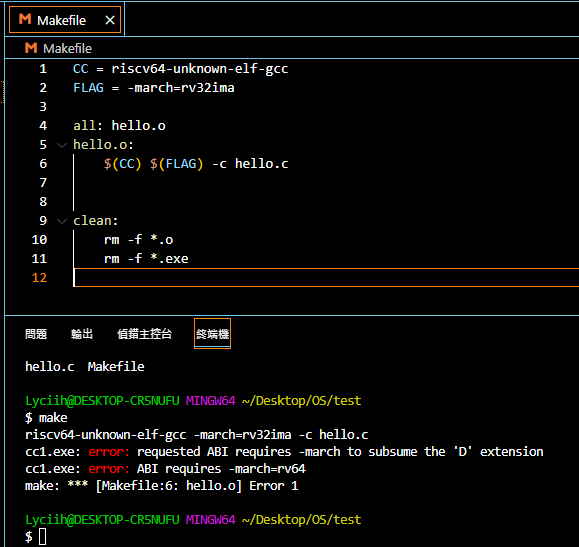
Global是全域的意思，const是常量要放在唯讀，static的記憶體位置是固定的所以也當作全域

int auto\_var是區域變數，在執行時才宣告。

首先寫一個hello.c



為了不需要每次都打很長的 riscv64-unknown-elf-gcc 指令及參數，我練習寫一個Makefile，並且先只用-march=rv32ima參數試試



結果產生錯誤，上面說還需要ABI的參數

查詢一下ABI是什麼東西

https://blog.csdn.net/zoomdy/article/details/79353313

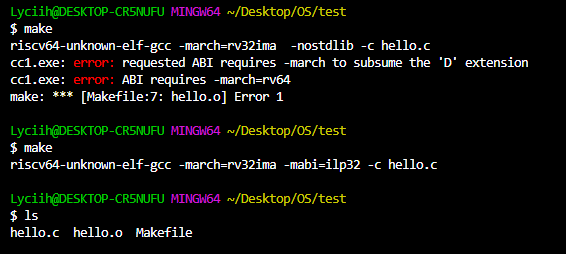




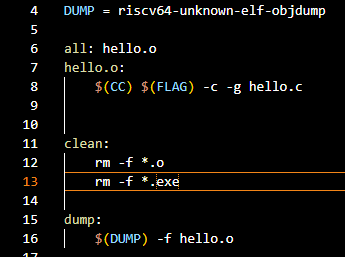
於是在makefile上加入-mbi=ilp32 (ilp32意思是說，float和double數都由整數寄存器傳遞)



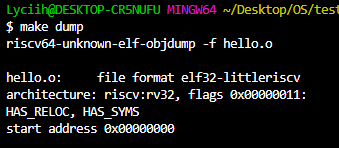
編譯成功



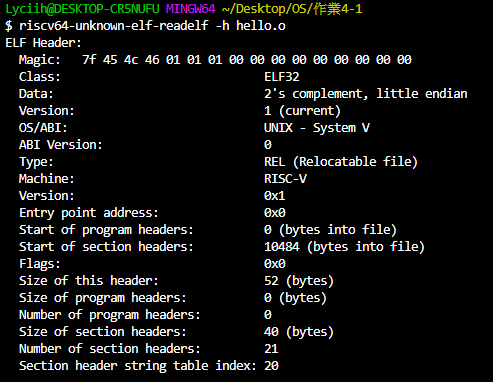
在Makefile中加入riscv64-unknown-elf-objdump 把它設為偽目標



