CS2002 Tutor:Marwan Fayed 170025298

W04 – Practical Report

1. Stack frames

1> Data analysis

Part 2:

Since the result is so long, in order to make the result more clear, I re-formatted output as results shows below:

```
pc2-033-l:~/Documents/CS2002/W04Practical/Files hl74$ ./main-stack
 Stack reserved for local variables are from 140721149507744 to 140721149507824
 Value of arguments:
 "val" : 1234 address : 140721149507816
 "depth" : 1 address : 140721149507812
 "*end" : 0x7ffc3220f958 address : 140721149507800
----------------------
 Value for local variables are:
 "quot" : 123 address : 140721149507792
 "mod" : 4 address : 140721149507784
 "*start" : 0x7ffc3220f8f0 address : 140721149507776
 "len" : 0 address : 140721149507772
==================

 Stack reserved for local variables are from 140721149507664 to 140721149507744
 Value of arguments:
 "val" : 123 address : 140721149507736
 "depth" : 2 address : 140721149507732
 "*end" : 0x7ffc3220f958 address : 140721149507720
----------------------
 Value for local variables are:
 "quot" : 12 address : 140721149507712
 "mod" : 3 address : 140721149507704
 "*start" : (nil) address : 140721149507696
 "len" : 0 address : 140721149507692
==================

 Stack reserved for local variables are from 140721149507584 to 140721149507664
 Value of arguments:
 "val" : 12 address : 140721149507656
 "depth" : 3 address : 140721149507652
 "*end" : 0x7ffc3220f958 address : 140721149507640
----------------------
 Value for local variables are:
 "quot" : 1 address : 140721149507632
 "mod" : 2 address : 140721149507624
 "*start" : (nil) address : 140721149507616
 "len" : 0 address : 140721149507612
==================

 Stack reserved for local variables are from 140721149507504 to 140721149507584
 Value of arguments:
 "val" : 1 address : 140721149507576
 "depth" : 4 address : 140721149507572
 "*end" : 0x7ffc3220f958 address : 140721149507560
----------------------
 Value for local variables are:
 "quot" : 0 address : 140721149507552
 "mod" : 1 address : 140721149507544
 "*start" : 0x7be260 address : 140721149507536
 "len" : 6 address : 140721149507532
==================
```

Since this is a recursive function, so the reading order should be from down to up, or to say from inside to outside. So this report will analyse the data that being outputted from depth 4 to depth 1.

By calculating absolute addresses for each %rbp, such as for basic recursion, by subtracting addresses 140721149507744-140721149507824 we get 80, where this number is always 80 when subtracting 140721149507664-140721149507744, which is the second recursion. Where we can find that there are 16 bytes offset by subtracting 80 by 16.

In addition to check whether the result suits to comments in part 1, the easiest way of doing that is to check single value first.

Arguments:
"val" is in stack space %rbp – 8, Compare with what I have commented in "itos-commented.s", operations on val is:

C	Assembler
val / 10	cqto idivq %rcx
val % 10	cqto idivq %rcx callq lab
compare val with 0	Cmpq $0, %rcx cmpq $0, -8(%rbp)

Then check the value above, we gets value 1, 12, 123, 1234 by several recursion, comparing with result "val" checked.

-12(%rbp) refers to the argument "depth", calculate the depth of recursion. The operation on it is to add 1 in recursion function

`start = itos_recur(quot, depth + 1, end);`	movl -12(%rbp), %eax addl $1 %eax callq itos_recur

Then check the value above, we will see depth goes from 4 to 1 which shows the depth of recursion from inside to outside. Then depth was checked.

-24(%rbp) refers to the pointer to pointer of end address, which points the address of address of start. Which can be checked with its address never being changed.

Local variables:
-32(%rbp) refers to "quot" : In C program quot refers to: quot = val / 10, which represented in assembly code is as:

movq %rax -32(%rbp)

Which copy the operated value of %rax (see "val") to quot.
Another operation on quot in C program is compare quot with 0 and if not equal to 0, start recursion, which represented in assembly code is as

cmpq $0 -32(%rbp)

```
je .LBB1_2
```

Where .LBB1_2 refers to “else:” part. Since quot will be used as recursive argument as val, the assembly code register quot to %rdi.

-40(%rbp) refers to “mod”, which will print the non-decimal part of the division. It represent in C program is same as the division of quot, but it calls function “labs”. Next operation on mod in assembly code is

```
Movq -40(%rbp), %rax
addq $48 %rax
```

Which adds an char ‘0’ + mod to *(*end)++ %rax by reading from c program (Confusion 1)
Values of quot and mod can be checked through output of program, which are obviously passed.

-48(%rbp) refers to pointer “start”. From assembly code in >.LBB1_5, related code on start is:

```
movq  -24(%rbp), %rax # %rax = *end
movq  (%rax), %rdx # %rdx = %rax
movq  %rdx, %rsi # %rsi = %rdx
addq  $1, %rsi # %rsi += 1
movq  %rsi, (%rax) # %rax = %rsi
movb  %cl, (%rdx) # %rdx = %cl
movq  -48(%rbp), %rax # %rax = *start
```

From comments afterwards, *start = %rsi = %rdx += 1 = *end += 1. Since only when the program cannot be a recursion start can points to malloc(len)

