50.002 COMPUTATIONAL STRUCTURES

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Procedures

Natalie Agus (Fall 2018)

1 Overview

In this notes, we are going to go through the standard steps on how β assembler take apart assembler instructions like ADDC, ADD, JMP, SHLC, etc into 32-bit machine language and puts them in the memory as a *stack*, and finally execute the instructions accordingly.

2 From C-code to Assembly Language to 32-bit Machine Language

The main goal is to translate the C-code (or any high-level language basically) into assembly language, then into 32-bit machine code, and understand how it works in the main memory. Lets dive right away into examples. Take for example this simple C-code:

```
int square(int n){
return n*n;
}

int main(){
square(4)
}
```

This is a simple code that return you the squared value of the argument: n. Upon running this C program, whatever is within main() function is by default executed by our computer. In this particular example, square(4) function is called.
It gets translated in the assembly language into a whole lot more complicated code:

```
CMOVE(4,R1)
PUSH(R1)
```

```
BR(square,LP)
DEALLOCATE(1)
HALT()

square: PUSH(LP)
PUSH(BP)
MOVE(SP,BP)

PUSH(R2)
LD(BP,-12,R2)
MUL(R2,R2,R0)

POP(R2)
MOVE(BP,SP)
POP(BP)
POP(LP)
JMP(LP)
```

Before we understand what each line means or how do we even translate from the C-code to this assembly language, we can at least convert them into 32-bit machine language. The above code is **instructions**, and they exist in the memory. Each line of the assembly language code above are **consecutively added in the memory 'rows'**. So the memory, they look something like the figure in the next page.

Counting how many words (32-bit lines) is the assembly language code:

1. The assembly language can consist of **macros** like PUSH and POP, CMOVE, etc.
2. Most macros is 1 operation each, **however PUSH and POP is 2 operations each**.
3. So the first step you should do is to **identify the PUSH and POP macros**, and convert them into two original β instructions as written in the β summary instruction set.
4. Therefore **in total for this example, we have 23 lines of 32-bit instructions, from address 0x0000 0000 to 0x0000 0058**.

3 Housekeeping Initial Values Example

Before we dive into the procedure to write assembly language, there are some special registers we have to keep in mind. Recall that R27 to R29 are special registers that house special pointers. **Assume for this example that the values of these registers are set to:**

1. R27 / BP: 0x0000 1000

Memory

Address	Content	Remarks
0x0000 0000	0xC03F 0004	This is CMOVE(4,R1), which is a macro for ADDC(R31, 4, R1)
0x0000 0004	0xC3BD 0004	These two lines is PUSH(R1), which is a macro for two operations: ADDC(SP,4,SP) and ST(R1,-4,SP)
0x0000 0008	0x643D FFFC	
0x0000 000C	0x779F 0002	This is BR(square, LP), which is a macro for BEQ(R31, square, LP). Note that 0x0000 0018 is the address of label 'square'.
0x0000 0010	0xC7BD 0004	This is DEALLOCATE(1) which is a macro for SUBC(SP, 4*1, SP)
0x0000 0014	0x0000 0000	This means HALT(), i.e: stop program
0x0000 0018	0xC3BD 0004	These two lines is PUSH(LP). Note that 0x0000 0018 is the address of label 'square'
0x0000 001C	0x679D FFFC	
0x0000 0020	0xC3BD 0004	These two lines is PUSH(BP)
0x0000 0024	0x677D FFFC	
0x0000 0028	0x837D F800	This is MOVE(SP, BP), which is a macro for ADD(SP, R31, BP)
0x0000 002C	0xC3BD 0004	These two lines is PUSH(R2)
0x0000 0030	0x645D FFFC	
0x0000 0034	0x605B FFF4	This is LD(BP, -12, R2)
0x0000 0038	0x8802 1000	This is MUL(R2, R2, R0)
0x0000 003C	0x605D FFFC	These two lines is POP(R2), which is a macro for LD(SP, -4, R2) and then ADDC(SP, -4, SP)
0x0000 0040	0xC3BD FFFC	
0x0000 0044	0x83BB F800	This is MOVE(BP, SP), which is a macro for ADD(BP, R31, SP)
0x0000 0048	0x637D FFFC	These two lines is POP(BP)
0x0000 004C	0xC3BD FFFC	
0x0000 0050	0x639D FFFC	These two lines is POP(LP)
0x0000 0054	0xC3BD FFFC	
0x0000 0058	0x6FFC 0000	This is JMP(LP)
...		
....		
....		

