**8** Parameter Passing

Pass by Value

1 // ex0801.c Pass by value

2 #include <stdio.h>

3 int x = 7;

4 void f(int a)

5 {

6 a = a + 1;

7 }

8 =====================

9 int main()

10 {

11 f(x);

12 printf("%d\n", x);

13 return 0;

14 }

When main calls f (line 11), it passes the value of x, which is 7. The parameter a in f receives this value. Thus, on entry into f, x and a are

x a

7

7

Note that x and a correspond to separate locations in memory. Thus, when f increments a (line 6), the value in x is unaffected:

x a

~~7~~ 8

7

Here is the assembler code for this program:

Example of Pass by Value

1 ; ex0801.a Pass by value

2 startup: bl main

3 halt

4 ;==============================================================

5 ; #include <stdio.h>

6 x: .word 7 ; int x = 7;

7 ; void f(int a)

8 f: push lr ; {

9 push fp

10 mov fp, sp

11

12 ldr r0, fp, 2 ; a = a + 1;

13 add r0, r0, 1

14 str r0, fp, 2

15

16 mov sp, fp

17 pop fp

18 pop lr

19 ret

20 ; }

21 ;===========================================================

22 ; int main()

23 main: push lr ; {

24 push fp

25 mov fp, sp

26

27 ld r0, x ; f(x);

28 push r0

29 bl f

30 add sp, sp, 1

31

32 ld r0, x ; printf("%d\n", x);

33 dout r0

34 nl

35

36 mov r0, 0 ; return 0;

37 mov sp, fp

38 pop fp

39 pop lr

40 ret

41 ; }

42

Pass by Address

f(&x);

1 // ex0802.c Pass by address

2 #include <stdio.h>

3 int x = 7;

4 void f(int \*p)

5 {

6 \*p = \*p + 1;

7 }

8 //===================

9 int main()

10 {

11 f(&x); // pass by address

12 printf("%d\n", x);

13 return 0;

14 }

The parameter p in f receives this address. Thus, on entry into f, p points to x:

x p

7

On line 6, f dereferences p twice: first to access the value in x and second to store a new value in x, after which x and p are

x p

~~7~~ 8

Because f changes the value of the argument x, we say that f has a *side effect*—that is, it has a non-local effect. In particular, it changes a variable not local to f (the global variable x). Here is the assembler code for the program:

Example of Pass by Address

1 ; ex0802.a Pass by address

2 startup: bl main

3 halt

4 ;==============================================================

5 ; #include <stdio.h>

6 x: .word 7 ; int x = 7;

7

8 ; void f(int \*p)

9 f: push lr ; {

10 push fp

11 mov fp, sp

12

13 ldr r0, fp, 2 ; \*p = \*p + 1;

14 ldr r0, r0, 0

15 add r0, r0, 1

16 ldr r1, fp, 2

17 str r0, r1, 0

18

19 mov sp, fp ; }

20 pop fp

21 pop lr

22 ret

23 ;==============================================================

24 ; int main()

25 main: push lr ; {

26 push fp

27 mov fp, sp

28

29 lea r0, x ; f(&x);

30 push r0

31 bl f

32 add sp, sp, 1

33

34 ld r0, x ; printf("%d\n", x);

35 dout r0

36 nl

37

38 mov r0, 0 ; return 0;

39 mov sp, fp

40 pop fp

41 pop lr

42 ret

43 ; }

Several Important Observations on Pass by Address

* The standard mechanism for returning a value to a calling function is for the called function to place the return value in r0 prior to executing the ret instruction. However, because pass by address can have side effects, it can also be used to return values. We saw this in the program above. It returns 8 to main by storing 8 in x. It does this by the dereferencing p, the parameter corresponding to x. One advantage of the pass-by-address mechanism over the r0 return mechanism is that the pass-by-address mechanism can return any number of values, one for each pass-by-address parameter. The r0 mechanism, however, can return only one value.
* To access a value parameter requires only one instruction (ldr or str) but to dereference a pass-by-address parameter requires a two-instruction sequence (ldr-ldr or ldr-str). Thus, there is a cost in both time and space associated with using pass by address.
* In general, it is better to pass an aggregate structure, such as an array, by address than by value. For example, suppose an argument in a function call is a 1000-slot array. If the array is passed by value, then the calling sequence would push a copy of the *entire array* onto the stack. If, however, the array is passed by address, then the calling sequence would push only the address of the array.

