University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

C to LC-3 Example: Finding an Absolute Value

Let's Act Like Compilers!

Let's have some fun!

Let's pretend to be a C compiler!

No, really, I expect to hear cheers.

Task: Convert a Number to Its Absolute Value

Here's our task:
- let the user **type in a number**,
- **convert** the number **to its absolute value**,
- then **print the result in hexadecimal**.

We'll write this function:*

```
int32_t find_abs (int32_t num);
```

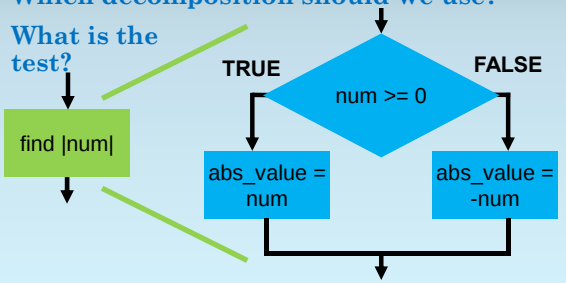
returns an `int32_t` | name of subroutine | `int32_t` as input

*We'll discuss function declarations/signatures in more detail later.

Decompose Finding Absolute Value

Which decomposition should we use?

What is the test?



A C Function to Find Absolute Value

Here's the function.

```
int32_t find_abs (int32_t num)
{
    int32_t abs_value;
    abs_value = (0 <= num ?
                 num : -num);
    return abs_value;
}
```

our function signature

conditional construct using conditional operator

Using `find_abs` Function in C

Also in the program **`translate.c`** (see web page for full version):

file scope variable

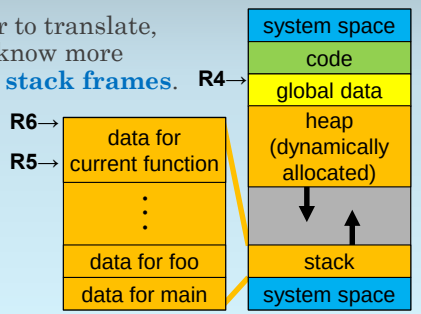
```
static int32_t the_number;

// ... and inside main ...
the_number = find_abs (the_number);
```

We will translate this function call first.

First, We Must Learn About Stack Frames in LC-3

But in order to translate, we need to know more about **LC-3 stack frames**.



LC-3 Stack Frames Contain Five Elements

Remember what's in a stack frame?

This is the order on the stack…

- Local variables
- Address of caller's stack frame
- Return address (R7 in LC-3)
- Outputs (return value)
- Inputs (parameters, arguments)

these form the **linkage**

Why are parameters on the bottom?

Stack Frame Creation Shared by Caller and Callee

Who chooses parameter values?
(Caller or callee?)

Caller pushes the parameters onto the stack.

For example, **`main`** pushes the input to **`find_abs`**.

Callee creates the remainder of the stack frame.

When JSR Returns, Return Value is on Top of Stack

Why is the return value next on the stack?

Local variables
Address of caller's stack frame
Return address (R7 in LC-3)
Outputs (return value)
Inputs (parameters, arguments)

these form the **linkage**

Return value remains on stack on return.

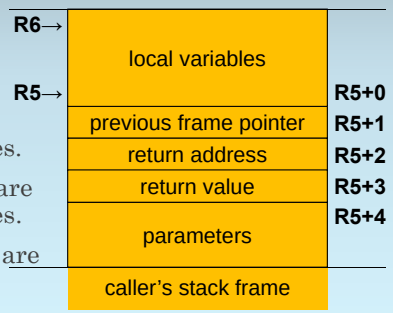
Local Variables and Parameters Accessed Using R5

R6 points to top of stack.

R5 points to bottom of local variables.

R5+0, -1, … are local variables.

R5+4, +5, … are parameters.



Compilers Use Symbol Tables to Locate Variables

How does a compiler generate instructions?

First, it builds a **symbol table** (like an assembler's, but with more information).

Here's an example for **`translate.c`**:

scope	identifier	type	from	offset	…
`translate.c`	`the_number`	`int32_t`	R4	0	…
`find_abs`	`abs_value`	`int32_t`	R5	0	…
`find_abs`	`num`	`int32_t`	R5	4	…

The Statement in Main Begins with a Function Call

Now we're ready to translate the statement:

```
the_number = find_abs (the_number);
```

Remember that for an assignment, the compiler generates instructions to...

1. evaluate the expression on the right, then
2. store the result into the address on the left.

So **first**, we must **call the function**.

Calling a Function Consists of Four Steps

Calling a function consists of four steps:

1. evaluate and push the parameters,
2. call the function (with JSR),
3. read the return value from the top of the stack, and
4. pop off the return value and the parameters.