Figure 1

2. R28 / LP: 0x0000 0000

3. R29 / SP: 0x0000 1000

4. R30 / XP: 0x0000 0000

Assume also the prior content of register R2 is 0x1234 5678.

4 Writing Assembly Language Procedure

The procedure for converting from high level code like C-code into the assembly language is divided into 5 steps. **Please pay attention to the word 'standard' in the following sections**. They are standard procedures that need to always be done in each step.

5 Writing Assembly Language Step 1: Calling Sequence - Arguments

The first part of the code that the assembler will write is to push arguments into the stack **in the reverse order, meaning push** arg_n**, then** arg_{n-1}**, ...** arg_2**, and finally the first argument** arg_1. In Java, C, Python, etc, we are familiar with **function arguments**. Before we can even execute the rest of the function codes, the machine need to process the arguments first and load them into memory **in terms of stack** for later access. So in this example, we call `square(4)`, meaning that the argument for this function call is 4. The first two lines of assembly code is written for loading arguments:

```
CMOVE(4,R1)
PUSH(R1)
```

Writing arguments into memory is not so straightforward. We need to first write a constant into the register using CMOVE, and then push it into the memory. Hence, CMOVE(4, R1) writes the value of constant 4 to register R1, and then ”push” the content of R1 into the memory with address SP: 0x0000 1000.

6 Writing Assembly Language Step 2: Calling Sequence - Branching and Cleanup

The next piece of code that the assembler will write is to call the function itself using BR (in this example, is, square), and then write cleanup codes, before doing HALT():

```
BR(square,LP)
DEALLOCATE(1)
HALT()
```

When the code BR(square, LP) is executed by the PC, this causes the PC to jump to the address of ’square’, which is 0x0000 0018. However, it also stores its **last known location + 4 bytes** : 0x0000 0010 at register R28 / LP. When this code is executed, register R28 / LP’s content will be 0x0000 0010.

After calling the function ’square’ with BR, the assembler will also **always write standard ’cleanup’ codes** to clear the memory stack, which is **DEALLOCATE**. Since there’s only 1 argument pushed to the stack, we only need to DEALLOCATE(1). Finally, the assembler will **always write** the **HALT()** and this marks the end of the Calling Sequence.

7 Writing Assembly Language Step 3: Standard Entry Sequence

After the assembler finishes writing the Calling sequence, it will begin writing the Entry sequence, which is the function itself, and in this case, it is 'square'. There are **three standard things** that should **always** be done in the beginning of entry sequence:

```
PUSH(LP)
PUSH(BP)
MOVE(SP,BP)
```

The first two lines 'stores' the previous contents of Registers R28 / LP and Registers R27 / BP into the stack. Then, it moves the BP to the SP, which indicates the beginning (BP) of the stack for this function 'square'. (Moving BP to the SP means to write the content of register R29 / SP into register R 27 / BP)

8 Writing Assembly Language Step 4: The Actual Code

The assembler now can begin writing the actual content of the code, which is n*n:

```
PUSH(R2)
LD(BP,-12,R2)
MUL(R2,R2,R0)
```

In this case, the assembler chose to use R2 as temporary space to do calculation MUL. **As a practice, the assembler will push the old value of R2 to the stack before using it**, hence PUSH R2. Then it loads the **first argument** into R2: **The address of the first argument is ALWAYS BP - 12**. Finally, it multiplies the content of R2 by itself (squaring it), and store its result in R0. **R0 is reserved register to keep the return value**.

9 Writing Assembly Language Step 5: Exit Sequence - Pop Regs from Actual Code

As a practice, the assembler always clean up the stack after its done with the task specified by the code. It will pop the stack in **reverse order**, so first it pops whatever register that was used **in the actual content of the code**, in this case, it is just R2:

```
POP(R2)
```

The line above means to pop the content at the top of the stack to R2. It effectively restores R2 to its original value before R2 was used to temporarily do the computations in the code: in this example, R2 was temporarily used to hold the argument value for later squaring (MUL).

