

# Computer Architecture

## Hw1

Mechanical Engineering  
2019-15838 주기영

### # 1. digitsum program

x10: lhs 값 저장  
x11: rhs 값 저장

```
14 digitsum:
15
16     add    x7, x10, x0      # lhs in x7
17     add    x28, x11, x0     # rhs in x28
18     li     x5, 0           # x5: sum, sum=0
19     li     x29, 10          # x29: constant(10)
20
21
22 loop_lhs:                  # x29(t4)에 저장해줌
23     lhs의 beq    x7, x0, loop_rhs # if lhs==0, break
24     자리함 remu   x6, x7, x29     # sum += lhs%10;
25     계산 Label add    x5, x5, x6
26     Label divu    x7, x7, x29     # lhs/=10
27     beq    x0, x0, loop_lhs # always to loop_lhs
28
29
30 loop_rhs:
31     rhs의 beq    x28, x0, done    # if rhs==0, break
32     자리함 remu   x6, x28, x29    # sum += rhs%10;
33     계산 Label add    x5, x5, x6
34     Label divu    x28, x28, x29   # rhs/=10
35     beq    x0, x0, loop_rhs # always to loop_rhs
36
37
38
39 done:
40     addi   x10, x5, 0          # store sum in x10(return value)
41
```

line 16~17

만약 x10(a0), x11(a1)에 저장된 lhs, rhs 값을  
x7(t2), x28(t3)에 옮겨두었음.

line 18

최종적인 digit sum을 저장하기 위해 x5(t0)  
register에 저장해두었음.

x5(t0): sum 변수 저장  
x6(t1): 자리수를 임시로 저장하는 Register  
x7(t2): lhs의 자리함 하기 위해 최초 저장  
x28(t3): rhs의 자리함 구하기 위하여 최초 저장  
x29(t4): 상수 10을 저장하는 역할

line 24: while문에서 lhs 값이 0이 된다면 while 문을 빠져나가야 함, 이를 위해서

beq 명령어를 이용하여 lhs가 저장되어 있는 x7과 항상 0이 저장되는 x0를 비교하여  
같은 때는 while 문을 빠져나와서 loop\_rhs를 실행하도록 설계

line 25: \* remu instruction

REMAinder Unsigned  $R[rd] = (R[rs1] \% R[rs2])$

→ remu를 이용하여 lhs를 10으로 나눈 나머지를 x6(t1)에 저장.

line 26: line 25에 저장해둔 나머지 값을 sum 변수를 저장하는 역할을 하는 register x5에 저장한다.

line 27: \* divu instruction

DIVide Unsigned  $R[rd] = (R[rs1] / R[rs2])$

→ divu를 이용하여 lhs 값에 lhs를 10으로 나눈 몫을 대신 저장한다.

line 28: while문의 끝에서 다시 반복을 위하여 항상 참이 되는 조건문을 이용하여 다시 loop\_lhs로 되돌아간다.

line 32-36: rhs의 자리수들을 sum에 더하기 위한 반복문을 위한 부분으로 Logic은 loop\_lhs와 완전히 같다.

line 40: sum이 저장된 x5 register의 값을 x10(a0)에 저장한다.



#2. Fibonacci (recursive를 이용하여 구현)

재귀함수에서 x와 x10이 계속하여 바뀌기 때문에 stack pointer를 이용해야 한다.

Fibonacci 함수 Label

```
10 fibonacci:
11
12     addi    sp, sp, -8
13     sw      x1, 4(sp)
14     sw      x10, 0(sp)
15
16     addi    x5, x10, -3
17     bge     x5, x0, L1
18
19     addi    x10, x0, 1
20     addi    sp, sp, 8
21     jalr    x0, 0(x1)
22
23 L1: 재귀함수 구현 Label
24     addi    x10, x10, -1
25     jal     x1, fibonacci
26
27     addi    x6, x10, 0
28     lw      x10, 0(sp)
29
30     addi    sp, sp, -4
31     sw      x6, 0(sp)
32
33     addi    x10, x10, -2
34     jal     x1, fibonacci
35
36     lw      x6, 0(sp)
37     lw      x1, 8(sp)
38
39     addi    sp, sp, 12
40
41     add     x10, x10, x6
```

line 12

fibonacci Label로 들어오면 먼저 return address와 argument를 저장하기 위해서 stack의 두 칸을 확보해야 한다. Stack은 아래로 발라주기 때문에 sp에서 8을 빼서 다시 저장하면 return address x와 fibonacci(n)을 호출할 때의 n을 저장할 수 있다.

line 13-14

현재 상태에서 돌아가야 할 곳 x와 fibonacci를 호출할 방식에 저장된 n 값을 stack에 저장해준다.

line 16, 17

※ bge instruction  $\neg(R[rs1] \geq R[rs2])$   
Branch Greater than or equal  $PC = PC + [Imm, 16'0]$

먼저 x5 register에 n-3을 저장하고, x0에 저장된 0과 비교하여  $n-3 \geq 0$ 이면 L1 label로 이동할 수 있도록 한다.