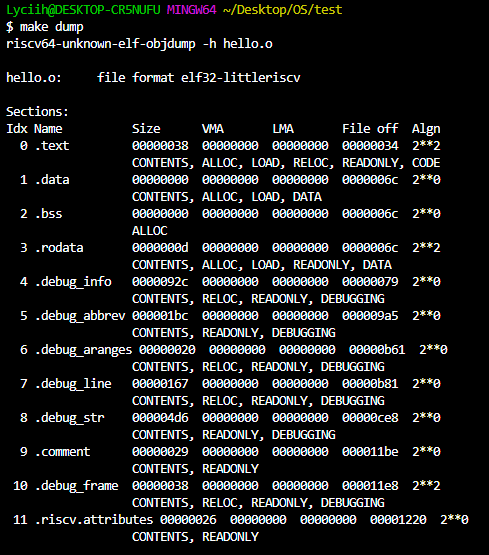
查看檔頭訊息



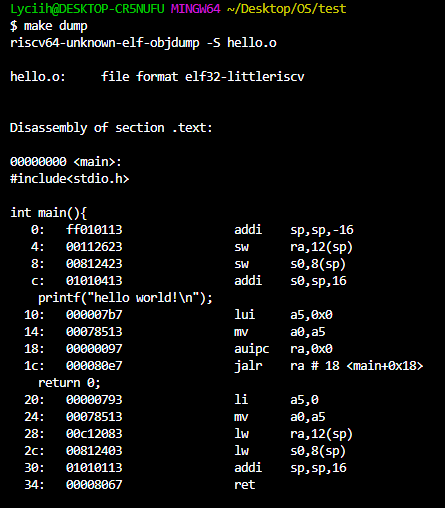
網路上有另一種作法是用readelf這個命令



查看Section header table



進行反組譯



最後我決定來查詢每個參數的意義

riscv64-unknown-elf-gcc -nostdlib -fno-builtin -march=rv32ima -mabi=ilp32 -g -Wall

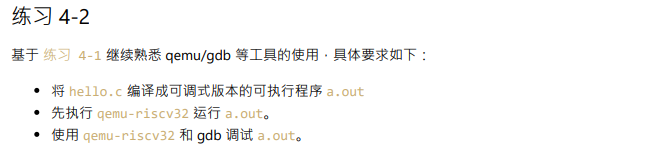
| -nostdlib | Do not look for object files in standard path.  不要在基本的路徑中尋找.o檔 |
| --- | --- |
| (-fno-pic)  -fno-builtin | don't generate position-independent code (default)  預設不生成與位置無關的代碼，這裡使用builtin選項 |
| -march | Generate code for given RISC-V ISA (e.g. RV64IM)  指定RISC-V的指令集架構，這裡使用rv32ima |
| -mabi | Specify integer and floating-point calling convention.  指定整數和浮點數的表示方式 |
| -g | Generate debug information in default format.  產生gdb中可以看到的debug資訊 |
| -Wall | Enable most warning messages.  打開全部的警告訊息 |

rv32ima: 由rv32i、m、a 三個選項組成

rv32i: RV32I Base Integer Instruction Set (32-bit 基本整數指令集，有 32個 32-bit暫存器)

m: Standard Extension for Integer Multiplication and Divison (新增了整數乘法除法指令)

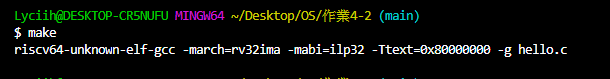
a: Standard Extension for Atomic Instructions (新增了 Atomic相關的指令)



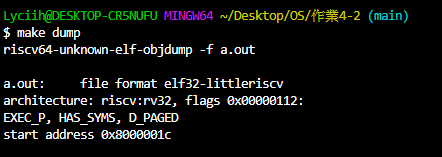
由於要直接輸出a.out，於是把-c參數拿掉，要用qemu運行，所以將起始點設在0x80000000



