Callercalleeagreement

Module 0372: Caller-callee agreement
Stack operation specification
Mutual agreement
Caller side
Callee side
The call frame
When there are local variables
When there are no local variables
Non-caller-callee specific
1. Stack operation specification
Register D is the designated stack point. Although register D is a general-purpose register, it is
designated by convention in order to specify the agreement between the caller and the callee.
To p u s h t h e va l u e a l re a d y l o a d e d i n t o re g i s t e r X (X c a n b e A , B , C , o r D) , t h e fo l l ow i n g C c o d e i s
used
// to push the value of register X *(–D) = X; The stack pointer is decremented first to reserve the space, then the content of register X is
copied to the location pointed to by register D. The equivalent TTPASM code is as follows:
dec d st (d),x // x is a placeholder for register a,b,c, or d In both the C and TTPASM implementations, it is clear that the stack pointer always points to
the last item pushed on the stack. This is because the stack pointer is adjusted prior to
overwriting the location that the stack pointer points to. In terms of notation, we can use D+
to refer to the address that is currently pointed to by the stack pointer.
To p o p t h e " t o p " o f t h e s t a c k t o re g i s t e r X , t h e o p e r a t i o n i s re ve r s e d. T h e fo l l ow i n g i s t h e C
code implementation:
X = *(D++);
The content pointed to by the stack pointer is copied to the register specified by X, then the
stack pointer is incremented to deallocate the location where the content was retrieved from.
The following is the TTPASM code:
ld x,(d) // x is a placeholder for register a,b,c, or d inc d Because of this stack convention, items that are pushed earlier occupy memory locations that
are higher in addresses. For example, the result of push 2 can be describe by the following:
location content
D+0 2
In this notation D+0 refers to the location that is pointed to by the stack pointer, D. The +
indicates an offset of zero.
However, if push 13 follows immediately, then stack becomes as follows:
Offset from where D points to content
+1 2
+0 13
In this table, the notation D+1 references the address that is one byte higher than where the
stack pointer (D) points to. Not that at this point, the stack pointer is decremented in the
process to push 13. As a result, the location that stores the value of 2 has not changed, but its
offset from where the stack pointer points to is changed because the stack pointer has
changed.
1. Mutual agreement
The mutual agreement between the caller and callee starts with how the stack operates, as
describe in the previous section.
1. 1 Caller side
if there are arguments:
arguments are pushed in reverse order, the last argument is pushed first
on TTP, arguments are pushed contiguously (no gaps between them)
this implies the last argument has the highest address in a call frame

pushed after all the arguments (if any)
as a result, the return address has an address that is lower than all the arguments
the stack pointer points to the return address at the entry point of the function being
called
jump to the entry point of the callee
Assuming to refer to arguments (each with a width of one byte) in a call to
function func, the frame is partially constructed by the caller at this point:
Offset from where D points
to
content
+n
+n-
+
+
return address to the instruction immediately after ‘jmpi
arg 1 arg n n arg n arg n 1 arg 1 func
the callee should return to the instruction right after the jump instruction with the return
address popped
In our example, the stack should like the following at the instruction right after the jmpi func
instruction:
Offset from where D points to content
+n-
+n-
... ...
+
if the callee returns a scalar value:
the return value is assumed to be in register A
the value of registers A, B and C are not assumed preserved
if there are arguments:
the caller has the responsibility to clean up the stack space used by arguments, if any
1. 2 Callee side
at the entry point of the callee:
the stack pointer points to the return address
if there are parameters:
the first parameter starts at the address immediately after (higher than) that of
the return address
the last parameter has the highest address
parameters are contiguous in TTP
in the callee’s code:
additional stack space may be utilized for local variables
the callee is responsible for moving the stack pointer lower to reserve space for the
arg n
arg n 1
arg 1
local variables
In the ongoing example, if we assume function func has local variables to , each
being one byte wide, then the stack looks like the following after the entry code of the callee
moves the stack pointer and allocate space for the local variables:
Offset from where D points to content
+n+m+
+n+m
... ...
+1+m
+m return address
+m-
... ...
+
Note that local variables are not restricted to one byte wide. Arrays and structures, for
example, will be more than one byte wide.
To c o n t i n u e w i t h a d d i t i o n a l a g re e m e n t s b e t we e n t h e c a l l e r a n d c a l l e e :

at the exit point of the callee:
if the callee has a scalar return value:
use register A to store the return value
the callee is responsible for popping the return address
the callee uses the popped return address to return to the caller

