컴퓨터 구조

PA1 Write-up

2013-11826 임주경

1. Introduction

GCD, Fibonacci, Maze 세 개의 C source code들을 64-bit RISC-V 어셈블리어로 작성한다.

2. Implementation

- GCD

```
gcd:
        #------Your code starts here------
        #LHS: a0, RHS: a1
        addi sp, sp, -24
        sd x5, 16(sp)
        sd x6, 8(sp)
        sd x7, 0(sp)
        add x5, a0, x0
        add x6, a1, x0
        bne x5, x6, loop
        add x10, x5, x0
        ld x7, 0(sp)
        ld x6, 8(sp)
        ld x5, 16(sp)
        addi sp, sp, 24
        jalr x0, 0(x1)
loop:
        beq x5, x6, exit
        bge x6, x5, else
        sub x5, x5, x6
        beq x0, x0, loop
else:
        add x7, x5, x0
        add x5, x6, x0
        add x6, x7, x0
        beq x0, x0, loop
exit:
        add x10, x5, x0
        ld x7, 0(sp)
        ld x6, 8(sp)
        ld x5, 16(sp)
        addi sp, sp, 24
        jalr x0, 0(x1)
        #Load return value to reg a0
        #------Your code ends here------
```

LHS는 a0, RHS는 a1에 할당되어있다. 임시레지스터로 x5, x6, x7을 사용하기 위해 스택 메모리에 24 byte를 만들어 초기값을 저장한다. LHS와 RHS값을 x5, x6에 저장하고 둘의 값이 다르면 branch를 사용해 loop로 이동한다. RHS값이 LHS값 보다 클 경우 else branch로 이동해서 x7 레지스터를 사용해 둘의 위치를 swap해주고 다시 loop로 이동하며, GCD의 핵심 알고리즘은 loop에서 진행된다. LHS, RHS값이 같으면 exit branch로 이동해서 a0에 최종 x5값을 저장하고, 임시레지스터를 초

기값으로 불러서 복구한 뒤, 함수가 return된다.

- Fibonacci

```
fibonacci:
        #------Your code starts here------
        addi sp, sp, -40
        sd x5, 32(sp)
        sd x6, 24(sp)
        sd x7, 16(sp)
        sd x28, 8(sp)
        sd a0, 0(sp)
        addi x5, x0, 1
        sd x5, 0(a0)
        add x7, x5, x5
        blt a1, x7, exit
        addi a0, a0, 8
        sd x5, 0(a0)
        addi a0, a0, -8
loop:
        bge x7, a1, exit
        ld x5, 0(a0)
        addi a0, a0, 8
        ld x6, 0(a0)
        addi a0, a0, 8
        add x28, x5, x6
        sd x28, 0(a0)
        addi a0, a0, -8
        addi x7, x7, 1
        beq x0, x0, loop
exit:
        ld a0, 0(sp)
        ld x28, 8(sp)
        ld x7, 16(sp)
        ld x6, 24(sp)
        ld x5, 32(sp)
        addi sp, sp, 40
        jalr x0, 0(x1)
        #Load return value to reg a0
        #------Your code ends here------
```

피보나치 배열의 시작 주소는 a0, 배열의 길이는 a1에 저장되어 있다.

임시레지스터로 x5, x6, x7, x28 네 개를 사용하며, 스택에 40 byte의 공간을 만들어 네 개의 초기값과 a0의 초기값을 저장한다. x5를 사용해서 첫번째 피보나치 수 1을 배열에 저장하며, n이 1일 경우 exit branch로 이동해서 함수를 마친다. 그렇지 않을 경우, 두번째 피보나치 수 1을 배열에 저장하며, loop로 이동한다. loop에서는 x7을 알고리즘의 for 구문의 내부 변수 i로 사용한다. Branch의 매 시작마다 i가 n에 도달할 경우 exit branch로 이동한다. 그렇지 않을 경우, x5에 F(i-2)를 x6에 F(i-1)을 저장하고, x28에 x5 + x6 즉, F(i)를 저장한 뒤, sd명령어를 통해 배열의 i번째에 저장한다. 위의 과정을 반복하며, 배열에 n개의 수를 채운 뒤, exit로 이동해서 a0, x5, x6, x7, x28을 초기값으로 복구하고, 함수를 종료한다.

- Maze

함수는 solve_maze와 your_func 두개를 사용한다. 한 번 실행되는 solve_maze는 변수로 사용하기 위한 x5-x7, x28-x30과 jump 명령어에 사용될 주소 저장 레지스터 x1의 초기값을 스택에 저장한다. 각 레지스터에 저장된 값은 다음과 같다. a1 : width, a2: height, x5 : maze배열의 시작주소, x6 : x_pos, x7 : y_pos, x28 : depth, x29 : prev_trav, x30 : MAX_DEPTH.

```
solve_maze:
        #------Your code starts here------
        #maze: a0, width: a1, height: a2
        addi sp, sp, -56
        sd x1, 48(sp)
        sd x30, 40(sp)
        sd x29, 32(sp)
        sd x28, 24(sp)
        sd x7, 16(sp)
        sd x6, 8(sp)
        sd x5, 0(sp)
        add x5, x0, a0
        add x6, x0, x0
        add x7, x0, x0
        add x28, x0, x0
        addi x29, x0, 2
        addi x30, x0, 20
        jal x1, your_funct
        ld x5, 0(sp)
        ld x6, 8(sp)
        ld x7, 16(sp)
        ld x28, 24(sp)
        ld x29, 32(sp)
        ld x30, 40(sp)
        ld x1, 48(sp)
        addi sp, sp, 56
        jalr x0, 0(x1)
        #Load return value to reg a0
        #------Your code ends here------
```