10 Writing Assembly Language Part 6: Exit Sequence - Standard Exit Sequence

Then it will **always do** the following **standard exit sequence**:

```
MOVE(BP,SP)
POP(BP)
POP(LP)
JMP(LP)
```

MOVE(BP, SP) transfer the content of the base pointer (register R27 / BP) to the stack pointer (register R29 / SP). And then, it POP(BP): move the content of the top of the stack to register BP, and reduce the content in Register R29 / SP by 4 bytes. Then it also POP(LP): move the content of the current top of the stack to register R28 / LP (store the content of the top of the memory stack to R28 / LP). Finally, it executes back the address of the instruction as pointed by the content of register R28 / LP, i.e: execute instruction at Mem[Reg LP].

11 Running the Assembly Code

So after all these codes are written by the assembler, and nicely loaded into the memory in terms of 32-bits instructions as shown in Figure 3, they are run by the CPU's PC. **The PC is the 'pointer' of the CPU that executes instruction from the memory**.

The machine will set the PC to execute the first line of the instruction.

In this example, this means that the PC value begins at 0x0000 0000

The following shows what the PC does step by step to execute the codes:

1. The initial value of PC is 0x0000 0000. This means to execute the instruction 0xC03F 0004 CMOVE(4, R1). Recall for all β instructions, PC is always increased by 4 after the instruction is done (which means to move to the next line of instruction).

   Notable state changes:

   (a) R1 content is now 0x0000 0004

2. PC is now at 0x0000 0004 after executing CMOVE(4,R1). At this address, the instruction is 0xC3BD 0004 : ADDC(SP, 4, SP), which is to increase the content of R29 / SP by 4 and store it back at SP, and then increase the PC by 4.

   Notable state changes:

   (a) R29 / SP content is now 0x0000 1004

3. PC is now at 0x0000 0008 after executing ADDC(SP, 4, SP). At this address, the instruction is 0x643D FFFC : ST(R1, -4, SP), which is to store the content of R1 to the address : SP - 4, and then increase the PC by 4.

   Notable state changes:

   (a) The 'stack' is basically a part of memory data, now the 'stack' only has 1 item, and the block of memory containing the stack looks like this:

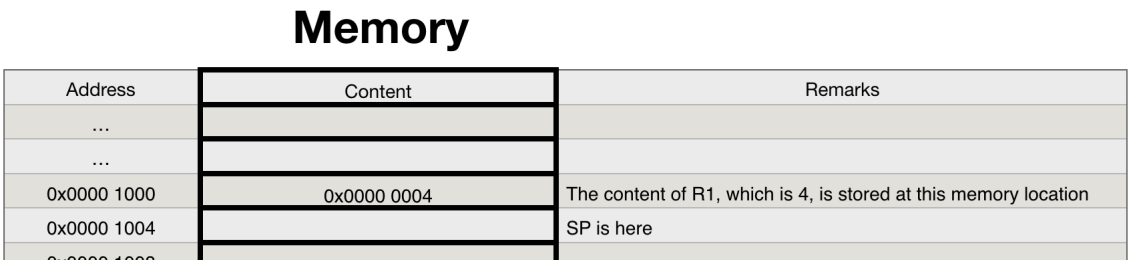

Figure 2

   (b) We count the number of 'items' in the stack from the original Reg SP location (which is 0x0000 1000) before the PC executes any instructions for this code.

4. PC is now at 0x0000 000C. At this address, the instruction is 0x779F 0002 : BEQ(R31, square, LP). This means to store the value of PC + 4 to register R28 / LP if R31 is equal to zero (which is always true since R31 is reserved register to 0), and then move PC to the address of 'square'.

   Notable state changes:

   (a) R28 / LP content is now 0x0000 0010

   (b) PC is now set to 0x0000 0018

   Explanation on how to get the 32-bit instruction for BEQ fast (or any instruction that requires literal computation of 'label':

   (a) Count how many lines of instructions are there between BR and the **first line of instruction of the function 'label'** (not including BR but including that first line of instruction of function 'label')

   (b) Remember, theres **2 lines of instructions for each PUSH or POP, and 1 line of instruction for each of every other β instruction**.

