

Lab1

一、实验目的

1. 搭建AArch64交叉编译工具链及QEMU模拟器环境
2. 开发基础裸机程序框架
3. 实现工程化构建系统
4. 配置调试支持环境
5. 建立自动化构建脚本

二、实验过程

1. 环境配置

1.1 交叉工具链安装

获取ARM官方工具链（版本11.2）：

wget

```
https://developer.arm.com/-/media/Files/download  
s/gnu/11.2-2022.02/binrel/gcc-arm-11.2-2022.02-  
x86_64-aarch64-none-elf.tar.xz
```

解压并重命名：

```
tar -xzf gcc-arm-11.2-2022.02-x86_64-aarch64-  
none-elf.tar.xz
```

```
mv gcc-arm-11.2-2022.02-x86_64-aarch64-none-elf  
aarch64-none-elf
```

1.2 环境变量配置

编辑 `~/.bashrc` 文件添加工具链路径：

```
export  
PATH=$PATH:/home/smile_laughter/os/aarch64-none-  
elf/bin
```

生效配置：

```
source ~/.zshrc
```

1.3 验证安装

执行版本检查命令确认安装成功：

```
aarch64-none-elf-gcc --version
```

```
> aarch64-none-elf-gcc --version  
aarch64-none-elf-gcc (GNU Toolchain for the Arm Architecture 11.2-2022.02 (arm-11.14)) 11.2.1 202  
20111  
Copyright (C) 2021 Free Software Foundation, Inc.  
This is free software; see the source for copying conditions. There is NO  
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  
  
~/.config/ezsh/zshrc base 06:41:54 PM
```

1.4 QEMU安装

```
sudo apt-get update
sudo apt-get install qemu qemu-system
```

1.5 CMake安装

```
sudo apt-get install cmake
```

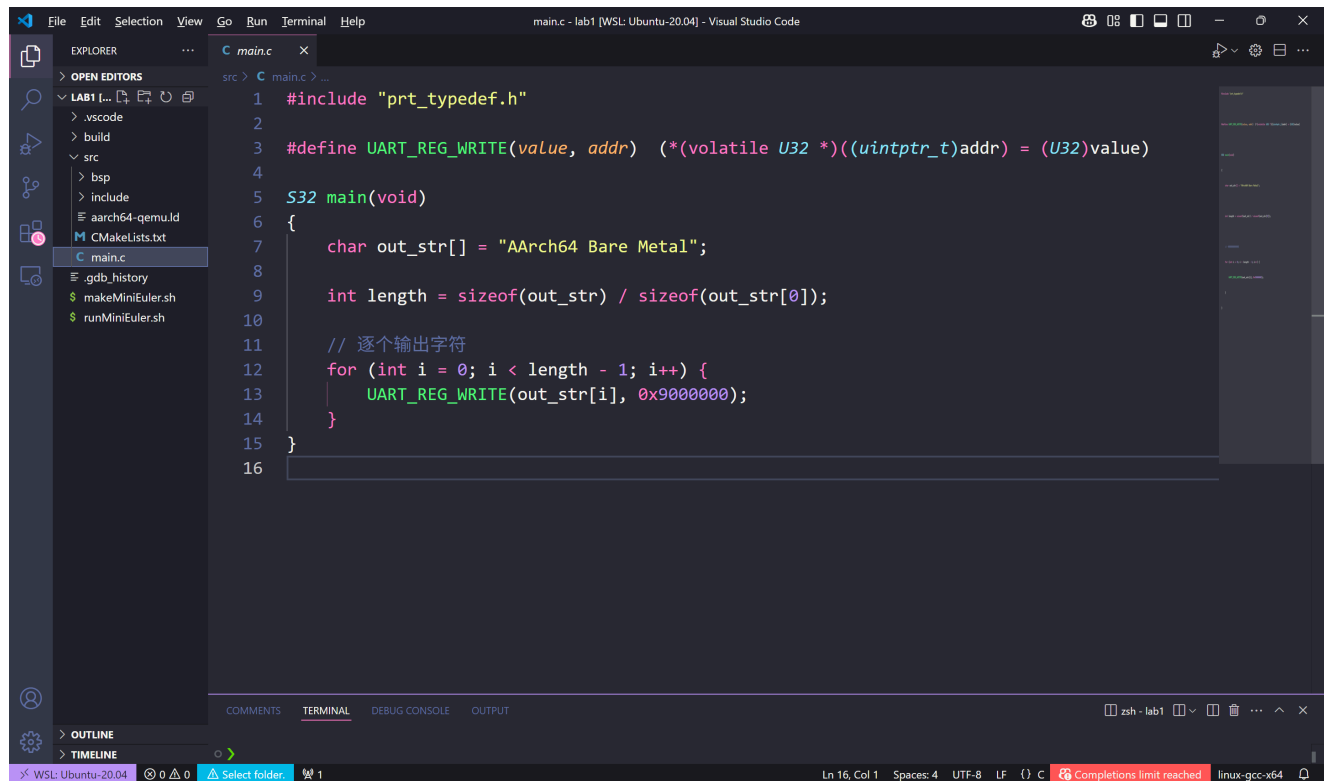
2. 裸机程序开发

2.1 项目目录结构

```
src/
├── bsp/
│   ├── CMakeLists.txt
│   ├── start.S
│   └── prt_reset_vector.S
├── include/
│   └── prt_typedef.h
└── main.c
```