進行編譯



看一下檔頭資訊



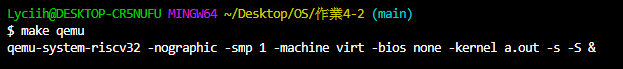
接著使用qemu來執行看看

先來了解教材中qemu的參數

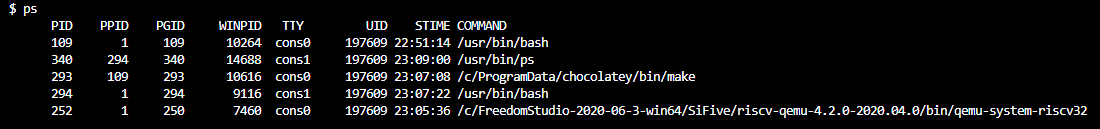
qemu-system-riscv32 -nographic -smp 1 -machine virt -bios none -kernel os.elf -s -S &

| -nographic | disable graphical output and redirect serial I/Os to console  關閉圖形輸出，並將I/O對接到當前的終端機 |
| --- | --- |
| -smp | set the number of CPUs  設定cpu的核心數 |
| -machine | selects emulated machine  指定模擬的機器，這裡選virt(虛擬的) |
| -bios | set the filename for the BIOS  指定bios設定檔，這裡沒有，所以是none |
| -kernel | 指定使用的kernel 這裡用我們寫的os.elf |
| -s | shorthand for -gdb tcp::1234  縮寫(設定讓gdb用來除錯的端口為1234號) |
| -S | freeze CPU at startup (use 'c' to start execution)  剛啟動時凍結CPU的運行(用來等gdb打開) |
| & | (Linux指令參數)在背景執行 |

設定完Makefile檔後便執行看看

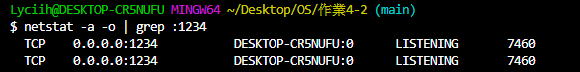


使用ps指令查看運行中的程式，qemu執行中，等待我們用gdb去測試



另外我們查看1234port是不是真的被我們的qemu使用。

PS中的winpid是7460，而使用netstat查詢占用port的程式，也是7460，沒錯。



接著使用gdb來調試。

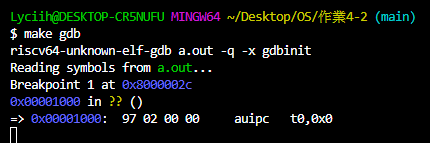
一樣先瞭解gdb的各種參數

riscv64-unknown-elf-gdb os.elf -q -x gdbinit

| -q | Do not print version number on startup.  在啟動時不顯示版本資訊 |
| --- | --- |
| -x | Execute GDB commands from FILE.  啟動gdb後執行指定檔案中的指令 |

Gdbinit檔中的gdb指令

| set disassemble-next-line on | 反組譯下一行要執行的程式 |
| --- | --- |
| b \_start | 在 \_start 的位置設立斷點 |
| target remote : 1234 | 設定遠端除錯的端口 這裡是1234號 |
| c | 讓CPU開始執行程式 |