line 19-21

만약에  $n \leq 2$ 인 경우에는 바로 value 1을 반환할 수 있도록 하는 code  
먼저 처음에 line 12-14에서 확보한 stack 영역을 반납하고, return value를 저장하는 x10 register에 1 값을 저장한다.  
이 경우에 line 12-14에서의 x1/x10과 값이 변하지 않았으므로, stack에서 x와 x10을 다시 꺼낼 필요가 없다. 마지막으로 return address를 이용하여 호출 프로그램으로 복귀한다.

line 24-25

$n \geq 3$ 인 경우에 재귀함수 구현을 위한 Label의 맨 처음 부분  
인식 register인 x10에 n-1을 먼저 저장한다. 그 후에 x10에 n-1을 저장하고, 다음 명령어의 주소를 x1에 저장하고 fibonacci(n-1)을 호출하는 것이다.

line 27

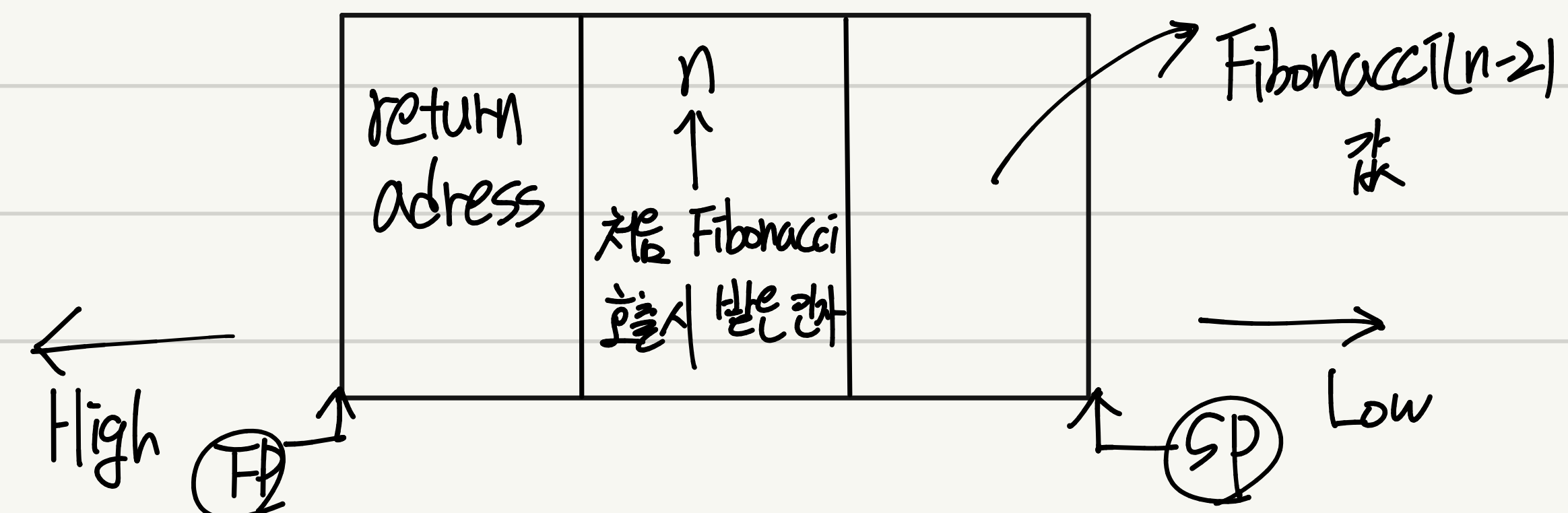
line 27에서는 fibonacci(n-1) 값을 x10에 저장해둔 상태이다. fibonacci(n-2)를 호출하기 위해 stack에 저장된 n을 load해야 한다. 따라서 원래의 fibonacci(n-1) 값을 임시로 x6에 저장해둔다.

line 28

Stack에 저장된 n 값을 x10에 다시 load하는 부분이다.

line 30, 31

x6는 임시 register라서 fibonacci(n-2)를 호출하기 전후로 값이 바뀐다. 따라서 이를 stack에 저장하기 위한 부분이다. line 31이 끝난 직후 stack 상태는 다음과 같다.





**line 33,34** x10에 원래 받았던 인자 n을 다시 받고, addi instruction을 이용하여  $Fibonacci(n-2)$ 를 호출한다. line 25와 마찬가지로 x10에  $n-2$  값을 넣고, 다시 recursive 알고리즘을 따라간다.

**line 36**  $Fibonacci(n-1)$ 을 저장해둔 stack 위치에서 x6 register에 값을 저장한다.  
이것을 한 직후에  $\left[ \begin{array}{l} x6: Fibonacci(n-1) \text{ 값 저장} \\ x10: Fibonacci(n-2) \text{ 값 저장} \end{array} \right)$  된 상태가 되었다.

**line 37**  $Fibonacci(n-1)$ ,  $Fibonacci(n-2)$ 를 모두 저장되었으므로, 이제  $Fibonacci(n)$ 을 호출 하였던 호출 프로그램으로 복귀해야 한다. 따라서 stack에 저장된 return address를 다시 x1 register에 저장해준다.