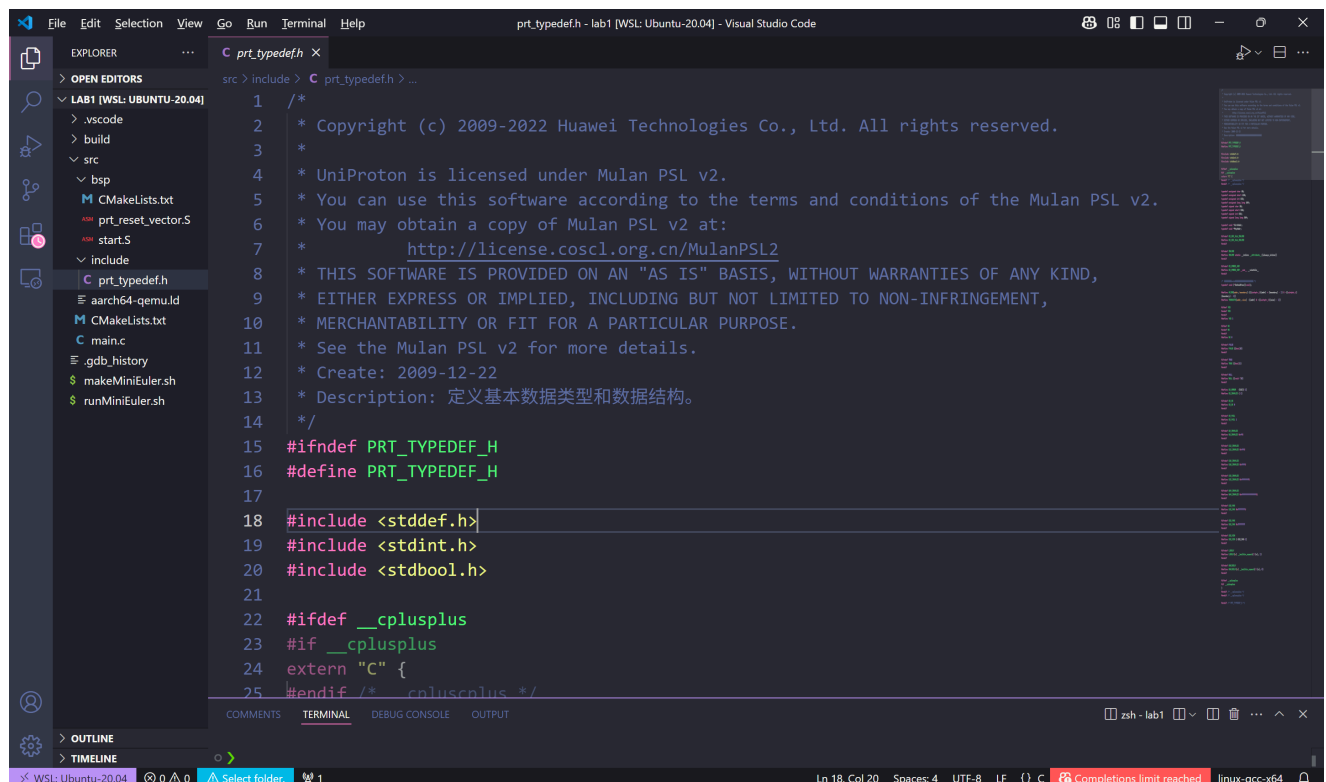
2.2 核心代码实现

main.c：实现基础输出功能



```
1 #include "prt_typedef.h"
2
3 #define UART_REG_WRITE(value, addr) (*(volatile U32 *)((uintptr_t)addr) = (U32)value)
4
5 S32 main(void)
6 {
7     char out_str[] = "AArch64 Bare Metal";
8
9     int length = sizeof(out_str) / sizeof(out_str[0]);
10
11     // 逐个输出字符
12     for (int i = 0; i < length - 1; i++) {
13         UART_REG_WRITE(out_str[i], 0x90000000);
14     }
15 }
16
```

prt_typedef.h：定义基础数据类型和宏



```
1 /*
2  * Copyright (c) 2009-2022 Huawei Technologies Co., Ltd. All rights reserved.
3  *
4  * UniProton is licensed under Mulan PSL v2.
5  * You can use this software according to the terms and conditions of the Mulan PSL v2.
6  * You may obtain a copy of Mulan PSL v2 at:
7  * http://license.coscl.org.cn/MulanPSL2
8  * THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OF ANY KIND,
9  * EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO NON-INFRINGEMENT,
10  * MERCHANTABILITY OR FIT FOR A PARTICULAR PURPOSE.
11  * See the Mulan PSL v2 for more details.
12  * Create: 2009-12-22
13  * Description: 定义基本数据类型和数据结构。
14  */
15 #ifndef PRT_TYPEDEF_H
16 #define PRT_TYPEDEF_H
17
18 #include <stddef.h>
19 #include <stdint.h>
20 #include <stdbool.h>
21
22 #ifdef __cplusplus
23 #if __cplusplus
24 extern "C" {
25 #endif /* cplusplus */
26
```