a1, a2는 건드리지 않으며, 처음 실행될 traverse(0, 0, 0, T_RIGHT)에 맞게 변수값을 저장한다. MAX_DEPTH값 20은 x30에 저장하며, 건드리지 않는다. 점프 명령어를 이용해 your_funct로 이동한다.

```
your_funct:
        addi sp, sp, -40
        sd a3, 32(sp)
        sd a4, 24(sp)
        sd a5, 16(sp)
        sd x31, 8(sp)
        sd x1, 0(sp)
        blt x28, x30, xpos
        addi a0, x0, -1
        beq x0, x0, exit
exit:
        ld x1, 0(sp)
        ld x31, 8(sp)
        ld a5, 16(sp)
        ld a4, 24(sp)
        ld a3, 32(sp)
        addi sp, sp, 40
        jalr x0, 0(x1)
xpos:
        bge x6, x0, ypos
        addi a0, x0, -1
        beq x0, x0, exit
ypos:
        bge x7, x0, width
        addi a0, x0, -1
        beq x0, x0, exit
width:
        blt x6, a1, height
        addi a0, x0, -1
        beq x0, x0, exit
height:
        blt x7, a2, dead_end
        addi a0, x0, -1
        beq x0, x0, exit
```

```
dead_end:
        mul a3, x7, a1
        add a3, a3, x6
        slli a3, a3, 3
        add x5, x5, a3
        ld x31, 0(x5)
        sub x5, x5, a3
        add a3, x0, x0
        beq x31, x0, success_x
        addi a0, x0, -1
        beq x0, x0, exit
success_x:
        addi x31, a1, -1
        bne x6, x31, min
        beq x0, x0, success_y
success_y:
        addi x31, a2, -1
        bne x7, x31, min
        add a0, x0, x28
        beq x0, x0, exit
min:
        addi x31, x0, -1
        add a4, x0, x29
        beq x0, x0, traverse_up
```

```
traverse_up:
        addi a3, x0, 3
        beq x29, a3, traverse_left
        addi x7, x7, -1
        addi x28, x28, 1
        add x29, x0, x0
        jal x1, your_funct
        add x31, x0, a0
        addi x7, x7, 1
        addi x28, x28, -1
        add x29, x0, a4
        beq x0, x0, traverse_left
traverse_left:
        addi a3, x0, 2
        beq x29, a3, traverse_right
        addi x6, x6, -1
        addi x28, x28, 1
        addi x29, x0, 1
        jal x1, your_funct
        add a5, x0, a0
        addi x6, x6, 1
        addi x28, x28, -1
        add x29, x0, a4
        blt a5, x0, traverse_right
        blt x31, x0, left_result
        blt a5, x31, left_result
        beq x0, x0, traverse_right
left_result:
        add x31, x0, a5
        beq x0, x0, traverse_right
```

```
traverse_right:
        addi a3, x0, 1
        beq x29, a3, traverse_down
        addi x6, x6, 1
        addi x28, x28, 1
        addi x29, x0, 2
        jal x1, your_funct
        add a5, x0, a0
        addi x6, x6, -1
        addi x28, x28, -1
        add x29, x0, a4
        blt a5, x0, traverse_down
        blt x31, x0, right_result
        blt a5, x31, right_result
right_result:
        add     x31, x0, a5
        beq x0, x0, traverse_down
traverse_down:
        addi a3, x0, 0
        beq x29, a3, min_exit
        addi x7, x7, 1
        addi x28, x28, 1
        addi x29, x0, 3
        jal x1, your_funct
        add a5, x0, a0
        addi x7, x7, -1
        addi x28, x28, -1
        add x29, x0, a4
        blt a5, x0, min_exit
        blt x31, x0, down_result
        blt a5, x31, down_result
        beq x0, x0, min_exit
down_result:
        add x31, x0, a5
        beq x0, x0, min_exit
```

```
min_exit:
        add a0, x0, x31
        ld x1, 0(sp)
        ld x31, 8(sp)
        ld a5, 16(sp)
        ld a4, 24(sp)
        ld a3, 32(sp)
        addi sp, sp, 40
        jalr x0, 0(x1)
        #Ret
        jr      ra
        .size   your_funct, .-your_funct
        #------Your code ends here
```

your_funct 내부에서 사용하기 위해 레지스터 a3, a4, a5, x31, x1의 초기값을 스택에 저장한다. Maze.c의 초기 if구문은 각각의 조건에 따라서 branch로 만들었다. 소스코드 상의 순차대로 진행되며, return 해야 할 경우, exit branch로 이동해서 해당하는 return 값을 a0에 저장된 채 함수가 종료된다. Traverse에 들어가기 전, min branch에서 min = -1을 실행하고, Traverse up 부터 순차적으로 branch를 이용해 명령을 수행한다. 수행과정에서 재귀적 traverse 수행은 jump 명렁어를 이용해 your_funct로 이동해서 얻은 return 값을 min값이나 result값으로 저장한다. Traverse up을 제외한 나머지에서 result값을 min값에 저장해야 하는 경우의 if 구문은 따로 branch를 만들어서 수행하였다. 최종적으로 min_exit branch에서 min값을 return하며, 각 레지스터를 초기값으로 복구한 뒤, 함수를 종료한다.

3. Result

GCD, Fibonacci, Maze에서 모두 정상적인 output을 확인하였다.