M11212913 HW2 report

在這份作業我遇到了兩個問題

1. 呼叫函式時卡在 ret 沒辦法回到原本呼叫函式的下一行
   解決方法: 原先在終端輸入完個數及元素後就卡死, 使用 RARS 發現卡在 ret. 每次呼叫函式時要返回的地址會被保存在 ra 暫存器中, 由於在層層呼叫的過程中我都沒有先在被呼叫函式中把 ra 保存在 stack 在繼續呼叫其他函式, 才導致後面要 ret 時無法取得正確的 ra 地址, 讓 ra 錯亂, 後面我將每個被呼叫的函式開頭都先把 ra 保存在 stack, 等要 ret 這個函式前取出, 才正確回到呼叫我的下一行.

2. 基址, lo, hi, 這三個要傳遞給函式的參數會在函式內在呼叫其他函式時, 它們的值會錯亂
   解決方法: 這三個參數我分別存在 a0 到 a3 供函式間調用, 但在層層調用中如果 ret 回到呼叫者在下一行之後如果還有要用到 a0 到 a3 那個值已經不是期望輸入的. 跟保存 ra 一樣, 在被呼叫的函式中用 s 系統的暫存器來存放 a 系列的值, 確保呼叫到的函式 ret 時, 能夠歸還原始值, 讓下一行之後會用到 a0 到 a3 的程式能夠輸入期望的值.

以下為每個程式區塊描述

---

sort(addr, count)

```
sort:
    # ==== 保存 ra 用於回到呼叫我的下一行 分配 stack 來儲存 結束時歸還====
    addi    sp, sp, -8        # 保存 ra 分配 stack
    sd      ra, 0(sp)

    mv      t0, a1            # t0 = array size
    addi    t0, t0, -1        # t0 = array size - 1 = hi
    mv      a1, zero          # a1 = 0 -> 給 quicksort 的 lo
    mv      a2, t0            # a2 = array size - 1 -> 給 quicksort 的 hi
    jal     quicksort

    ld      ra, 0(sp)
    addi    sp, sp, 8
    ret
```

執行 sort 函式時, 我先保存地址 ra 到 stack, 用於 sort 函式結束時回到呼叫我的下一行程式碼.

先把 array size(a1) 複製到 t0, 再把 size - 1 複製到 a2 作為輸入 quicksort 的參數 hi, 把 zero 複製到 a1 作為 quicksort 的參數 lo, 加上原本的 a0 基址, 呼叫 quicksort時傳入這三個參數.

quicksort 結束時拿出儲存的地址, 歸還空間, 回到 main 那邊呼叫我的下一行.

quicksort(addr, lo, hi)

```
quicksort:
    # ==== 保存 ra 與分配用到的 s 系列暫存器 stack 結束時復原歸還空間 ====
    addi    sp, sp, -32       # 保存 ra 分配 stack
    sd      ra, 0(sp)
    sd      s0, 8(sp)
    sd      s1, 16(sp)
    sd      s2, 24(sp)

    # ==== 避免在調用函式 a0 到 a3 被覆蓋 先移到 callee 的 s 系列暫存器 ====
    mv      s0, a0            # s0 = 基址
    mv      s1, a1            # s1 = lo
    mv      s2, a2            # s2 = hi
```

執行 quicksort 時, 保存地址 ra, 用於回到呼叫我的下一行程式碼, 並且把 a0 到 a3 用於存放基址與 lo, hi的暫存器放到被呼叫者使用的 s 系列暫存器保存, 放到 stack, 這樣每次遞迴或是呼叫其他函式都能避免因為覆蓋 a0 到 a3, 導致我後面要傳遞的參數是錯的, 函式結束時歸還.

```
    # ==== if lo < 0 || hi < 0 || lo >= hi goto quicksort_exit ====
    blt     s1, zero, quicksort_exit
    blt     s2, zero, quicksort_exit
    bge     s1, s2, quicksort_exit
```

如果我的 lo < 0 或 hi < 0, 或 lo >= hi 的話就離開當前層的 quicksort, 回到呼叫我的下一行.

```
    # ==== 呼叫 partition 取 j ====
    mv      a0, s0
    mv      a1, s1
    mv      a2, s2
    jal     partition
    mv      t0, a4              # t0 = j
```

把每層遞迴的基址, lo, hi, 複製到 a 系列暫存器用於輸入 partition 的參數, 最後把 partition 的回傳值 a4 複製到 t0 暫存, 給後續遞迴用.

```
    # ==== 遞迴左半 ====
    mv      a0, s0            # 基址
    mv      a1, s1            # lo
    addi    a2, t0, 0        # hi = j
    jal     quicksort
```

把基址, lo, hi = j(partition的回傳值), 複製給 a 系列用於傳遞給左半遞迴的參數, 左半結束時會跳到 quicksort_exit, 歸還空間, 回到呼叫我的下一行.

```
    # ==== 遞迴右半 ====
    mv      a0, s0          # 基址
    addi    a1, t0, 1       # lo = j + 1
    mv      a2, s2          # hi
    jal     quicksort
```

