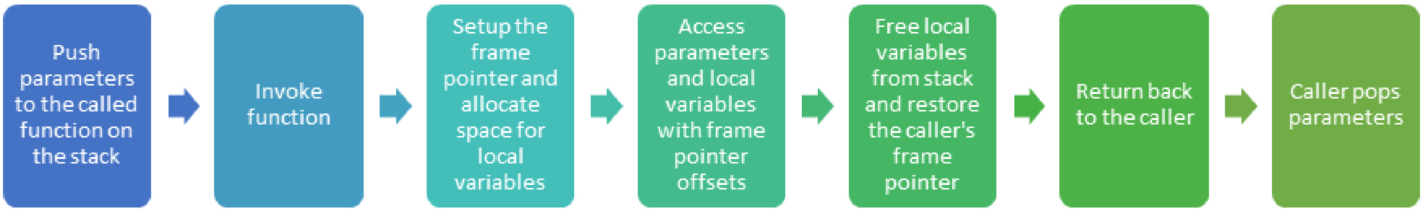
- EventHelix.com
- eventstudio *model object and message flows*
- visualether *Wireshark pcap to call flow*
- 5G NR *call flows architecture*
- LTE *call flows architecture*
- company *contact us support*

C to assembly: function calling


Push parameters to the called function on the stack
Invoke function
Setup the frame pointer and allocate space for local variables
Access parameters and local variables with frame pointer offsets
Free local variables from stack and restore the caller's frame pointer
Return back to the caller
Caller pops parameters

Even though most programming is now carried out in high level languages, a good understanding of the generated assembly code really helps in debugging, performance analysis and performance tuning.

Here we present a series of articles describing C to assembly translation. We will be mapping C code to pseudo-assembly. The concepts learnt here can easily be applied to understand the generated code for any real processor assembler.

In this article, we will discuss the assembly code generated for function calling, parameter passing and local variable management. Before we go any further we need to discuss a few things about the pseudo-assembler.

Pseudo assembler basics

- Processor registers are designated as R0, R1, etc.
- The MOVE instruction has the source on the left side and destination on the right side.
- Register RETURN_VALUE_REGISTER is used to return values to the calling function.
- The stack in the pseudo-procesor grows from higher address to lower address. Thus a push results in a decrement to the stack pointer. A pop results in an increment to the stack pointer.
- Register STACK_POINTER is used to point the stack.
- Register FRAME_POINTER is used as the frame pointer. The frame pointer serves as an anchor between the called and the calling function.
- When a function is called, the function first saves the current value of the FRAME_POINTER on the stack. It then saves the value of the STACK_POINTER register in FRAME_POINTER register. This is followed by decrements the STACK_POINTER register to allocate space for local variables.
- The FRAME_POINTER register is used to access local variables and parameters. Local variables are located at a negative offset to the frame pointer. Parameters passed to the function are located at a positive offset to the frame pointer.
- When the function returns, the FRAME_POINTER register is copied into the STACK_POINTER register. This frees up the stack used for local variables. The value of FRAME_POINTER register for the caller of this function is restored from the stack by a pop.

Function calling

The following block shows the C code and the corresponding generated assembly code.

C Code

```
int CallingFunction(int x)
{
   int y;
   CalledFunction(1,2);
   return (5);
}
```

```

void CalledFunction(int param1, int param2)
{
   int local1, local2;
   local1 = param2;
}
```

The generated assembly code is shown along with the corresponding C code.