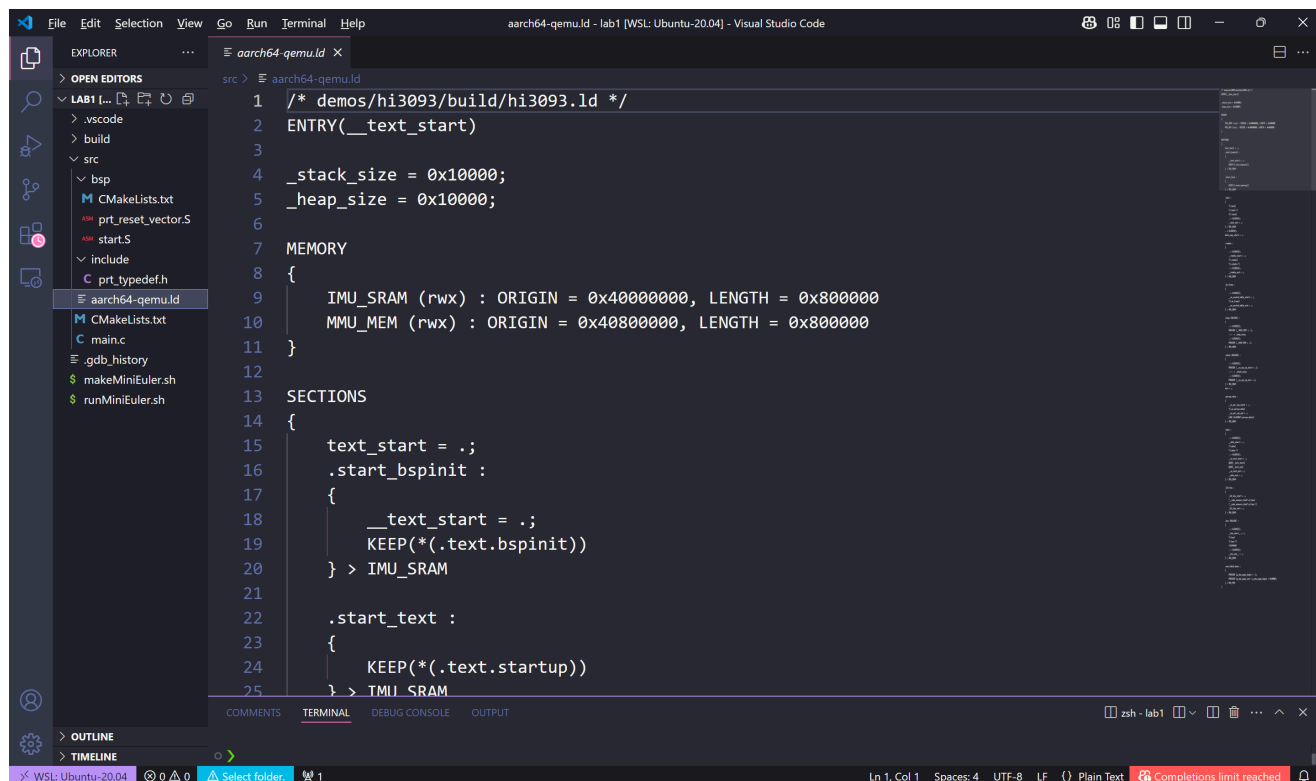
start.S：系统启动和EL级别检查

```
1  .global  OsEnterMain
2  .extern  __os_sys_sp_end
3
4  .type    start, function
5  .section .text.bspinit, "ax"
6  .balign  4
7
8  .global  OsElxState
9  .type    OsElxState, @function
10 OsElxState:
11 MRS     x6, CurrentEL // 把系统寄存器 CurrentEL 的值读入到通用寄存器 x6 中
12 MOV     x2, #0x4 // CurrentEL EL1: bits [3:2] = 0b01
13 CMP     w6, w2
14
15 BEQ     Start // 若 CurrentEl 为 EL1 级别, 跳转到 Start 处执行, 否则死循环。
16
17 OsEl2Entry:
18 B       OsEl2Entry
19
20 Start:
21 LDR     x1, =__os_sys_sp_end // 符号在ld文件中定义
22 BIC     sp, x1, #0xf // 设置栈指针
23
24 B       OsEnterMain
25
```

prt_reset_vector.S: 主程序入口和异常处理

```
1  DAIF_MASK = 0x1C0 // disable SError Abort, IRQ, FIQ
2
3  .global  OsVectorTable
4  .global  OsEnterMain
5
6  .section .text.startup, "ax"
7  OsEnterMain:
8  BL      main
9
10 MOV     x2, DAIF_MASK // bits [9:6] disable SError Abort, IRQ, FIQ
11 MSR     DAIF, x2 // 把通用寄存器 x2 的值写入系统寄存器 DAIF 中
12
13 EXITLOOP:
14 B       EXITLOOP
15
```