1. 3 The call frame
2. 3. 1 When there are local variables

var 1 var m
arg n
arg n 1
arg 1
var m
var 1

The call frame (or just "frame") refers to a region of stack space that provides the data context for a function to operate. This includes the space of local variables, the return address, and the parameters. The call frame is not allocated by the callee. Instead, it is partially constructed by the caller, including the parameters and the return address, and partially constructed by the callee, allocating the additional space for local variables.
After a frame is fully allocated, the stack pointer points to the base of the frame. Let us consider the following example:
void f (uint8t *x, uint8*t y) uint8$_t a, b$;
In this case, the caller is responsible to push the arguments and the return address. As a result, the following table describes the partially constructed frame at the entry point of function f:

Offset from where D points to	item description
+2	parameter y
+1	parameter x
+0	return address

Note how local variables a and b are not yet allocated. Function f is responsible for allocating the space needed for the local variables. Because a and b are both 1-byte wide, they need a total of 2 bytes in the frame. These two bytes are allocated at the entry point of function f. As a result, the local variables are located lower (in terms of addresses) than the return address. The following table show the content of the frame after local variables are explicitly allocated by the entry code of function f:

Offset from where D points to	item description
+4	parameter y
+3	parameter x
+2	return address
+1	local variable b
+0	local variable a

The offset to items of the frame may be defined as follows:
The function f is responsible to allocate and deallocate the space needed for local variables.
As a result, the shell of function f is as follows:
Because the callee is responsible for popping the return address, the stack looks like the following immediately after it returns to the caller:

Offset from where D points to	item description
+1	parameter y

f$_a$: 0*//offsettolocalvariablea, itisatthebaseoftheentireframefb:* *fa*1+*//offsettolocalvariablebflvs:* *fb*1+*//thetotalnumberofby*
*//theoffsettoparameterx, the*1+*isneededtoskipthereturnaddressfy:* *fx*1+*//theoffsettoparameteryf* : *ldib*, *f$_l$vssubd*, *b//allocate*
0*parameterx*

Note how the return address is not on the stack. However, parameters x and y (known as arguments from the caller's perspective) are still on the stack. As per the agreement, the caller needs to deallocate these two bytes on the stack. For this example, the following TTPASM code that immediately follows jmpi f in the call deallocates the arguments:
inc d // deallocate parameter x inc d // deallocate parameter y

1. 3. 2 When there are no local variables

A function may have zero (no) local variables. In this scenario, the approach to use func$_{lvs}$ as a label to represent the number of bytes used by local variables is still correct, but the label should be defined as zero. It is important to note that the return address is pointed to by the stack pointer in this case.
For example, let us consider the following C function definition:

two
parameters as follows at the entry point of function g:
Offset from where D points to description of item
+2 parameter y
+1 parameter x
+0 return address
The corresponding TTPASM code can be as follows:
g: $g_lvs : 0gx:\ glvs1+//add1toskipoverthereturnaddressgy:\ gx1+ldib, g_lvssubd, b//"allocate"localvariableseventhoughthereisnon$
4 Non-caller-callee specific
Any content below where the stack pointer points can be modified asynchronously.
In an architecture that supports interrupts, hardware interrupt can occur between the execution of any two instructions. A hardware interrupt invokes (calls) the corresponding ISR (interrupt service routine). This causes the return address to the interrupted code to be pushed on the stack. Imagine the stack as follows prior to an interrupt:
Offset from where D points to item description
+0 last item pushed
-1 remnant X
When an interrupt occurs, the processor pushes the return address from the ISR to the interrupted code on the stack, resulting in the following state:
Offset from where D points to item description
+1 last item pushed
+0 remnant X return address to interrupted code
Note how "last item pushed" is not affected because it was where the stack pointer points to when the interrupt occurs. However, "remnant X" is now overwritten by the return address to resume the execution of code prior to the interrupt.
After the ISR returns, the stack can be described as follows:
Offset from where D points to item description
+0 last item pushed
-1 remnant X return address to interrupted code
From the perspective of the code that was interrupted, the state of the stack has not changed, as long as there is no reference to locations below where the stack pointer points! Note that during the execution of an ISR, more locations on the stack may be overwritten. In other words, the overwriting of the location immediately below where the stack pointer points to at the moment of the interrupt is the minimum part of the stack to be used during the execution of an ISR.