Pseudo Assembler Code

```
int CallingFunction(int x)
{
   int y;
      // Reserving space for local variable y (4 bytes)
      PUSH FRAME_POINTER
      MOVE STACK_POINTER, FRAME_POINTER
      ADD #-4, STACK_POINTER
   CalledFunction(1,2);
      // Pushing the second parameter on the stack
      PUSH #2
      // Pushing the first parameter on the stack
      PUSH #1
      // Calling the CalledFunction()
      CALL_SUBROUTINE _CalledFunction
      // Pop out the parameters after return
      ADD #8, STACK_POINTER
   return (5);
      // Copy the returned value 5 into R0 (As a convention, D0 is used to pass
      // return values)
      MOVE #5, RETURN_VALUE_REGISTER
}
      // Freeing up the stack space taken by local variables
      MOVE FRAME_POINTER, STACK_POINTER
      POP FRAME_POINTER
      // Return back to the calling function
      RETURN_FROM_SUBROUTINE

void CalledFunction(int param1, int param2)
{
   int local1, local2;
      // Reserving space for local1, local2 (4 bytes each)
      PUSH FRAME_POINTER
      MOVE STACK_POINTER, FRAME_POINTER
      ADD #-8, STACK_POINTER
   local1 = param2;
      MOVE 12(FRAME_POINTER), -4(FRAME_POINTER)
}
      // Freeing up the stack space taken by local variables
      MOVE FRAME_POINTER, STACK_POINTER
      POP FRAME_POINTER
      // Return back to the calling function
      RETURN_FROM_SUBROUTINE
```

Function calling sequence

The generated assembly code is best understood by tracing through the invocation of CalledFunction() from CallingFunction().

Pushing parameters

CallingFunction() pushes values 2 followed by 1 on the stack. These values correspond to param2 and param1 respectively. (Note that pushing order is reverse of the declaration order.). This is implemented by the PUSH instruction. The PUSH instruction pre-decrements the STACK_POINTER register and then copies the value to the address pointed to by the STACK_POINTER.

```
CalledFunction(1,2);
      // Pushing the second parameter on the stack
      PUSH #2
      // Pushing the first parameter on the stack
      PUSH #1
```

parameters.asm.cpp hosted with ❤ by GitHub view raw

Address	Stack contents	Pointing Registers	Notes
0x00010020	2		Second parameter passed to CalledFunction
0x0001001C	1	STACK_POINTER	First parameter passed to CalledFunction

Invoke function

CallingFunction() invokes the CalledFunction() by the CALL_SUBROUTINE instruction. CALL_SUBROUTINE pushes the return address on the stack and transfers control to CalledFunction().

```
// Calling the CalledFunction()
      CALL_SUBROUTINE _CalledFunction
```

called.asm.cpp hosted with ❤ by GitHub view raw

Address	Stack contents	Pointing Registers	Notes
0x00010020	2		Second parameter passed to CalledFunction
0x0001001C	1		First parameter passed to CalledFunction
0x00010018	Return address into CallingFunction()		Address of the next instruction in CallingFunction that should be executed when CallingFunction returns

Setup the frame pointer and allocate space for local variables

CalledFunction() sets up the stack after invocation. This involves allocating space for local varialbles and setting up the frame pointer:

- Saves the CallingFunction()'s FRAME_POINTER register on the stack with the PUSH statement.
- Copies the STACK_POINTER register into the FRAME_POINTER register.
- Decrements the stack pointer by 8 to create space for the local variables local1 and local2.

```
void CalledFunction(int param1, int param2)
{
   int local1, local2;
      // Reserving space for local1, local2 (4 bytes each)
      PUSH FRAME_POINTER
      MOVE STACK_POINTER, FRAME_POINTER
      ADD #-8, STACK_POINTER
```

frame-pointer.asm.cpp hosted with ❤ by GitHub view raw

Address	Stack contents	Pointing Registers	Notes
0x00010020	param2 (2)		Second parameter passed to CalledFunction
0x0001001C	param1 (1)		First parameter passed to CalledFunction
0x00010018	Return address into CallingFunction()		Address of the next instruction in CallingFunction that should be executed when CallingFunction returns
0x00010014	FRAME_POINTER register of the CallingFunction()	FRAME_POINTER	The frame pointer of the CalledFunction has been pushed on the stack. The STACK_POINTER is then copied into the FRAME_POINTER register. This defines the frame pointer for the CalledFunction.
0x00010010	local1		Space allocated to local1 variable
0x0001000C	local2	STACK_POINTER	Space allocated to local2 variable