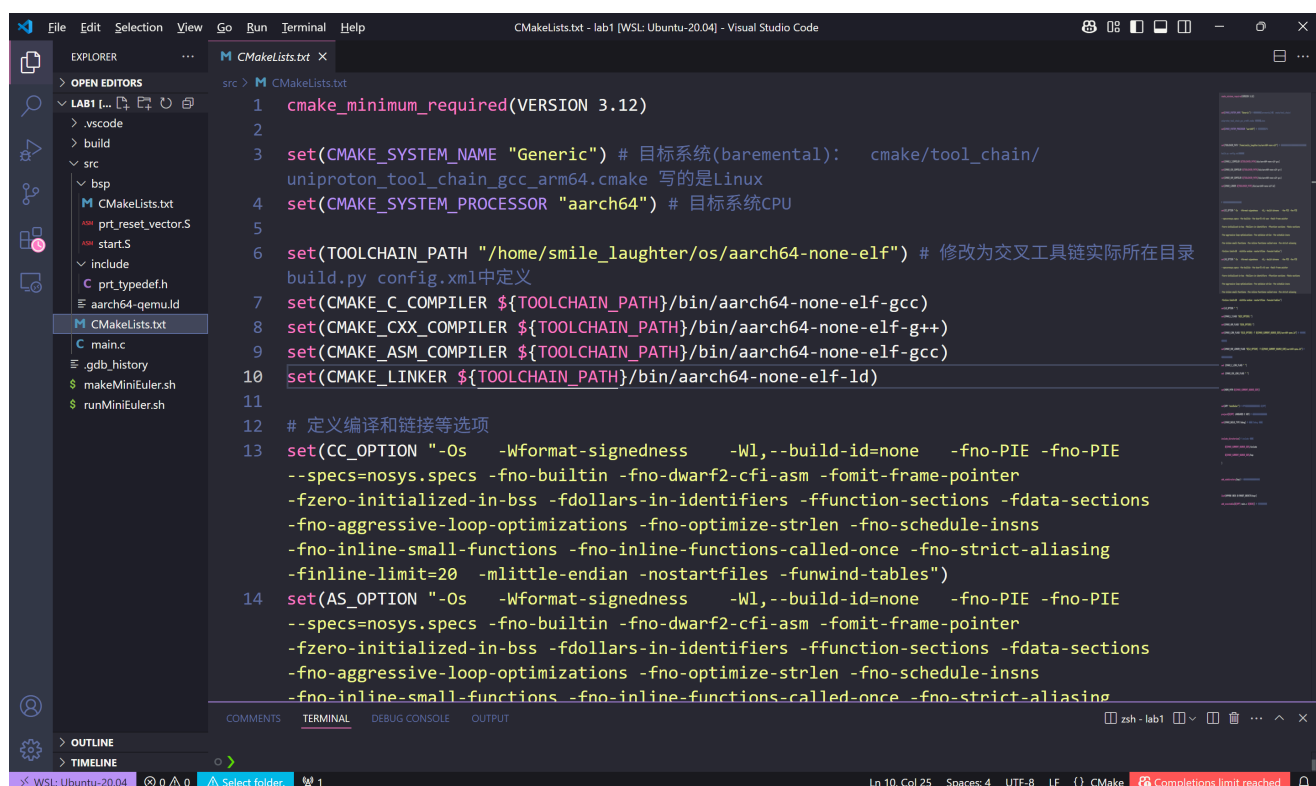
aarch64-qemu.ld: 内存布局定义



```
1 /* demos/hi3093/build/hi3093.ld */
2 ENTRY(__text_start)
3
4 _stack_size = 0x10000;
5 _heap_size = 0x10000;
6
7 MEMORY
8 {
9     IMU_SRAM (rwx) : ORIGIN = 0x40000000, LENGTH = 0x800000
10    MMU_MEM (rwx) : ORIGIN = 0x40800000, LENGTH = 0x800000
11 }
12
13 SECTIONS
14 {
15     text_start = .;
16     .start_bspinit :
17     {
18         __text_start = .;
19         KEEP(*(.text.bspinit))
20     } > IMU_SRAM
21
22     .start_text :
23     {
24         KEEP(*(.text.startup))
25     } > MMU_SRAM
```