**line 38** 세 개의 stack pointer를 이용하였으므로 이를 반납한다.

**line 39** 반환값을 저장하는 x10 register에  $Fibonacci(n)$ 을 반환하기 위해서 현재  $Fibonacci(n-2)$ 가 저장된 x6 register의 값과  $Fibonacci(n-1)$ 이 저장된 x10 register의 값을 더해서 x10에 저장된다.

### #3 Tree (Tree 순회하며 합 구하기)

```
10 tree:
11
12     li    a7, 12          # constant:12
13     li    s2, 0           # sum
14     li    s3, 1           # curr
15     li    s4, 0           # head
16
17     lw     a3, 0(a0)       # memberwise copy
18     sw     a3, 0(a1)
19     lw     a3, 4(a0)
20     sw     a3, 4(a1)
21     lw     a4, 8(a0)
22     sw     a4, 8(a1)
23     li     a5, 1
24
25     beq    zero, zero, .L2 # go to L2
26 .L5:
27
28     addi   a4, s4, 0       # a4 : store head
29     addi   s4, s4, 1       # head+=1
30
31     mul    a4, a4, a7      # address calculate
32                                     12xhead 값 계산
33     add    s5, a1, a4      # s5: &queue[head]
34
35     lw     a5, 4(s5)       # a5 : node->left
36
```

s5: while문 내에서 node를 저장하는 register

**line 12** a7: address 계산에 위하여 상수 12 저장

**line 13-15** s2, s3, s4 register를 이용하면서 tree 함수에서 사용할 sum, curr, head value를 저장한다. 각자 초기값을 li instruction을 이용하여 저장해준다.

**line 17-22**  $queue[0] = *(root)$ 를 구현하는 것 queue의 첫 번째에 root node를 넣어야 한다. 유의할 점은 구조체의 member들을 memberwise copy해야 되는 것이다. a0은 root의 주소, a1은 queue의 첫 번째 원소의 주소를 가르키므로, line 17-22와 같이 각각의 요소를 register에 load/save를 반복하면서

**line 23** L2로 이동 (L2는 while문의 시작 부분이며, 조건을 판별을 포함, 첫 번째 존재

**line 28-29** register a4에 head value를 저장하고, head value에 1을 더해준다.

**line 31** queue에서  $queue[head]$ 의 주소를 찾기 위해서 상수 12를 이용하여  $12 \times head$  값을 계산한다.

**line 33** s5 register는 while문에서 계속하면서 현재 while문의 node를 가르키게 된다.

**line 35**  $node \rightarrow left$ 를 a5 register에 저장.



```

37 #####
38 beq    a5,zero,.L3    # if a5==nullPtr, go to .L3
39
40 lw     a4,4(s5)        # a4 : store node->left
41
42 addi   a3,s3,0         # a3 : store curr
43 addi   s3,s3,1         # curr+=1
44
45 mul    a3, a3, a7      # adress calculate
46
47 add    a5,a1,a3        # a5 : &queue[curr]
48
49 lw     a3,0(a4)        # memberwise copy
50 sw     a3,0(a5)
51 lw     a3,4(a4)
52 sw     a3,4(a5)
53 lw     a4,8(a4)
54 sw     a4,8(a5)
55 .L3:
56
57 lw     a5,8(s5)        # Have same logic at .L5 below #####
58 beq    a5,zero,.L4
59
60
61 lw     a4,8(s5)
62
63 addi   a3,s3,0
64 addi   s3,s3,1
65
66 mul    a3, a3, a7
67
68 add    a5,a1,a3
69
70 lw     a3,0(a4)
71 sw     a3,0(a5)
72 lw     a3,4(a4)
73 sw     a3,4(a5)
74 lw     a4,8(a4)
75 sw     a4,8(a5)

```

.L2의 # 밑의 부분과  
원천이 동일한 Logic을 갖는다.

```

76 .L4:
77
78 lw     a5,0(s5)        # node->value
79 add    s2, s2, a5      # sum+=node->value
80
81 .L2:
82
83 bne    s4,s3,.L5      # if head!=curr go to .L5
84 addi   a0,s2,0         # store curr sum value in a0 register
85

```

line 38) node->left가 null pointer이면 if 문을 빠져나온다.

line 40) node->left를 a4 register에 저장한다

line 42-47) line 28-33과 마찬가지로 현재의 curr value를 이용하여 queue[curr]의 주소를 a5 register에 저장하고, curr value에 1을 더한다.

line 49-50) node->left의 주소가 저장된 a4 register와 queue[curr]의 주소가 저장된 a5 register를 이용하여 Memberwise copy를 진행한다.

line 78) node의 주소가 저장된 s5 register를 이용하여 node->value를 a5에 load 한다.

line 79)  $sum \pm node \rightarrow value$

line 83) 처음 while 문에 들어올 때, head != curr이면 while 문을 수행하도록 한다.

line 84) return value를 저장하는 a0 register에 현재의 sum value를 저장한다.

3번 문제의 경우 원래 register 1,2와 마찬가지로 x0...x31로 짜리가 계속하여 segfault 오류가 발생하여  
혹시하여 register의 이름으로 짜게 되었습니다. (물론 포현의 문제는 아니었습니다.)