   (c) So the first instruction of function 'square' is actually 3 lines away from BR(Square, LP), 1 line from DEALLOCATE(1), 1 line from HALT(), and 1 line from the first instruction of PUSH - which is ADDC.

(d) The literal (in the 32-bit machine code) is this number (the number of instructions from BR to the first instruction of the function 'square') **subtracted by 1**,

(e) Hence the literal is 2.

(f) Express the literal in 16 bits: 0000 0000 0000 0010

(g) The OPCODE for BEQ is 011101

(h) Ra is R31: 11111

(i) LP is R28: 11100

(j) The 32-bit instruction is therefore 011101 11100 11111 0000 0000 0000 0010

(k) Convert to hex: 0111 0111 1001 1111 0000 0000 0000 0010 = 0x779F 0002

5. PC is now at 0x0000 0018. At this address, the instruction is 0xC3BD 0004 : ADDC(SP,4,SP), which is to increase SP by 4 and store it back at SP, and then finally increase PC by 4.

   Notable state changes:

   (a) R29 / SP content is now 0x0000 1008

6. PC is now at 0x0000 001C. At this address, the instruction is 0x679D FFFC : ST(LP, -4, SP), which is to store the content of register R28 / LP to the address : SP - 4, and then increase the PC by 4.

   Notable state changes:

   (a) Now the 'stack' has 2 items (2 PUSH code so far), and the block of memory containing the stack currently looks like this:

Memory

Address	Content	Remarks
...		
...		
0x0000 1000	0x0000 0004	The content of R1, which is 4, is stored at this memory location
0x0000 1004	0x0000 0010	The content of R29 / LP is stored at this memory location
0x0000 1008		SP is here
0x0000 100C		

Figure 3

7. PC is now at 0x0000 0020. At this address, the instruction is 0xC3BD 0004 : ADDC(SP, 4, SP), which is to increase SP by 4 and store it back at SP, and then finally increase PC by 4.

   Notable state changes:

(a) R29 / SP content is now 0x0000 100C

8. PC is now at 0x0000 0024. At this address, the instruction is 0x677D FFFC : ST(BP, -4, SP), which is to store the content of register R27 / BP to the address : SP - 4, and then increase the PC by 4.

   Notable state changes:

   (a) Now the 'stack' has 3 items (3 PUSH code so far), and the block of memory containing the stack currently looks like this:

Memory

Address	Content	Remarks
...		
...		
0x0000 1000	0x0000 0004	The content of R1, which is 4, is stored at this memory location
0x0000 1004	0x0000 0010	The content of R29 / LP is stored at this memory location
0x0000 1008	0x0000 0000	The content of R27 / BP is stored at this memory location
0x0000 100C		SP is here
0x0000 0010		

Figure 4

9. PC is now at 0x0000 0028. At this address, the instruction is 0x837D F800 : ADD(BP, R31, SP), which is to add the content of register BP by 0 and store it as the content of register SP, and then finally increase PC by 4.

   Notable state changes:

   (a) R27 / BP content is now 0x0000 100C

   (b) The state of the stack is,

Memory

Address	Content	Remarks
...		
...		
0x0000 1000	0x0000 0004	The content of R1, which is 4, is stored at this memory location
0x0000 1004	0x0000 0010	The content of R29 / LP is stored at this memory location
0x0000 1008	0x0000 0000	The content of R27 / BP is stored at this memory location
0x0000 100C		SP is here, BP is here
0x0000 0010		

Figure 5

10. PC is now at 0x0000 002C. At this address, the instruction is 0xC3BD 0004 : ADDC(SP, 4, SP), which is to increase SP by 4 and store it back at SP, and then finally increase PC by 4.

Notable state changes:

(a) R29 / SP content is now 0x0000 1010

11. PC is now at 0x0000 0030. At this address, the instruction is 0x645D FFFC : ST(R2, -4, SP), store the content of R2 (as set for example, the content of R2 happens to be 0x1234 5678) to the address: SP - 4, and finally increase the PC by 4.