3. 工程构建

3.1 主CMakeLists.txt



```
1 cmake_minimum_required(VERSION 3.12)
2
3 set(CMAKE_SYSTEM_NAME "Generic") # 目标系统(baremetal):  cmake/tool_chain/
4                                     uniproton_tool_chain_gcc_arm64.cmake 写的是Linux
5 set(CMAKE_SYSTEM_PROCESSOR "aarch64") # 目标系统CPU
6
7 set(TOOLCHAIN_PATH "/home/smile_laughter/os/aarch64-none-elf") # 修改为交叉工具链实际所在目录
8                                     build.py config.xml中定义
9 set(CMAKE_C_COMPILER "${TOOLCHAIN_PATH}/bin/aarch64-none-elf-gcc)
10 set(CMAKE_CXX_COMPILER "${TOOLCHAIN_PATH}/bin/aarch64-none-elf-g++)
11 set(CMAKE_ASM_COMPILER "${TOOLCHAIN_PATH}/bin/aarch64-none-elf-gcc)
12 set(CMAKE_LINKER "${TOOLCHAIN_PATH}/bin/aarch64-none-elf-ld)
13
14 # 定义编译和链接等选项
15 set(CC_OPTION "-Os -Wformat-signedness -Wl,--build-id=none -fno-PIC -fno-PIC
16 --specs=nosys.specs -fno-builtin -fno-dwarf2-cfi-asm -fomit-frame-pointer
17 -fzero-initialized-in-bss -fdollars-in-identifiers -ffunction-sections -fdata-sections
18 -fno-aggressive-loop-optimizations -fno-optimize-strlen -fno-schedule-insns
19 -fno-inline-small-functions -fno-inline-functions-called-once -fno-strict-aliasing
20 -finline-limit=20 -mlittle-endian -nostartfiles -funwind-tables")
21
22 set(AS_OPTION "-Os -Wformat-signedness -Wl,--build-id=none -fno-PIC -fno-PIC
23 --specs=nosys.specs -fno-builtin -fno-dwarf2-cfi-asm -fomit-frame-pointer
24 -fzero-initialized-in-bss -fdollars-in-identifiers -ffunction-sections -fdata-sections
25 -fno-aggressive-loop-optimizations -fno-optimize-strlen -fno-schedule-insns
26 -fno-inline-small-functions -fno-inline-functions-called-once -fno-strict-aliasing
```

3.2 BSP层CMakeLists.txt

```
set(SRCS start.S prt_reset_vector.S)
add_library(bsp OBJECT ${SRCS})
```

4. 编译运行

4.1 编译流程

```
mkdir build && cd build
cmake ../src
cmake --build .
```

4.2 依赖问题解决

安装Python 3.6开发环境:

```
sudo apt-get install -y build-essential libssl-
dev libffi-dev
wget
https://www.python.org/ftp/python/3.6.15/Python-
3.6.15.tar.xz
tar -xf Python-3.6.15.tar.xz
cd Python-3.6.15
./configure --enable-shared
make && sudo make install
```

4.3 运行测试

```
qemu-system-aarch64 -machine virt -m 1024M \  
    -cpu cortex-a53 -nographic -kernel  
build/miniEuler
```

预期输出： AArch64 Bare Metal

三、测试分析

1. 运行原理

QEMU启动参数说明：

- qemu-system-aarch64代表启动armv8架构的虚拟机
- -machine virt 来指定虚拟机类型为 virt
- -m 1024M 来指定虚拟机内存大小为 1024M
- -cpu cortex-a53 来指定虚拟机的 CPU 类型为 cortex-a53
- -nographic 来禁用图形界面
- -kernel build/miniEuler 来指定内核映像文件为我们自己的操作系统内核miniEuler
- -S 选项表示在启动时暂停虚拟机并等待 gdb 连接。默认服务器端口为1234

2. 关键技术分析

main.c：定义了一个宏UART_REG_WRITE，用于实现将字符写入地址为0x90000000的位置，main.c文件的主要功能就是将字符串AArch64 Bare Metal逐个输入0x90000000处，实现字符串输出。关于为什么向内存地址0x90000000输入字符串就

可以实现字符串输入，这是一种操作系统与硬件交互的规定，这个地址被规定为操作系统与I/O设备之间的接口而非我们平常理解的地址。

start.S:

首先声明两个外部定义的变量OsEnterMain 和 os_sys_sp_end，其中OsEnterMain来自于prt_reset_vector.S，os_sys_sp_end来自于脚本aarch64-qemu.ld，然后定义.text.bspinit包含本行之后的代码，表示一个可分配可执行的段，接着定义整个程序的入口OsElxState，程序运行时会跳转到此处开始执行（入口的定义在链接脚本aarch64-qemu.ld中实现）程序开始运行时，首先会执行指令MRS x6,CurrentEL，将系统寄存器的值读入到通用寄存器x6

（MRS表示Move to register from system），然后比较该寄存器的低32位w6与0x4是否相等，由于CurrentEL系统寄存器中的索引第2位和第三位表示EL级别，所以等价于判断当前的EL级别是否为1，即内核态，若当前级别处于内核态，则进入start区，首先利用链接文件中定义的全局栈低指针__os_sys_sp_end对栈区进行初始化，然后跳转到prt_reset_vector.S中的OsEntermain进行执行；若当前级别处于用户态，则无法陷入操作系统内核执行内核指令，进入死循环。

prt_reset_vector.S: 跳转进入操作系统内核的主程序main.c，执行完毕后返回（BL指令返回），由于此时还未设置中断处理，故将DAIF寄存器中的SError、IRQ和FIQ位禁用，最后进入死循环。

