Parameterandlocalvariables

Module 0307: Parameters and local variables About this module Parameters on the stack Local variables A more mechanical way to access frame items Exercise 1 Exercise 2

1. About this module

Prerequisites: 0304 , 0306 Objectives: This module discusses how parameters are passed in assembly language code in a way that is compatible with C code. Local variables are also discussed in this module

1. Parameters on the stack

Because there can many parameters, and recursion is to be supported (along with multi- threading), it is necessary to store parameters in a per-invocation basis, much like the return address. Let us examine the following caller code in C: subtract(3 , 5); The equivalent code in TTP assembly is as follows: ldi a, 5 dec d st (d),a // push 5 ldi a, 3 dec d st (d),a // push 3 ldi a,retAddr dec d st (d),a // push return address jmpi subtract retAddr: inc d inc d // deallocate parameters // now the stack should be balanced Note how the parameters are pushed in reverse order. This means the second parameter is pushed first, and then the first parameter. Another implication is that the first parameter has a lower address on the stack than the second parameter. Furthermore, also note how the return address is pushed last. Here is the subroutine code in C: int subtract (int x, int y) return x-y; The matching code in TTP assembly is as follows: subtract: // entry point of subroutine cpr c,d // c is a copy of the stack pointer ldi a, 1 // offset to find parameter x add c,a // now c is the address to x ld a,(c) // now a is parameter x inc c // now c is the address to y ld b,(c) // now b is parameter y sub a,b // perform the subtraction // a already has the right result to return ld b,(d) // d is never changed, still pointer // to the return address inc d // callee only deallocates the retAddr // NOT the parameters! jmp b // now return and continue in caller It is important to leave register D alone and not to use it directly to compute the address of parameters.

1. Local variables

Local variables of a subroutine are allocated on a per invocation basis on the stack, much like parameters. However, parameters are allocated and initialized by the caller, while local variables are allocated by the called subroutine. As a result, local variables have addresses that are below that of the return address. The method to access local variables is the same as accessing parameters. The main difficulty of accessing local variables is tracking the stack pointer as items are pushed and popped. In many real architectures, there is a "frame point" that points to a fixed place relative to items used by a subroutine. TTP is too primitive to dedicate yet another register for this purpose.

1. A more mechanical way to access frame

items The parameters, return address, and local variables collectively make up the call frame of an invocation of a function. From the perspective of the callee, the parameters, and return address are already pushed on the stack at the entry point. However, a callee is responsible to allocate stack space for its own local variables. It is best to use symbolic names to reference the offset to items in a frame from where the stack pointer points to after the entire frame is allocated. For example, consider the following C code: void someFunc (uint8t x, $uint8$t y, uint8$_t z$)$uint8_t a, b, c$;$The corresponding frame looks like the following$: $offset from where SP points to what 6 z 5 y 4 x 3 return address 2 c 1 b 0 a From the callee's perspective, some label definitions can be used to fi$ $someFunc_a$: 0//$offset from where D points to to find local var a someFunc b:$ $someFunc a$ 1+//$offset to local var b some Func c:$ $someF$ //$offset to local var c some Func local Var Size:$ $someFunc c$ 1+//$bytes used for local vars Parameters are also in a call frame, they can also$ $someFunc x:$ $someFunc local Var Size$ 1+//$add 1 to skip over the return address some Func y:$ $someFunc x$ 1+$someFunc z:$ $someFunc y$ 1+ $The \shell" of this callee consists of the code to allocate and deallocate the local variables, as well as the code to return to the caller$: $someFu$ //$entry point of the callee ldia, someFunc_l ocal Var Size subd, a$//$allocate stack space for local variables$//$...code for the body of someFu$ $ldia, someFunc_b$//$a = b-SP load offset to a register add a, d$//$a = b$ the address of the item on the frame W hether the address of a frame $void some Func(uint8$t x, $uint8$t y, $uint8_t z$)$uint8_t a, b, c$; //$...some Other Func(a, z)$; //$...The code to call some Other Func is then as follo$

Exercise 1

Implement the following program in TTP assembly language:

1 include <stdint,h>

void swap (uint8t pX, $uint8$t pY); int main () uint8$_t x, y$; $x = 2; y = 5; swap(x, y); return 0; void swap(uint8$t pX, $uint8$t pY) ldi$ $z-SP add a, d$//$a = z lda, (a)$//$a = z decd$//$at this point, the SP points to one byte below local var a$!$st(d), a$//$push z ldia, some Func$ //$reg a = (local var a)-SP, add 1 to compensate for the second argument already pushed add a, d$//$reg a = (local var a)$ $decd st(d), a$//$push return address in cd$//$deallocate the first argument in cd$//$deallocate the second argument uint8_t t; t = *pX; p$

Exercise 2

Implement the following program in TTP assembly language:

2 include <stdint.h>

uint8t power (uint8t n); int main () uint8$_t x$; $x = power(3); return 0; uint8$t power (uint8$t n) if(!n) return 1; else return 2 *$