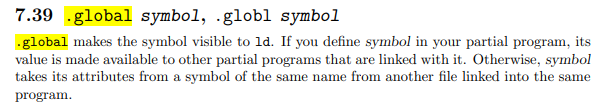
成功運行反組譯的功能，但是卻卡住了。

課堂上詢問老師，原來還缺乏啟動的開機程式以及可以輸出文字的uart驅動，所以我們回頭來把這兩個程式補上。

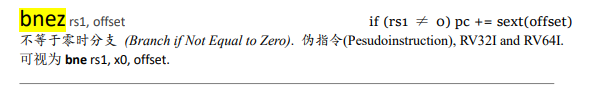
因此我們先用組合語言來寫一個開機的程式。

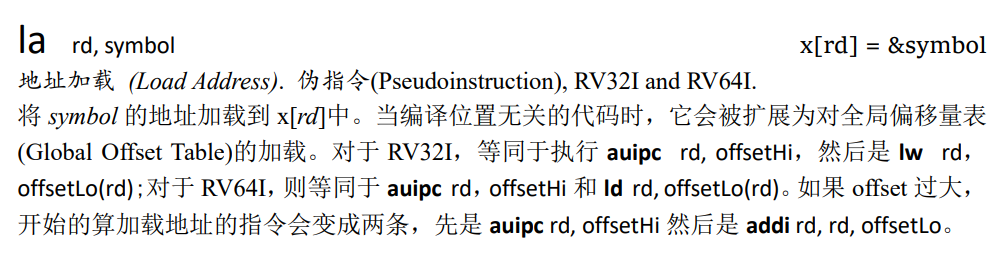
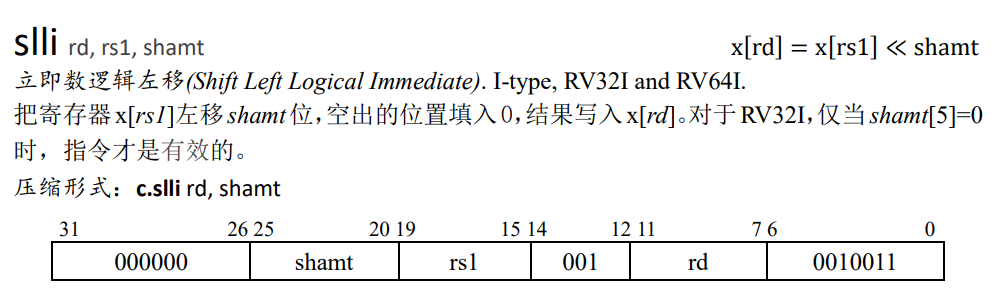


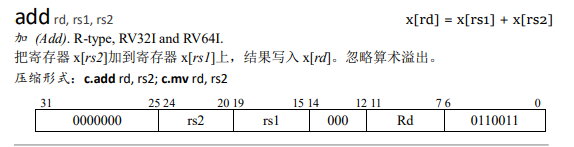
翻閱ref資料，可以查到每個指令的意思

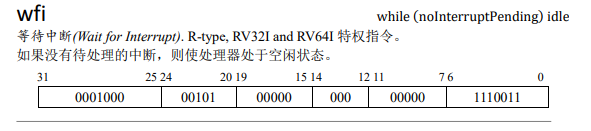


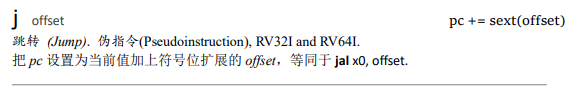






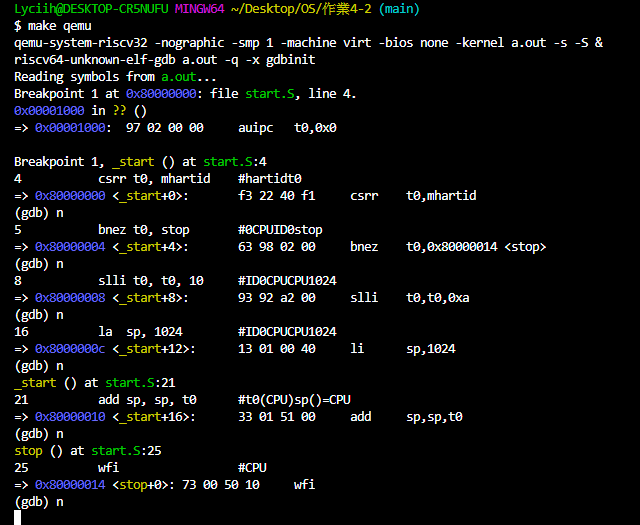






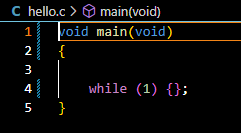
將開機程式單獨編譯，再用gdb測試，成功!!