链接脚本：本文件为链接脚本文件，首先通过ENTRY声明整个程序的入口为text_start，接着定义堆区、栈区大小均为0x10000，然后使用MEMORY定义两个内存空间IMU_SRAM与MMU_MEM，并定义了它们的起始位置分别为0x40000000,0x40800000、大小均为0x800000以及权限均为“rwx”，最后在SECTIONS中定义了text_start的具体位置，然后将start.S文件中的代码段.text.bspinit插入__text_start后，如此操作，实质上则定义了整个程序的入口为start.S文件中的OsElxState，以及定义了堆区空间和栈区空间的起始位置和结束位置，并将起始代码段、堆栈空间保存于内存空间IMU_SRAM。

作业

作业1

使用命令查看NZCV 寄存器

```
printf "N=%d Z=%d C=%d V=%d\n", $cpsr >> 31 & 1,
$cpsr >> 30 & 1, $cpsr >> 29 & 1, $cpsr >> 28&1
```

执行 CMP w6, w2 之前

```
sh aarch64-none-elf-gdb
0x000000004000001c ? b 0x40000030 <OsEnterMain>
0x0000000040000020 ? b 0x40000020 <OsEnterReset>
0x0000000040000024 ? udf #0
0x0000000040000028 ? .inst 0x400200d8 ; undefined
0x000000004000002c ? udf #0
Breakpoints
Expressions
History
Memory
Source
18 .type OsElxState, @function
19 // 在大多数情况下, @function 和 function 的作用是相同的, 都表示该符号是一个函数。
20 OsElxState:
21 MRS x6, CurrentEL // 把系统寄存器 CurrentEL 的值读入到通用寄存器 x6 中
22 MOV x2, #0x4 // CurrentEL EL1: bits [3:2] = 0b01
23 CMP w6, w2
24
25 BEQ Start // 若 CurrentEl 为 EL1 级别, 跳转到 Start 处执行, 否则死循环。
26
27 OsEl2Entry:
Variables

>>> printf "N=%d Z=%d C=%d V=%d\n", $cpsr >> 31 & 1, $cpsr >> 30 & 1, $cpsr >> 29 & 1, $cpsr >> 28 & 1
N=0 Z=1 C=0 V=0
>>>
```

执行 CMP w6, w2 之后

```
sh aarch64-none-elf-gdb
0x0000000040000020 ? b 0x40000020 <OsEnterReset>
0x0000000040000024 ? udf #0
0x0000000040000028 ? .inst 0x400200d8 ; undefined
0x000000004000002c ? udf #0
0x0000000040000030 ? bl 0x40000040 <main>
Breakpoints
Expressions
History
Memory
Source
20 OsElxState:
21 MRS x6, CurrentEL // 把系统寄存器 CurrentEL 的值读入到通用寄存器 x6 中
22 MOV x2, #0x4 // CurrentEL EL1: bits [3:2] = 0b01
23 CMP w6, w2
24
25 BEQ Start // 若 CurrentEl 为 EL1 级别, 跳转到 Start 处执行, 否则死循环。
26
27 OsEl2Entry:
28 B OsEl2Entry
29 // 无条件跳转到 OsEl2Entry 标签处, 导致程序进入死循环。
Variables

>>> printf "N=%d Z=%d C=%d V=%d\n", $cpsr >> 31 & 1, $cpsr >> 30 & 1, $cpsr >> 29 & 1, $cpsr >> 28 & 1
N=0 Z=1 C=1 V=0
>>>
```

分析:

Z=1 说明 $w6 - w2 == 0$, 即 $w6 == w2$, 因此 BEQ (branch if equal) 指令成立, 程序会跳转到 Start 的位置

作业2

程序入口分析

1. 入口脚本:

程序通过 `if __name__ == "__main__"` 启动, 解析命令行参数 (如 `cpu_plat`、`compile_option` 等), 设置默认参数后初始化 `Compile` 类实例。

2. 参数处理:

支持 6 个参数, 包括目标 CPU 类型 (`cpu_plat`)、编译模式 (如 `normal` / `coverity`)、运行平台类型 (`FPGA` / `SIM`) 等。未传参时使用默认值。

3. 多平台编译:

遍历 `globle.cpus_[cur_cpu_type]` 中定义的目标 CPU 平台, 为每个平台创建 `Compile` 实例并调用 `UniProtonCompile` 启动编译流程。

关键函数功能分析

1. `Compile.__init__()`

- 功能: 初始化编译环境, 包括路径配置 (如 `home_path`、`build_dir`)、编译选项 (如 `compile_option`)、平台信息 (通过 `getOsPlatform()` 检测 `arm64/x86`)。

2. `Compile.get_config()`

- 功能: 从 `config.xml` 加载目标平台的编译配置, 包

括编译器路径（`hcc_path`）、内核类型（`core`）、库类型（`lib_type`）等。

- **依赖**：调用 `get_cpu_info()` 解析 XML 配置，失败时终止进程。

3. `Compile.prepare_env()`

- **功能**：准备编译环境，依次调用：

1. `get_config()` 加载平台配置。
2. `setCmdEnv()` 设置日志路径和构建时间标签。
3. `SetCMakeEnviron()` 注入环境变量（如 `CPU_TYPE`、`HCC_PATH`），供 CMake 和 Makefile 使用。