Thus, for a large array, pass by value is grossly inefficient compared to pass by address.

Pass by Value-Result

1 // ex0803.c Pass by value-result (not supported by C)

2 #include <stdio.h>

3 int x = 7;

4 void f(int $a) ; a is a value-result parameter

5 {

$ indicates that a is a value-result parameter

6 a = a + 1;

7 }

8 //===================

9 int main()

10 {

11 f(x);

12 printf("%d\n", x);

13 return 0;

14 }

Example of Pass by Value-Result

1 ; ex0803.a Pass by value-result (not supported by C)

2

3 startup: mov sp, 0 ; initialize stack pointer

4 mov fp, 0 ; initialize frame pointer

5 bl main

6 halt ; back to operating system

7 ;==============================================================

8 ; #include <stdio.h>

9 x: .word 7 ; int x = 7;

10 ; void f(int $a)

11 f: push lr ; {

12 push fp

13 mov fp, sp

14

15 ldr r0, fp, 2 ; a = a + 1;

16 add r0, r0, 1

17 str r0, fp, 2

18

19 mov sp, fp

20 pop fp

21 pop lr

22 ret

23 ; }

24 ;===========================================================

25 ; int main()

26 main: ; {

27 push lr

28 push fp

29 mov fp, sp

30

31 ld r0, x ; f(x);

32 push r0

33 bl f

Code that creates side effect

34 pop r0

35 st r0, x

36

37 ld r0, x ; printf("%d\n", x);

38 dout r0

39 nl

40

41 mov r0, 0 ; return 0;

42 mov sp, fp

43 pop fp

44 pop lr

45 ret

46 ; }

Pass by Name

f(a+b);

1 // ex0804.c Pass by name (not supported by C)

2 #include <stdio.h>

# indicates that x is a name parameter

3 int a = 1;

4 void f(int #x)

5 {

6 printf("%d\n", x); // displays 3

7 a = a + 2;

8 printf("%d\n", x); // displays 5

9 }

10 //===================

11 int main()

12 {

13 int b = 2;

14 f(a+b);

15 return 0;

16 }

The call of f on line 14 conceptually passes the expression a+b—not its value—to f. This expression replaces every occurrence of the parameter x in f. Thus, lines 6 and 8 both become

printf("%d\n", a+b);

Calling function

br

thunk

calling

sequence

What should the calling sequence pass to the called function? The called function has to call the thunk. So the calling sequence *has to pass the address of the thunk*. Here is the thunk and the calling sequence for the function call on line 14 in the program in ex0804.c:

1 br @L0 branch over the thunk to the calling sequence

2

3 @L1: ld r0, a

4 ldr r1, fp, -1 thunk that evaluates a+b and returns its value

5 add r0, r0, r1

6 str r0, fp, -2 store value of a+b in implicit variable

7 add r0, fp, -2 get address of implicit variable

8 ret

9

10 @L0: sub sp, sp, 1

11 lea r0, @L1

12 push r0 calling sequence

13 bl f

Pass address

of thunk

14 add sp, sp, 2

sp

Points to main’s stack frame

fp ~ ~

saved fp stack frame

saved lr for f (called function)

x

Points to the thunk

implicit var

b stack frame

saved fp for main (calling function)

saved lr

~ ~

1. Load r0 with the address of the thunk, which is in the name parameter x. x is at the offset 2 in f’s stack frame (see the stack diagram above).

