RISC-V register naming convention for function calls

Legal Notice

As stated in the Canvas Syllabus, all course materials and relevant files placed in the course Google Drive must not be shared by the students outside of the course curriculum on any type of public domain site or for financial gain. Thus, if any course slides, quiz/ material, lab, etc. is found in any type of publicly available site (e.g., GitHub, stack Exchange), or for monetary gain (e.g., Chegg), then the original poster will be cited for misusing CSE12 course-based content and will be reported to UCSC for academic dishonesty.

In the case of sites such as Chegg.com, we have been able to locate course material shared by a previous quarter student. Chegg cooperated with us by providing the student's contact details, which was sufficient proof of the student's misconduct leading to an automatic failing grade in the course.

RISC-V register naming convention

As stated in lectures, the 32 registers in RV64I are numbered, x0 to x31. X0 is hardwired to always read zero. As for the rest of the registers, the RISC V reference sheet assigns the following naming convention (ignore all registers beyond x31 below):

REGISTER NAME, USE, CALLING CONVENTION

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

In the RARS text editor, when you refer to registers, you would use the register names mentioned in the NAME column above as opposed to their numbered names, example s1 instead of x9 and t1 instead of x6.

How register names are used in relation to function calls

The RISC-V register naming convention is not an arrangement hardwired into the RISC-V architecture. So technically, we would not cause the RISC-V processor to crash if we used random registers for random purposes. Except of course for the stack pointer register, sp/x2, and global pointer, gp/x3, which are preloaded with a set value. However, a typical assembly program might have hundreds of thousands of functions and it can get *very*

difficult to
keep track of how a particular register's value changes over multiple function calls. Thisis why we adopt this register naming convention as a matter of principle so that even if an assembly code has functions written by different authors, they would all blend seamlessly with each other when the overall program is compiled and executed.

The simplified explanation of the register usage protocol in RISC-V assembly

Before we discuss this, it is important to classify a function as being **caller vs callee**, and **leaf function vs non-leaf function**.

Caller vs Callee

The function that calls another function is the Caller. The function that is being called is known as callee. Example, in the C code below:

```
int f1()
{ int n=f2(-3,2,10);

        return 0;}

int f2(int a, int b, int c)
{ int n=a+b;

        return f3(n,c);}

int f3(int a, int b)
{ return a*b;}
```

Here, f1 is caller to f2 and f2 is callee to f1. f2 is caller to f3 and f3 is callee to f2.

Leaf vs non-leaf function

A function which does not call any other function is a leaf function. The exact opposite is a non-leaf function. A non-leaf function engages in a ***nested function call.***

In the above C example, only f3 is a leaf function since it in turn does not call anyone else

Who saves what register?

As per the RISC-V convention, before a **caller** does the function call, it must:

1. Save "temporaries" t0- t6, a0-a7 registers, to stack ***as needed***. That is, after we return from a function call, will we still be using certain t or a registers? If so, then you need to preserve their values in the main memory lest they are corrupted by the callee.
2. Load callee arguments in a0-a7 registers. As a caller, the only thing you know about a callee is which registers it accepts as arguments, what computation it does on them, and which register it return the value. 3. Now run **jump and link (jal)** instruction. Jal first loads the value PC+4 to return address register ra and then it jumps to callee function.

As per the RISC-V convention, when we are within the **callee** function, we must first do some setup before doing the actual function computation:

1. Allocate memory for new ***frame*** (sp = sp – frame), that is how much storing to memory will be done by callee.
2. Save callee-saved registers s0-s11, return address register ra to stack in memory ***as needed***. That is, for our

function computation, will we still be using certain s registers? If so, then you as callee need to preserve their values in the main memory lest their original values are meant to be used again by caller. If callee is a **leaf function**, then there is no need to bother with saving ra to memory since ra is not tampered with at all within leaf function.

Now callee is ready to do the computation. After having done so, let's do some housekeeping before we return to caller:

1. Place return value in a0 and a1 register, as per callee's description
2. Restore any callee-saved registers, and ra if you had previously saved them to memory. 3. "Pop" stack (sp = sp + frame size). That is, the frame of memory we had allocated for callee in main memory has now been cleared. This is essential since, if we keep on calling successive functions without clearing their stack frame after each call, then we will eventually run out of memory during program runtime! 4. Return to function using the **ret** instruction. Ret is actually a pseudo instruction but it suffices to not know what the original RISCV instruction it is representing is. Suffice to say, ret simply takes us back to that location in main memory whose address is given by ra. If ra value was not unintentionally tampered with, then this location should be in the caller right after where we had called the current function in the first place.

So now are back in the caller! Let's do some stuff before we proceed with the remaining code in caller:

1. Restore any caller-saved registers **as needed**. That is , if the caller had saved any t or a register to stack, then it needs to retrieve those original values.

Now, we are ready to movie on in the caller source code!

The examples in Lab4 folder

add_function.asm invokes a **leaf function** sum. We would have described sum in C as:

```
int sum (int n1, int n2)
{int n=n1+n2;
return n;}
```

The same is translated into RISC-V assembly code in *add_function.asm*. For Lab 4, it is suggested strongly to study this asm file in detail so that it helps you finish part 1 in Lab.

Multiply_function.asm invokes a nested function call. It contains two functions, multiply and sum. We would have described multiply in C as:

```
int multiply (int n1, int n2)
{
int result=0; int t4;

for (int i=0; i<n2; i++)

{ t4=sum (result, n1);
        result=t4;

}
return t4;}
```

The same is translated into RISC-V assembly code in *multiply_function.asm*.

nested function call in assembly, then you can easily create runtime errors where your memory references will run out of range!