4. `Compile.CMake()`

- **功能**：生成 CMake 构建文件。

- **流程**：

1. 创建临时构建目录（`build_tmp_dir`）和日志目录。
2. 根据编译模式拼接 `cmake` 命令（如 `fortify` 模式启用静态代码分析工具）。
3. 执行 CMake 命令生成 Makefile，失败时记录错误。

5. `Compile.make()`

- **功能**：执行 `make` 编译命令。

- **步骤**：

1. `make clean` 清理旧构建。
2. `make all` 编译目标代码。
3. `make install` 安装产物到指定路径（仅限 `normal / coverity` 模式）。

- **日志处理**：将编译输出重定向到日志文件，并格式化日志内容。

6. `Compile.UniProtonCompile()`

- **功能**：编译流程总控。
- **分支逻辑**：
 - 若 `cpu_type == 'clean'`：调用 `UniProton_clean()` 清理构建缓存。
 - 否则：根据 `make_choice` 遍历目标平台，调用 `SdkCompaille()` 执行完整编译流程。

7. `Compile.MakeBuildef()`

- **功能**：调用外部脚本 `make_buildef` 生成构建定义文件（如 `prt_buildef.h`），用于配置内核和硬件平台参数。

构建流程总结

1. **环境初始化**：解析参数、加载配置、设置环境变量。
2. **代码生成**：通过 CMake 生成平台相关的 Makefile。
3. **编译与安装**：执行 `make` 命令编译目标代码并安装产物。
4. **多平台支持**：支持按需编译多个 CPU 架构，通过配置文件实现跨平台兼容性。
5. **清理机制**：提供 `clean` 选项删除临时文件和输出目录。

作业3

1.在wsl远程调试项目：

首先在打开一个wsl终端（对应虚拟机终端），并启动程序，加上-S指令冻结CPU，使程序在入口停止等待调试：

```
> sh runMiniEuler.sh -S
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build
/miniEuler -s -S
```

然后再新打开一个wsl终端（这里对应虚拟机的终端），在新的终端中启动调试客户端：

```
> aarch64-none-elf-gdb build/miniEuler
GNU gdb (GNU Toolchain for the Arm Architecture 11.2-2022.02 (arm-11.14)) 11.2.90.20220202-git
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=aarch64-none-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.linaro.org/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
/home/smile_laughter/.gdbinit:1: Error in sourced command file:
Undefined command: "dashboard". Try "help".
Reading symbols from build/miniEuler...
```

在gdb下输入指令：target remote localhost:1234远程连接到第一个终端中运行的程序，并使用disassemble进行反汇编：

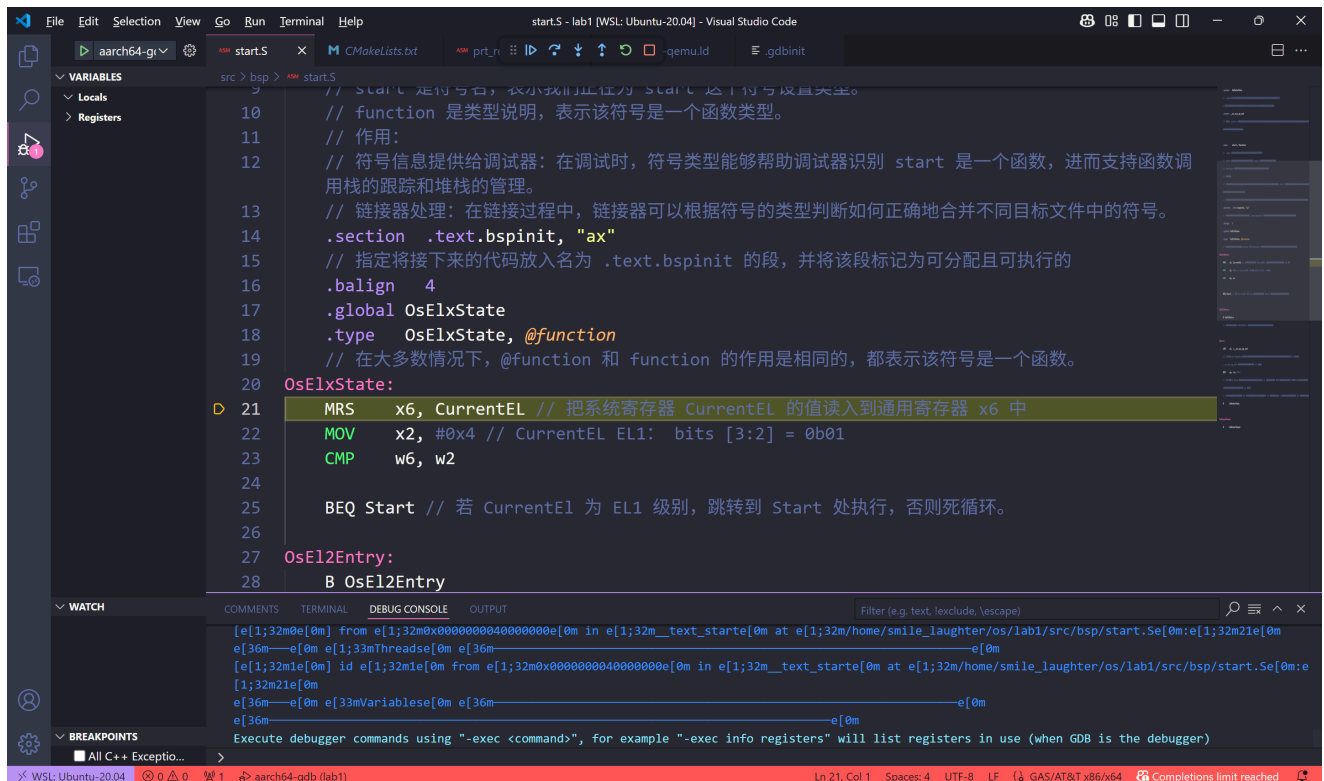
```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
__text_start () at /home/smile_laughter/os/lab1/src/bsp/start.S:21
21      MRS      x6, CurrentEL // 把系统寄存器 CurrentEL 的值读入到通用寄存器 x6 中
(gdb) disas
Dump of assembler code for function __text_start:
=> 0x0000000040000000 <+0>:      mrs      x6, currentel
   0x0000000040000004 <+4>:      mov      x2, #0x4                                // #4
   0x0000000040000008 <+8>:      cmp      w6, w2
   0x000000004000000c <+12>:     b.eq     0x40000014 <Start> // b.none
End of assembler dump.
(gdb)
```