gdb偵測到的最後一行是wfi，因為我們的開機程式最後並沒有跳到kernel的指令，所以CPU執行完最後一行就休息了

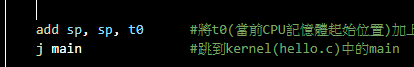


接著來寫kernel

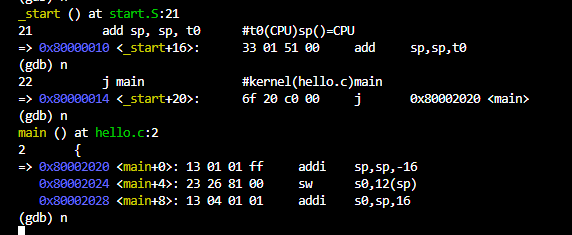
因為還不能用printf，先將hello.c改為while就好



然後在開機程式的最後一行加上跳到main的指令



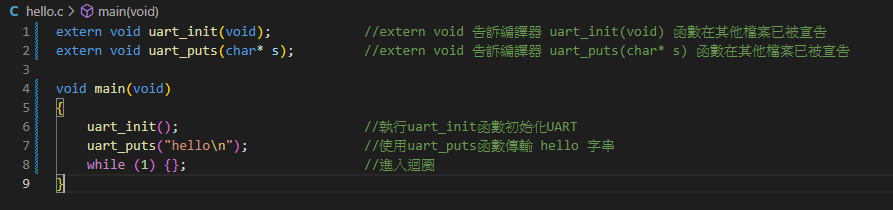
用gdb測試，成功跳到main了!!

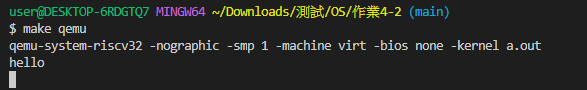


接著來寫uart驅動程式

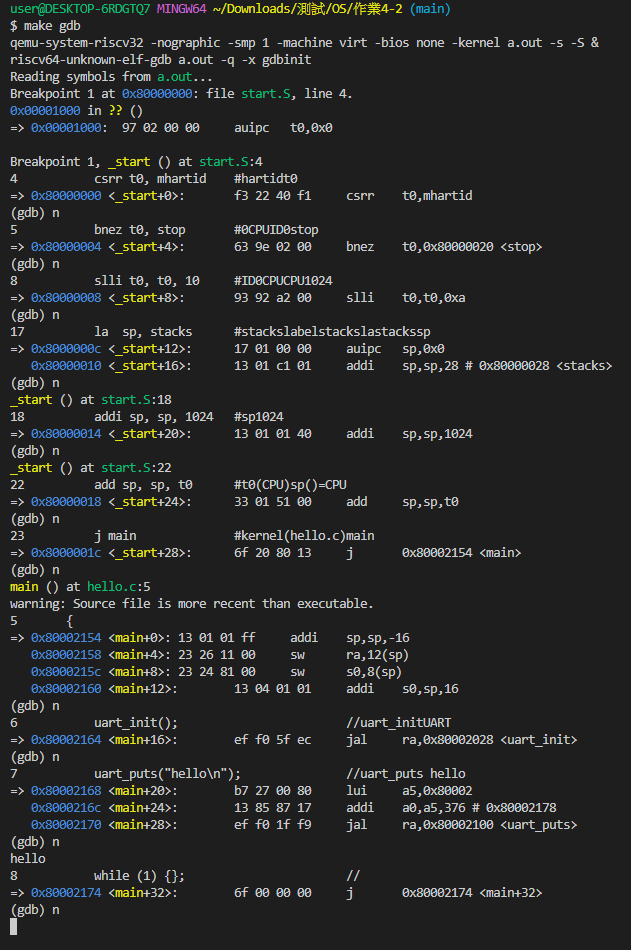
回到我們的hello.c

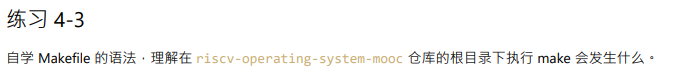
使用剛寫好的uart驅動函數來輸出文字

make完用qemu執行看看



成功!! 接著用gdb執行看看





為避免排版上篇幅過長，我把我對命令的理解直接打成註釋