空間歸還後, 把呼叫左半 quicksort 前的基址, lo(把回傳值 j + 1), hi 複製到 a 系列給右半遞迴用, 結束時會跳到 quicksort_exit, 回到 sort 函式呼叫我的下一行.

```
quicksort_exit:
    ld      ra, 0(sp)
    ld      s0, 8(sp)
    ld      s1, 16(sp)
    ld      s2, 24(sp)
    addi    sp, sp, 32
    ret
```

空間歸還後, 回到呼叫我的下一行

---

partition(addr, lo, hi)

```
partition:
    # ==== 保存 ra 與分配用到的 s 系列暫存器 stack 結束時復原歸還空間 ====
    addi    sp, sp, -40
    sd      ra, 0(sp)
    sd      s0, 8(sp)
    sd      s1, 16(sp)
    sd      s2, 24(sp)
    sd      s3, 32(sp)

    # ==== 避免在調用函式 a0 到 a3 被覆蓋 先移到 callee 的 s 系列暫存器 ====
    mv      s0, a0          # s0 = 基址
    mv      s1, a1          # s1 = lo
    mv      s2, a2          # s2 = hi
```

一樣先保存 ra 用於返回呼叫我的下一行, a0 到 a3 用於存放基址與lo, hi的暫存器放到被呼叫者使用的 s 系列暫存器保存, 避免後續調用錯亂.

```
    # ==== pivot = array[lo] ====
    slli    t0, s1, 3
```

```
    add     t0, s0, t0
    ld      t1, 0(t0)
```

由於 partition 只須回傳 index j, 所以接下來除了 j, 其他操作都是在不需在函式間調用的臨時暫存器 t 系列上面. 將 s1(lo) * 8 放到 t0, t0 在與基址相加得到 array[lo] 的地址, 最後從 t0 指向的地址讀取 data 後放到 t1 (pivot = array[io])

```
    # ==== 初始化 index i(left), j(right) ====
    addi    t2, s1, -1        # t2 = i = lo - 1
    addi    t3, s2, 1         # t3 = j = hi + 1
```

初始化 parition 用 index, i = lo -1 放到 t2, 與 j = hi + 1 放到 t3.

```
loop_forever:
```

loop_forever 此標籤用來跳回來實現無限次執行從這到 j loop_forever 之間的程式碼, 直到 i >= j 才會跳出.

```
do_i_add_one:
    addi    t2, t2, 1         # i = i + 1
    slli    t4, t2, 3         # t4 = i * 8
    add     t4, s0, t4        # t4 = 基址 + i * 8
    ld      t5, 0(t4)         # t5 = array[i]
    blt     t5, t1, do_i_add_one  # if array[i] < pivot 繼續 do_i_add_one
```

t2 = i + 1, 在把 t2 的值 x 8 存到 t4, t4 在與基址相加得到 array[i], 把 t4 指向的地址讀取讀取值後放到 t5, 在與 pivot(t1) 比較, 如果 t5 < pivot(t1), 繼續 t2 = t2 + 1, 實現 do i = i + 1, while array[i] < pivot.

```
do_j_sub_one:
    addi    t3, t3, -1        # j = j - 1
    slli    t6, t3, 3         # t6 = j * 8
    add     t6, s0, t6        # t6 = 基址 + j * 8
    ld      t5, 0(t6)         # s3 = array[j]
    bgt     t5, t1, do_j_sub_one  # if array[j] > pivot 繼續 do_j_sub_one
```

t3 = j - 1, 在把 t3 的值 x 8 存到 t6, t6 在與基址相加得到 array[j], 把 t6 指向的地址讀取讀取 data 後放到 t5, 這邊重複用 t5 是因為我在函式中不需要保存 t5 的值, 我只需要有個暫存能存當前值來做比較. 後面一樣將 t6 指向的地址讀取值放到 t5, 與 pivot(t1) 比較, 如果 t5 > pivot(t1), 繼續 t3 = t3 - 1, 實現 do j = j - 1, while array[j] > pivot.

```
    # ==== if i >= j return j ====
    bge     t2, t3, partition_exit
```

如果 index i >= index j, 跳到 partition_exit return index j.

```
    # ==== swap array[i] 和 array[j] ====
    mv      a5, t4           # a0 = &array[i]
    mv      a6, t6           # a1 = &array[j]
    jal     swap             # swap array[i] with array[j]
```

t4 = 當前要 swap 的 array[i] 地址, t6 = 當前 array[j] 地址, 兩者個別存到暫存器 a5, a6 用於跟 swap 函式間傳遞的參數.

```
    # === 無限迴圈 ===
    j       loop_forever

partition_exit:
    mv      a4, t3          # a4 = j

    ld      ra, 0(sp)
    ld      s0, 8(sp)
    ld      s1, 16(sp)
    ld      s2, 24(sp)
    ld      s3, 32(sp)
    addi    sp, sp, 40
    ret
```

t3(index j) 複製到 a4, 用於出函式之後傳遞參數(遞迴的邊界), 歸還空間, 回到呼叫 partition 的下一行.

```
# === partition 的 swap(a, b), swap array[i] with array[j] ===
swap:
    ld      t0, 0(a5)
    ld      t1, 0(a6)
    sd      t1, 0(a5)
    sd      t0, 0(a6)
    ret
```

ld 將 a5 指向地址讀取值後放到 t0, a6 指向地址讀取值後放到 t1,
sd 把 t1 值寫進 a5 地址, t0 值寫進 a6 地址, 實現兩邊位址的值互換