Accessing parameters and local variables with frame pointer offsets

Code in the CalledFunction() accesses passed parameters by taking positive offsets from the frame pointer. Local variables are accessed by taking negative offsets from the frame pointer. The example presented here shows the code for param2 assignment to local1.

```
    local1 = param2;
        MOVE 12(FRAME_POINTER), -4(FRAME_POINTER)
```

local.asm.cpp hosted with ❤ by **GitHub** **view raw**

Address	Frame pointer relative addressing	Stack contents	Pointing Registers	Notes
0x00010020	FRAME_POINTER+12	param2 (2)		Second parameter passed to CalledFunction
0x0001001C	FRAME_POINTER+8	param1 (1)		First parameter passed to CalledFunction
0x00010018		Return address into CalledFunction()		Address of the next instruction in CallingFunction that should be executed when CallingFunction returns
0x00010014		FRAME_POINTER register of the CallingFunction()	FRAME_POINTER	The frame pointer of the CalledFunction has been pushed on the stack. The STACK_POINTER is then copied into the FRAME_POINTER register. This defines the frame pointer for the CalledFunction.
0x00010010	FRAME_POINTER-4	local1		Space allocated to local1 variable
0x0001000C	FRAME_POINTER-8	local2	STACK_POINTER	Space allocated to local2 variable

Free local variables from stack and restore the caller's frame pointer

Before the function returns, the stack setup at the start of the function has to be undone. This is accomplished by the following steps:

- Copy the FRAME_POINTER register into the STACK_POINTER register. This will free the stack entries allocated for local variables local1 and local2.
- Pop the saved frame pointer from the stack. (This will make sure that the CallingFunction() gets its original frame pointer value on return).

```
  // Freeing up the stack space taken by local variables
        MOVE FRAME_POINTER, STACK_POINTER
        POP FRAME_POINTER
```

free-local.asm.cpp hosted with ❤ by **GitHub** **view raw**

Address	Stack contents	Pointing Registers	Notes
0x00010020	2		Second parameter passed to CalledFunction
0x0001001C	1		First parameter passed to CalledFunction
0x00010018	Return address into CallingFunction()		Address of the next instruction in CallingFunction that should be executed when CallingFunction returns

Return back to the caller

The processor now executes the RETURN_FROM_SUBROUTINE instruction. This instruction pops the return address from the stack and transfers control to the CallingFunction() at this address.

```
   // Return back to the calling function
        RETURN_FROM_SUBROUTINE
```

return.asm.cpp hosted with ❤ by **GitHub** **view raw**

Address	Stack contents	Pointing Registers	Notes
0x00010020	2		Second parameter passed to CalledFunction
0x0001001C	1		First parameter passed to CalledFunction

Caller pops parameters

The CallingFunction() now pops the parameters that were passed to the CalledFunction(). This is done by adding 8 to the stack pointer.

```
    // Pop out the parameters after return
        ADD #8, STACK_POINTER
```

caller-pops.asm.cpp hosted with ❤ by **GitHub** **view raw**

Address Stack contents Pointing Registers Notes

Explore more

- Understand assembly code generation for while loop, for loop, structure access and array indexing in C
- Learn how if-else and switch statements are implemeted in assembly
- Map C++ code constructs into C code

EventStudio

- call flow gallery
- sequence diagrams
- use cases & more
- testimonials
- download free trial

VisualEther

- Wireshark gallery
- visualize Wireshark
- auto diagnose
- select fields
- download free trial

Telecom+networking

- 5G NR tutorials and call flows
- 5G NR blog
- LTE tutorials and call flows
- Long Term Evolution blog
- TCP/IP flows | blog

Software Design

- object oriented design
- design patterns
- embedded design
- fault handling
- Software Design blog

Follow

- medium
- twitter
- linkedin
- facebook

Company

- contact us
- blog