Evaluate Expressions Used for Parameter Values

Step 1: Evaluate and push parameters.

```
the_number = find_abs (the_number);
```

The function is called with one parameter.

Where is it?

Let's look it up in the symbol table!

scope	identifier	type	from	offset	
translate.c	the_number	int32_t	R4	0	...
find_abs	abs_value	int32_t	R5	0	...
find_abs	num	int32_t	R5	4	...

Read the Variable `the_number` into R0

LDR R0,R4,#0

We're finally ready to write code!

First, read `the_number` into R0.

scope	identifier	type	from	offset	
translate.c	the_number	int32_t	R4	0	...
find_abs	abs_value	int32_t	R5	0	...
find_abs	num	int32_t	R5	4	...

Push R0 (Parameter Value) onto the Stack

```
LDR R0,R4,#0
ADD R6,R6,#-1
STR R0,R6,#0
```

Next, push R0 onto the stack.

Remember the two instructions used to push?

Call the FIND_ABS Function

```
LDR R0,R4,#0
ADD R6,R6,#-1
STR R0,R6,#0
JSR FIND_ABS
```

Step 2: Call the function.

Is there an LC-3 instruction for that?

Read the Return Value from the Top of the Stack

```
LDR R0,R4,#0
ADD R6,R6,#-1
STR R0,R6,#0
JSR FIND_ABS
LDR R0,R6,#0
```

Step 3: Read the return value.

Remember that after JSR, the return value is on top of the stack.

Is there an LC-3 instruction for that?

Pop Return Value and Parameter(s) from Stack

```
LDR R0,R4,#0
ADD R6,R6,#-1
STR R0,R6,#0
JSR FIND_ABS
LDR R0,R6,#0
ADD R6,R6,#2
```

Step 4: Pop return value and parameter.

Is there an LC-3 instruction for that?

That's it for the function call.

Now what?

Write Return Value Back into `the_number`

```
the_number = find_abs (the_number);
```

R0 now holds the value of the right side.

So we need to store into `the_number`.

Where is `the_number` again?

Let's look it up in the symbol table!

scope	identifier	type	from	offset	...
translate.c	the_number	int32_t	R4	0	...
find_abs	abs_value	int32_t	R5	0	...
find_abs	num	int32_t	R5	4	...

Store R0 into `the_number`

```
LDR R0,R4,#0
ADD R6,R6,#-1
STR R0,R6,#0
JSR FIND_ABS
LDR R0,R6,#0
ADD R6,R6,#2
STR R0,R4,#0
```

Write R0 into `the_number`.

Is there an LC-3 instruction for that?

scope	identifier	type	from	offset	...
translate.c	the_number	int32_t	R4	0	...

Reference Version of Function Call and Assignment

```
LDR R0,R4,#0
ADD R6,R6,#-1
STR R0,R6,#0
JSR FIND_ABS
LDR R0,R6,#0
ADD R6,R6,#2
STR R0,R4,#0
```

That's it for the function call!

You can find both the C code and the LC-3 code on the web page.

```
the_number = find_abs (the_number);
```


Code for a Function Consists of Four Parts

Now we're ready to translate `find_abs`.

A function's code consists of four parts:

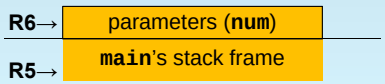
1. set up the stack frame,
2. execute the statements,
3. tear down the stack frame (leaving the return address on the stack with LC-3),
4. and return (RET).

Stack Appearance on Entry to `find_abs`

When **`find_abs`** starts execution, the **stack appears as shown below**…

R6 points to the **parameters**, which **are already on the stack** (pushed by the caller).

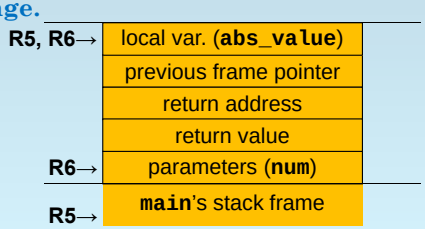
Below parameters is the caller's stack frame, and **R5** points into it (somewhere).



Stack Frame for `find_abs` (During Execution of Code)

The stack frame should look like this…

Setting up the stack frame means making this change.



Make Space for the Remainder of the Stack Frame

```
FIND_ABS
ADD R6,R6,#-4
```

First, make space on the stack.

Is there an LC-3 instruction for that?

How many locations do we need?

Save Caller's Frame Pointer into Stack Frame

```
FIND_ABS
ADD R6,R6,#-4
STR R5,R6,#1
```

Next, save the caller's frame pointer (R5).

Is there an LC-3 instruction for that?

Where does it go?

Note: offset depends on space for local variables.

Set Frame Pointer for `find_abs`

```
FIND_ABS
ADD R6,R6,#-4
STR R5,R6,#1
ADD R5,R6,#0
```

Next, set R5 to point to the lowest local variable.