-52(%rbp) refers to “len” only available in else bracket, which only appears in the deepest recursion, as the values in the image are always 0 because len has not accessed.
Operations on len is add depth with 3 or 2 and malloc it, in assembler code it represents as:

```
movl  $3, %edx # %edx = 3
movl  $2, %esi # %esi = 2
cmovll  %edx, %esi # if compare gets 0 then %esi = %edx
addl  %esi, %eax # %eax += %esi
movl  %eax, -52(%rbp) # local variable len = %eax
movslq  -52(%rbp), %rdi # %rdi = len
callq  malloc # call function "malloc"
```

2> confusion:
Several confusion available during reading and commenting through the assembly code and c file:
In itos0, line 108, addq added 48 to %rax
In itos0, line 109, where values of %al come from and what are stack frame that %al and %cl located?
In itos_stack.c, line 22, what is mean by *(*end)++ = ‘-’, especially what is mean by “-”
In itos_stack.c, line 26, why the end pointer to pointer = ‘0’ + mod
In itos_stack.c, line 41, what is mean by ‘\0’
In stack.c, line 8, what this condition holds?

2. Division by invariant integers using multiplication
Part 3:
<1>
Since ceil($2^{65}/10$) * n/2^{65} – floor(n/10) = (ceil($2^{65}/10$) - $2^{65}/10$)*n/2^{65}+(n/10-floor(n/10))

$2^{65} \equiv 2^1 \equiv 2 (\text{mod}10)$, hence $\text{ceil}(2^{65} / 10) - 2^{65}/10 = 8/10$.
n/10-floor(n/10) can be up to 9/10 for $n \equiv 9(\text{mod}10)$, for $n = 2^{63} - 1$ this gives an upper bound on the error, which is:
$8/10 * (2^{63}-1)/2^{65} + 9/10 = 2^3/10 * (2^{63}-1)/2^{65} + 9/10 = 2^3/10*2^{63}/2^{65} - 2^3/10*1/2^{65} + 9/10$
$= 11/10 - 1/(10*2^{62}) \approx 1.1$ →since the latter number was so small

<2>
$2^{66} \equiv 4$ (mod10) hence $\text{ceil}(2^{66} / 10) - 2^{66}/10 = 6/10$.
n/10-floor(n/10) can be up to 9/10 for $n \equiv 9(\text{mod}10)$, for $n = 2^{63} - 1$,
$6/10 * (2^{63}-1)/2^{66} + 9/10 = 3/40 + 9/10 - 6/(10*2^{66}) \approx 0.975 < 1$
So formula 3 does give us floor(n/10) as desired.

<3> instructions:
data transfer:

movq %rdx %r14 : copy values from %rdx to %r14 movabsq $7378697629483820647, %rcx : copy absolute value (num.) to %rcx

Arithmetic:

imulq %rcx : signed multiply shrq $63, %rax : right shifts (unsigned divides) Count specified by the constant $63, the 16 bit contents of effective address (%rax) sarq $2, %rdx : Right shift, count specified by the contents (2), the 8-bit %rdx contents of effective address.

control flow:

retq : return start callq itos_recure : start an recursion with arguments in %esi, %rdi, %rdx

Represent negative number:
negq %rbx (Line 59 in itos_1) to handle negative data by make the negative data be positive using negq

Extension:
1. Analysis on itos1.s
In itos1.s, arguments are stored in %r15, %r14, %rbx, which are pushed into the stack so that according to the memory hierarchy, register is very fast so that this instructions can be faster than using (%rbp – 8) as arguments.

```
.cfi_startproc
pushq   %r15
.cfi_def_cfa_offset 16
pushq   %r14
.cfi_def_cfa_offset 24
pushq   %rbx
.cfi_def_cfa_offset 32
.cfi_offset %rbx, -32
.cfi_offset %r14, -24
.cfi_offset %r15, -16
```

Local arguments are operated directly without copy data of them into another register.

Negative numbers are handled with negq.

Confusion: what roles are “leaq” playing in this file?

2. (Original makefile is stored in BackUp folder)
For making MIPS architecture, an warning happens that what I have done is same as what I have learnt in Lecture 8. I cannot find the reason why it goes wrong.

```
pc2-033-l:~/Documents/CS2002/W04Practical/Files hl74$ make
clang -c -S -Wall -Wextra -target MIPS -O0 itos.c -o itos4.s
error: unknown target triple 'MIPS', please use -triple or -arch
make: *** [Makefile:39: itos4.s] Error 1
pc2-033-l:~/Documents/CS2002/W04Practical/Files hl74$ make
clang -c -S -Wall -Wextra -arch MIPS -O0 itos.c -o itos4.s
clang-7: warning: argument unused during compilation: '-arch MIPS' [-Wunused-command-line-argument]
pc2-033-l:~/Documents/CS2002/W04Practical/Files hl74$ ls
itos0-commented.s  itos1.s  itos3.s  itos.h  itos-stack.c  itos-stack.s  main.c  main-stack*  stack.c  s
itos0.s            itos2.s  itos.c   itos.o  itos-stack.o  main*         main.o  Makefile     stack.h
pc2-033-l:~/Documents/CS2002/W04Practical/Files hl74$ ^C
pc2-033-l:~/Documents/CS2002/W04Practical/Files hl74$ ^C
pc2-033-l:~/Documents/CS2002/W04Practical/Files hl74$ make
clang -c -S -Wall -Wextra -target mips -O0 itos.c -o itos4.s
error: unable to create target: 'No available targets are compatible with this triple.'
1 error generated.
make: *** [Makefile:39: itos4.s] Error 1
```

Next I tried to make file as in https://github.com/MIPT-ILab/mips-traces/blob/master/Makefile but it still doesn’t work.