    Notable state changes:

    (a) Now the 'stack' has 4 items (4 PUSH code so far), and the block of memory containing the stack currently looks like this:

Memory

Address	Content	Remarks
...		
...		
0x0000 1000	0x0000 0004	The content of R1, which is 4, is stored at this memory location
0x0000 1004	0x0000 0010	The content of R29 / LP is stored at this memory location
0x0000 1008	0x0000 0000	The content of R27 / BP is stored at this memory location
0x0000 100C	0x1234 5678	The content of R2, is stored at this memory location
0x0000 0010		SP is here

Figure 6

    (b) Note R27 / BP content is still 0x0000 100C, although not shown in diagram.

12. PC is now at 0x0000 0034. At this address, the instruction is 0x605B FFF4 : LD(BP, -12, R2), store the content of Mem[BP - 12] to R2, and finally increase the PC by 4.

    Notable state changes:

    (a) Reg 27 / BP content is 0x0000 100C, and
    (b) BP - 12 = 0x0000 1000
    (c) The content of Mem[0x0000 1000] is 0x0000 0004 (see the stack figure 7), which is simply the value of the first argument
    (d) **So BP-12 is always the address of the first argument**
    (e) **If there's a second, third, etc argument, then the address of it in the stack will be BP-16, BP-20, etc**.
    (f) The content of R2 is now the content of Mem[0x0000 1000] which is 0x0000 0004.

13. PC is now at 0x0000 0038. At this address, the instruction is 0x8802 1000: MUL(R2, R2, R0), which is to multiply the content of R2 by the content of R2, and store it at R0, and finally increase the PC by 4. Notable state changes:

(a) We multiply the content of R2 by itself: 0x0000 0004 * 0x0000 0004 = 0x0000 0010 (16).

(b) R0 content is now 0x0000 0010

14. PC is now at 0x0000 003C. At this address, the instruction is 0x605D FFFC: LD(SP, -4, R2), which is to load the content of memory with address : SP - 4 to R2, and finally increase the PC by 4.

    Notable state changes:

    (a) R2 content is now the content of Mem[0x0000 1010 - 4] = 0x1234 5678

    (b) This simply means we restore back the **original value** of R2 before 'square' is executed.

15. PC is now at 0x0000 0040. At this address, the instruction is 0xC3BD FFFC: ADDC(SP, -4, SP), which is subtract SP by 4 and store it back at SP, and finally increase the PC by 4.

    Notable state changes:

    (a) Reg SP content is now 0x0000 100C

    (b) Now the 'stack' has 3 items (4 PUSH code and 1 POP code so far) and the block of memory containing the stack currently looks like this:

Memory

Address	Content	Remarks
...		
...		
0x0000 1000	0x0000 0004	The content of R1, which is 4, is stored at this memory location
0x0000 1004	0x0000 0010	The content of R29 / LP is stored at this memory location
0x0000 1008	0x0000 0000	The content of R27 / BP is stored at this memory location
0x0000 100C	0x1234 5678	SP is here, BP is here
0x0000 0010		

Figure 7

    (c) Note how the content 0x1234 5678 in address 0x0000 100C isn't exactly 'removed' from the memory. One simply moves the SP there which means that the next time a PUSH operation is called, a new value will be written here, which is as good as an 'empty' slot in the memory.

16. PC is now at 0x0000 0044. At this address, the instruction is 0x83BB F800: ADD(BP, R31, SP), which to add the content of register BP by zero and store it at register SP, and finally increase the PC by 4.

    Notable state changes:

    (a) Reg 29 / SP content is now 0x0000 100C

17. PC is now at 0x0000 0048. At this address, the instruction is 0x637D FFFC: LD(SP, -4, BP), which is to load the content with memory address : SP - 4 to Reg BP, and then increase the PC by 4. Notable state changes:

    (a) Reg 27 / BP content is now 0x0000 0000, which is restored to its original value before 'square' is executed.

18. PC is now at 0x0000 004C. At this address, the instruction is 0xC3BD FFFC : ADDC(SP, -4, SP), which is subtract SP by 4 and store it back at SP, and finally increase the PC by 4.