2.将调试集成到vscode中

在终端处运行程序（或者使用自动化脚本），加上-S指令冻结CPU，使程序在入口停止等待调试。

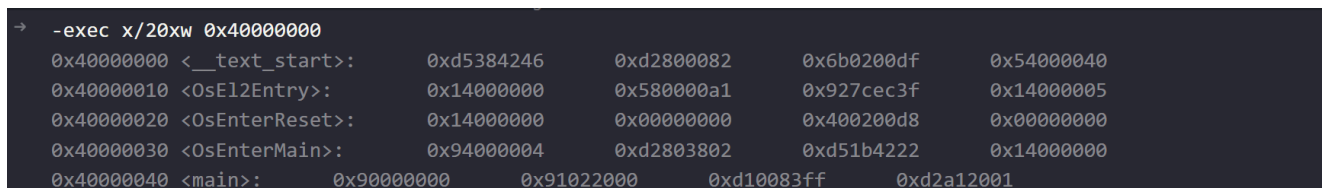
点击左侧的调试选项。

准备工作就绪后，开始调试，可以看到程序顺利在入口处停止：



```
10 // function 是类型说明，表示该符号是一个函数类型。
11 // 作用：
12 // 符号信息提供给调试器：在调试时，符号类型能够帮助调试器识别 start 是一个函数，进而支持函数调用栈的跟踪和堆栈的管理。
13 // 链接器处理：在链接过程中，链接器可以根据符号的类型判断如何正确地合并不同目标文件中的符号。
14 .section .text.bspinit, "ax"
15 // 指定将接下来的代码放入名为 .text.bspinit 的段，并将该段标记为可分配且可执行的
16 .balign 4
17 .global OsElxState
18 .type OsElxState, @function
19 // 在大多数情况下，@function 和 function 的作用是相同的，都表示该符号是一个函数。
20 OsElxState:
21 MRS x6, CurrentEL // 把系统寄存器 CurrentEL 的值读入到通用寄存器 x6 中
22 MOV x2, #0x4 // CurrentEL EL1: bits [3:2] = 0b01
23 CMP w6, w2
24
25 BEQ Start // 若 CurrentEL 为 EL1 级别，跳转到 Start 处执行，否则死循环。
26
27 OsEl2Entry:
28 B OsEl2Entry
```

在调试控制台处输入指令 `-exec x/20xw 0x40000000` 可以查看从地址 `0x40000000` 起始的20个四字节的地址空间中的值，以16进制形式表示：



```
-exec x/20xw 0x40000000
0x40000000 <__text_start>: 0xd5384246 0xd280082 0x6b0200df 0x54000040
0x40000010 <OsEl2Entry>: 0x14000000 0x580000a1 0x927cec3f 0x14000005
0x40000020 <OsEnterReset>: 0x14000000 0x00000000 0x400200d8 0x00000000
0x40000030 <OsEnterMain>: 0x94000004 0xd2803802 0xd51b4222 0x14000000
0x40000040 <main>: 0x90000000 0x91022000 0xd10083ff 0xd2a12001
```

此外，交叉编译工具链中可以执行gdb的所有调试指令，不过要在gdb指令前加上 `-exec`，如：

显示所有寄存器。 `-exec info all-registers`

查看寄存器内容。 `-exec p/x $pc`

修改寄存器内容。 `-exec set $x24 = 0x5` （将寄存器x24的值设置为0x5）

修改指定内存位置的内容。 `-exec set {int}0x40000000 = 0x1` 或者 `-exec set *((int *) 0x40000000) = 0x1`

五、心得体会

- 1.对操作系统的理解更加深入。
- 2.深入理解操作系统内核的底层原理。
- 3.学会对qemu模拟器运行的程序进行调试，对vscode的json配置更加清楚。
- 4.对Ubuntu的使用也更熟悉。