.

ldr r0, fp, 2

1. Save fp by pushing its contents onto the stack.

push fp

1. Reset fp to point to main’s stack frame. fp is pointing to main’s fp (see the stack diagram above). Thus, to reset fp, load fp from the stack location that fp points to.

ldr fp, fp, 0

The stack now looks like this:

Points to f’s stack frame

sp ~ ~

f’s fp

fp saved fp stack frame

saved lr for f

x

implicit var

b stack frame

saved fp for main

saved lr

~ ~

1. Call the thunk via the address in r0 from step 1 above.

blr r0

1. On return from the thunk, reset fp to point to f’s stack frame.

pop fp

1. Dereference the address returned by the thunk to get the value of the argument.

ldr r0, r0, 0

1. Use the value of the argument as indicated by the C code that references the name parameter. For this example, display the value of the argument.

dout r0

nl

1 ; ex0804.a Pass by name (not supported by C)

2 startup: bl main

3 halt ; back to operating system

4 ;==============================================================

5 ; #include <stdio.h>

6 a: .word 1 ; int a = 1;

7

8 f: push lr ; void f(int #x)

9 push fp ; {

10 mov fp, sp

11

12 ; get thunk addr ; printf("%d\n", x);

13 ldr r0, fp, 2

14 ; save f's fp

15 push fp

16 ; restore fp with caller's fp

17 ldr fp, fp, 0

18 ; call thunk

19 blr r0

20 ; restore fp with called function's fp

21 pop fp

22 ; dereference address returned by thunk

23 ldr r0, r0, 0

24 ; display value returned

25 dout r0

26 nl

27

28 ld r0, a ; a = a + 2;

29 add r0, r0, 2

30 st r0, a

31

32 ; get thunk addr ; printf("%d\n", x);

33 ldr r0, fp, 2

34 ; save f's fp

35 push fp

36 ; restore fp with caller's fp

37 ldr fp, fp, 0

38 ; call thunk

39 blr r0

40 ; restore fp with called function's fp

41 pop fp

42 ; dereference address returned by thunk

43 ldr r0, r0, 0

44 ; display value returned

45 dout r0

46 nl

47

48 mov sp, fp ; }

49 pop fp

50 pop lr

51 ret

52 ;==============================================================

53 main: push lr ; int main()

54 push fp ; {

55 mov fp, sp

56

57 mov r0, 2 ; int b = 2;

58 push r0

59

60 ; branch over thunk

61 br @L0 ; f(a+b);

62

63 ; thunk

64 @L1: ld r0, a

65 ldr r1, fp, -1

66 add r0, r0, r1

Get address of implicit variable

67 str r0, fp, -2

68 add r0, fp, -2

69 ret

Creates implicit variable

70

71 ; calling sequence

72 @L0: sub sp, sp, 1

73 lea r0, @L1

Passes address of thunk

74 push r0

75 bl f

76 add sp, sp, 2

77

78 mov r0, 0 ; return 0;

79 mov sp, fp

80 pop fp

81 pop lr

82 ret

83 ; }

The effect of passing an argument by name is to substitute it for the corresponding parameter in the called function. For example, in the program above, the argument a+b in effect is substituted for x in f, changing the printf statements from

printf("%d\n", x);

to

printf("%d\n", a+b);

f(a);

then the compiler generates the following thunk and calling sequence:

1 br @L0

2

3 @L1: lea r0, a ; thunk returns address of single-var argument

4 ret

5

6 @L0: lea r0, @L1

7 push r0

8 bl f

9 add sp, sp, 1

A name parameter that appears on the left side of an assignment statement is handled differently from name parameters that appear elsewhere. For example, consider the following program in which the name parameter x on line 6 appears on the left side of an assignment statement:

1 // ex0805.c Pass by name (not supported by C)

2 #include <stdio.h>

3 int a;

4 void f(int #x)

5 {

6 x = 5;

7 }

8 //===================

9 int main()

10 {

11 f(a);

12 return 0;

13 }

For line 6, the code does not *load* the value at the address returned by the thunk. Instead, it *stores* 5 (the value of the right side of the assignment statement) at the address returned by the thunk:

1 ; get the address of the thunk

2 ldr r0, fp, 2 ; x = 5;

3 ; save f's fp

4 push fp

5 ; restore fp with caller's fp

6 ldr fp, fp, 0

7 ; call thunk

8 blr r0

9 ; restore fp with f's fp

Store value at address returned by the thunk

10 pop fp

11

12 mov r1, 5

13 str r1, r0, 0