    Notable state changes:

    (a) Reg SP content is now 0x0000 1008

    (b) Now the 'stack' has 2 items (4 PUSH code and 2 POP code so far) and the block of memory containing the stack currently looks like this:

Memory

Address	Content	Remarks
...		
...		
0x0000 1000	0x0000 0004	The content of R1, which is 4, is stored at this memory location
0x0000 1004	0x0000 0010	The content of R29 / LP is stored at this memory location
0x0000 1008	0x0000 0000	SP is here
0x0000 100C	0x1234 5678	
0x0000 0010		

Figure 8

19. PC is now at 0x0000 0050. At this address, the instruction is 0x639D FFFC: LD(SP, -4, LP), which is to load the content with memory address : SP - 4 to Reg LP, and then increase the PC by 4. Notable state changes:

    (a) Reg 28 / LP content is now 0x0000 0010, which is restored to its original value before 'square' is executed.

20. PC is now at 0x0000 0054. At this address, the instruction is 0xC3BD FFFC : ADDC(SP, -4, SP), which is subtract SP by 4 and store it back at SP, and finally increase the PC by 4.

    Notable state changes:

    (a) Reg SP content is now 0x0000 1004

    (b) Now the 'stack' has 1 item (4 PUSH code and 3 POP code so far) and the block of memory containing the stack currently looks like this:

21. PC is now at 0x0000 0058. At this address, the instruction is 0x6FFC 0000 : JMP(LP, R31), meaning to move PC to the content of register LP. Notable state changes:

Memory

Address	Content	Remarks
...		
...		
0x0000 1000	0x0000 0004	The content of R1, which is 4, is stored at this memory location
0x0000 1004	0x0000 0010	SP is here
0x0000 1008	0x0000 0000	
0x0000 100C	0x1234 5678	
0x0000 0010		

Figure 9

(a) PC is now the content of Reg 28 / LP which is 0x0000 0010

22. PC is now at 0x0000 0010 after executing JMP in the previous step. This was the last PC + 4 known location before it executes BR(square, LP) back in the caller sequence. At this address, the instruction is 0xC7BD 0004 : SUBC(SP, 4*1, SP), meaning to subtract the content of Reg SP by 4 and store it back at Reg SP, and finally increase the PC by 4.

    Notable state changes:

    (a) Reg SP content is now 0x0000 1000, which is its original value before 'square' is executed.

    (b) Now the 'stack' has 0 item (4 PUSH code and 4 POP code so far) because the SP goes back to its original value of 0x0000 0001. The block of memory containing the stack currently looks like this:

Memory

Address	Content	Remarks
...		
...		
0x0000 1000	0x0000 0004	SP is here
0x0000 1004	0x0000 0010	
0x0000 1008	0x0000 0000	
0x0000 100C	0x1234 5678	
0x0000 0010		
0x0000 0014		

Figure 10

23. PC is now at 0x0000 0014. At this address, the instruction is 0x0000 0000. This simply means HALT() or end program.

24. Notice how the content of R0: 0x0000 0010 (16) **is indeed the return value** of the function 'square': square(4).

12 Summary of procedures you NEED to remember

1. PUSH and POP are just convenient macros for TWO instructions

2. R0 is reserved as the register for the return value of the function getting called
3. R27 and R30 are special registers
4. Always PUSH arguments in REVERSE order, meaning push argument n, then argument n-1, then argument n-2, ..., and finally argument 1. This is so that it is easy to access the arguments: BP - 12 is ALWAYS the first argument, and we can compute offsets from BP - 12 onwards to find the subsequent arguments.
5. Pay attention to all **standard procedures**, summarized below:

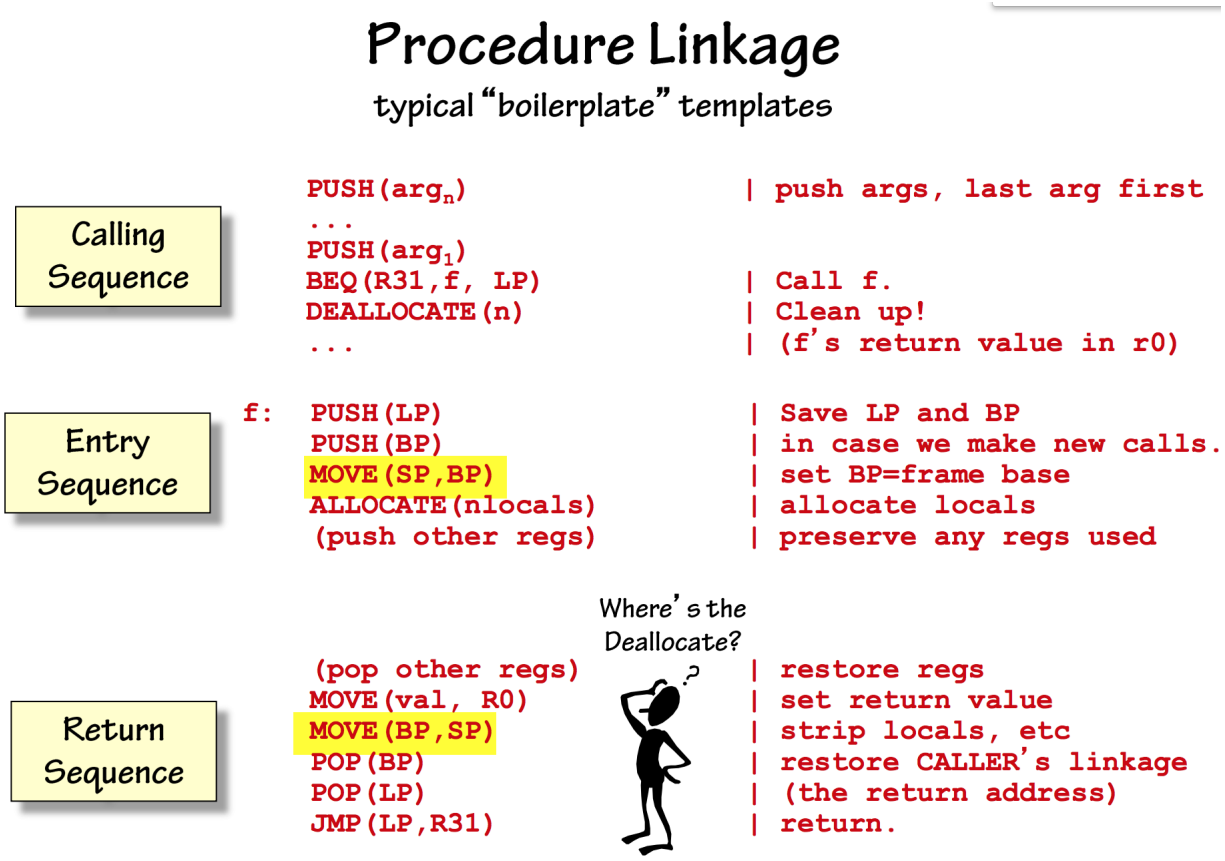

Figure 11

Answer to the question 'Where's the deallocate'? It is up there at the calling sequence after BEQ.

Calling sequence has two parts: pushing arguments in reverse order, BEQ to the function label 'f', and then cleanup.

Entry sequence has three standard lines, PUSH(LP), PUSH(BP), and MOVE(SP,BP).

You *may* allocate (with ALLOCATE(x)) and reserve some space (x-lines) in the memory if you want (in this example we dont do that). And then Push registers that you are going to use in the main code f to preserve its old value. The return sequence pops the registers you pushed in the main code f to restore them back into their old values (pop the latest pushed register first). You *may* MOVE(val, R0) IF you need to move the return value to another Register 'val'. In this example however we don't need to do that.

Finally, **the four standard exit sequence** is MOVE(BP, SP), POP(BP), POP(LP), and JMP(LP).

6. A memory contains both instructions (color coded in red above) and stack (or data, color coded in black).
7. Registers are in the CPU, not part of the memory (RAM)
8. We only have limited registers, and the ALU (logic unit) can only access registers to perform computations, so we need **stack** in the memory, i.e: a temporary space to perform computation and execute function code.
9. Each time you call a function (this case, is function `square`), the stack grows (arguments pushing, entry sequence, and registers pushing)
10. When the function returns (or ends), the stack diminishes (exit sequence, and pops)
11. When you PUSH(Rx), you are storing the CONTENT of Rx into the memory. Where in the memory? To the address pointed by SP
12. When you POP(Rx), you are loading the CONTENT of the memory with address pointed by SP back to register Rx