Is there an LC-3 instruction for that?

How much do we add?

Note: amount added depends on space for local variables.

Save Return Address into the Stack Frame

```
FIND_ABS
ADD R6,R6,#-4
STR R5,R6,#1
ADD R5,R6,#0
STR R7,R5,#2
```

Finally, save R7 into the stack frame.

Is there an LC-3 instruction for that?

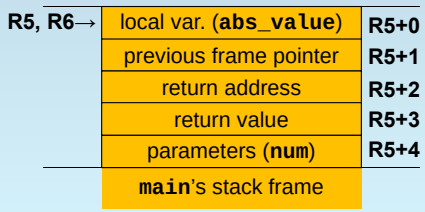
What are the base register and offset?

Note: always the same offset from R5.

Stack Frame for `find_abs` (During Execution of Code)

Now we can write code for the C statements.

Note that offsets match the symbol table.



Implement the First C Statement

Here's the first statement.

```
abs_value = (0 <= num ? num : -num);
```

We start with the **test**.

Where is num?

Look in the symbol table!

scope	identifier	type	from	offset	...
translate.c	the_number	int32_t	R4	0	...
find_abs	abs_value	int32_t	R5	0	
find_abs	num	int32_t	R5	4	...

Load Variable **num** into R0

LDR R0,R5,#4

Load **num** into R0.

Is there an LC-3 instruction for that?

scope	identifier	type	from	offset	...
translate.c	the_number	int32_t	R4	0	...
find_abs	abs_value	int32_t	R5	0	
find_abs	num	int32_t	R5	4	...

Check the Sign of Parameter **num**

LDR R0,R5,#4
BRn ELSE_COND

Branch on false (**num < 0**)

ELSE_COND

Is there an LC-3 instruction for that?

What are the condition codes?

Test is True, So Expression Value is **num**

```
abs_value = (0 <= num ? num : -num);
```

The test is true: **expression's value** is **num**.

Where is num?

Look in the symbol table!

scope	identifier	type	from	offset	...
translate.c	the_number	int32_t	R4	0	...
find_abs	abs_value	int32_t	R5	0	
find_abs	num	int32_t	R5	4	...

Load Variable **num** into R0

LDR R0,R5,#4
BRn ELSE_COND
LDR R0,R5,#4

Load **num** into R0.

ELSE_COND

Is there an LC-3 instruction for that?

scope	identifier	type	from	offset	...
translate.c	the_number	int32_t	R4	0	...
find_abs	abs_value	int32_t	R5	0	
find_abs	num	int32_t	R5	4	...

Conditional Operator is Complete: Branch to Assignment

```
LDR R0,R5,#4
BRn ELSE_COND
LDR R0,R5,#4
BRnzp DONE_COND
ELSE_COND


DONE_COND
```

Branch to assignment (to **`abs_value`**).

Is there an LC-3 instruction for that?

What are the condition codes?

Test is False, So Expression Value is `-num`

Now for the 'else' condition...

```
abs_value = (0 <= num ? num : -num);
```

The test is false: **expression's value** is `-num`.

Where is num?

Look in the symbol table!

scope	identifier	type	from	offset	...
`translate.c`	`the_number`	`int32_t`	R4	0	...
`find_abs`	`abs_value`	`int32_t`	R5	0	
`find_abs`	`num`	`int32_t`	R5	4	...

Load Variable **`num`** into R0

```
LDR R0,R5,#4
BRn ELSE_COND
LDR R0,R5,#4
BRnzp DONE_COND
ELSE_COND
LDR R0,R5,#4


DONE_COND
```

Load **`num`** into R0.

Ok, ok! I won't ask you!

Just be glad that you're a human.

Computers are dumb.

Negate R0

```
LDR R0,R5,#4
BRn ELSE_COND
LDR R0,R5,#4
BRnzp DONE_COND
ELSE_COND
LDR R0,R5,#4
NOT R0,R0
ADD R0,R0,#1
DONE_COND
```

Negate R0.

Is there an LC-3 instruction for that?

No.

But we can use two.

Finish the Assignment Operator

The right side's value is now in **R0**.

```
abs_value = (0 <= num ? num : -num);
```

Let's store it into **abs_value**.

Where is abs_value?

Look in the symbol table!

scope	identifier	type	from	offset	...
translate.c	the_number	int32_t	R4	0	
find_abs	abs_value	int32_t	R5	0	...
find_abs	num	int32_t	R5	4	...

Store R0 into Variable abs_value

```
LDR R0,R5,#4
BRn ELSE_COND
LDR R0,R5,#4
BRnzp DONE_COND
ELSE_COND
LDR R0,R5,#4
NOT R0,R0
ADD R0,R0,#1
DONE_COND
STR R0,R5,#0
```

Store R0 into **abs_value**.

Is there an LC-3 instruction for that?

identifier	type	from	offset
the_number	int32_t	R4	0
abs_value	int32_t	R5	0
num	int32_t	R5	4

We Have Translated the First C Statement!

```
LDR R0,R5,#4
BRn ELSE_COND
LDR R0,R5,#4
BRnzp DONE_COND
ELSE_COND
LDR R0,R5,#4
NOT R0,R0
ADD R0,R0,#1
DONE_COND
STR R0,R5,#0
```

The statement is complete!

```
abs_value =
    (0 <= num ?
     num : -num);
```


Implement the Second (and Last) C Statement

Here's the second statement.

```
return abs_value;
```

(Copy **abs_value** to return value, then RET.)

Where is abs_value?

Look in the symbol table!

scope	identifier	type	from	offset	...
translate.c	the_number	int32_t	R4	0	
find_abs	abs_value	int32_t	R5	0	...
find_abs	num	int32_t	R5	4	...

Load Variable **abs_value** into R0

LDR R0,R5,#0

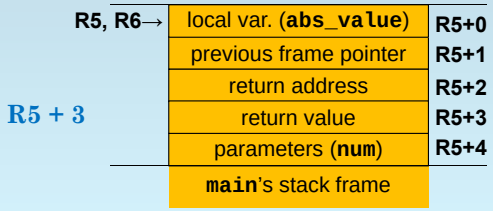
Load **abs_value** into R0.

Is there an LC-3 instruction for that?

scope	identifier	type	from	offset	...
translate.c	the_number	int32_t	R4	0	...
find_abs	abs_value	int32_t	R5	0	...
find_abs	num	int32_t	R5	4	...

Where is the Return Value Stored?

Where does the return value go?
Look in the stack frame!



Store R0 in Return Value Slot of Stack Frame

LDR R0,R5,#0
STR R0,R5,#3

Store R0 into return value slot.

Is there an LC-3 instruction for that?

We Have Translated the Code for **find_abs**!

LDR R0,R5,#0
STR R0,R5,#3

The statement is complete!

```
return abs_value;
```


Time to Tear Down the Stack Frame

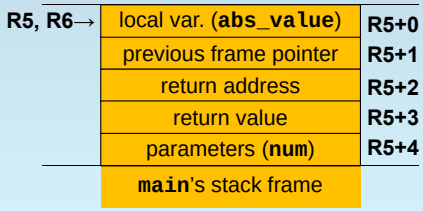
Time for Step 3: **tear down the stack frame**.

A function's code consists of four parts:

1. set up the stack frame,
2. execute the statements,
3. tear down the stack frame (leaving the return address on the stack with LC-3),
4. and return (RET).

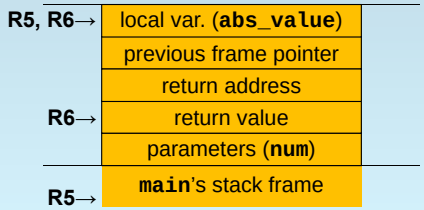
Stack Appearance Before Tearing Down Stack Frame

Here's the stack frame during execution of the statements in **`find_abs`**.



Stack Appearance After Tearing Down Stack Frame

We need to **pop down to the return value** and **reset R5 to `main`'s frame pointer**.



Restore Return Address from the Stack Frame

LDR R7,R5,#2

First, restore R7 from the stack frame.

Is there an LC-3 instruction for that?

Note: always the same offset from R5.

What are the base register and offset?

Restore Caller's Frame Pointer from the Stack Frame

```
LDR R7,R5,#2
LDR R5,R5,#1
```

Next, restore caller's frame pointer (R5).

Is there an LC-3 instruction for that?

Note: always the same offset from R5.

What are the base register and offset?

Pop Down to Return Value

```
LDR R7,R5,#2
LDR R5,R5,#1
ADD R6,R6,#3
```

Finally, pop the stack down to the return value.

Is there an LC-3 instruction for that?

Note: amount added depends on space for local variables.

How much do we add?

One More Step… Return!

Time for Step 4: **return to caller**.

A function's code consists of four parts:

1. set up the stack frame,
2. execute the statements,
3. tear down the stack frame (leaving the return value on the stack with LC-3),
4. and return (RET).

We're Done! It's Time to Return to the Caller

```
LDR R7,R5,#2
LDR R5,R5,#1
ADD R6,R6,#3
RET
```

Return to the caller.

Is there an LC-3 instruction for that?

Code is Available on the Web Page

Remember that this code is available on the web page:

- **`translate.c`** – the C version
- **`translate.asm`** – the LC-3 version

I took some liberties in the translation, but **the call and the function `find_abs` are as shown here**.

See comments in the code for details.

