50.002 COMPUTATIONAL STRUCTURES

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Stacks

Natalie Agus (Fall 2018)

1 Overview

This notes explain briefly the function of *stacks* during function call.

2 Function Call

Take a look at C-code snippet below:

```
int multiply(int a, int b){
int answer = a*b;
return answer;
}

int add(int a, int b){
int answer = a + b;
return answer;
}

int main(){
int res = multiply(2,5);
res = add(res, res);
}
```

These codes are translated into 32-bit machine **instructions** and then stored in the RAM to be executed by the program counter (PC), sequentially (line by line, top to bottom). In this simple code, when it is run (C will always perform whatever task written in the `main`), the program will multiply 2 by 5 and store it at `res`. Afterwards, it doubles the result. The final value is `res = 20`.

It is easy to trace the function calls in this high-level C-language. At first, we call the function `multiply` with arguments 2 and 5, and store the result at `res`. Then, we call the function `add` with argument `res`, and store back the value at `res`.

Notice that within each function `multiply` and `add`, there's a local variable `answer` and each function perform some sort of logic operation (multiply, and add). Recall that these operations can be done by the ALU, inside the CPU. Both functions require temporary storage to perform its code before returning the answer back to `main`. These **temporary** tasks are done using a data structure called **stack**. Both **stacks** and **instructions** are located in the RAM.

In the next note we are going to learn the **procedure** of function calling : i.e: code execution. Stack is created using procedure during function calling and cleaned up / shrinks once the function exits. So in the example above, a stack will grow when the `main` function calls `multiply` and shrinks when `multiply` returns. Similarly, it will grow again when the `add` function begins and shrinks when it returns.

3 Translating from C to assembly language

The C-code above is translated to a more primitive assembly language by the compiler / interpreter as follows:

```
.include beta.uasm


res = 0x120
. = 0x0000

ALLOCATE(100)

CMOVE(2, R1)
CMOVE(5, R2)
PUSH(R2)
PUSH(R1)
BR(multiply, LP)
DEALLOCATE(2)
PUSH(R0)
PUSH(R0)
BR(add, LP)
DEALLOCATE(2)
ST(R0, res)

DEALLOCATE(100)

multiply: PUSH(LP)
```

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

PUSH(R3)
PUSH(R4)
LD(BP, -12, R3)
LD(BP, -16, R4)
MUL(R3, R4, R0)
POP(R4)
POP(R3)

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

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

PUSH(R3)
PUSH(R4)
LD(BP,-12, R3)
LD(BP, -16, R4)
ADD(R3, R4, R0)
POP(R4)
POP(R3)

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

The next note will guide you on how to write the assembly language and its procedures. For now, things that you can pay attention to are:

1. The first line simply means we name memory address `0x00001234` as *res*
2. The second line means that we begin (storing the instructions below) at address $0x00000000$. That is what (dot) means (where is the next line stored in memory?)
3. That is, `ALLOCATE(100)` is at $0x00000000$
4. The PC will execute from the first line of memory onwards (in this case, its `ALLOCATE(150)` which is exactly what we want.

5. Whenever `BR`, this means to execute the function as specified by the first argument of `BR`. One has to push argument to the stack before BR is called, and DEALLOCATE once it is done.

6. Each function, multiply and add, has its own entry sequence and exit sequence

An alternative is to ”load” multiply and add first into a specific memory location below using (dot), and then redirect (dot) at 0x00000000 for PC to execute `ALLOCATE(100)`.

```
res = 0x120
. = 0xA4

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

PUSH(R3)
PUSH(R4)
LD(BP, -12, R3)
LD(BP, -16, R4)
MUL(R3, R4, R0)
POP(R4)
POP(R3)

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

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

PUSH(R3)
PUSH(R4)
LD(BP,-12, R3)
LD(BP, -16, R4)
ADD(R3, R4, R0)
POP(R4)
POP(R3)

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

. = 0x000
```

```
ALLOCATE(100)

CMOVE(2, R1)
CMOVE(5, R2)
PUSH(R2)
PUSH(R1)
BR(multiply, LP)
DEALLOCATE(2)
PUSH(R0)
PUSH(R0)
BR(add, LP)
DEALLOCATE(2)
ST(R0, res)

DEALLOCATE(100)
```

Tryout the code above using BSIM, and observe the contents of each register step by step to understand how this works.

4 Growing and Shrinking Stack

A stack grows ONLY when PUSH instructions are executed, and shrinks ONLY when POP instructions are executed.

5 Hardware Structure

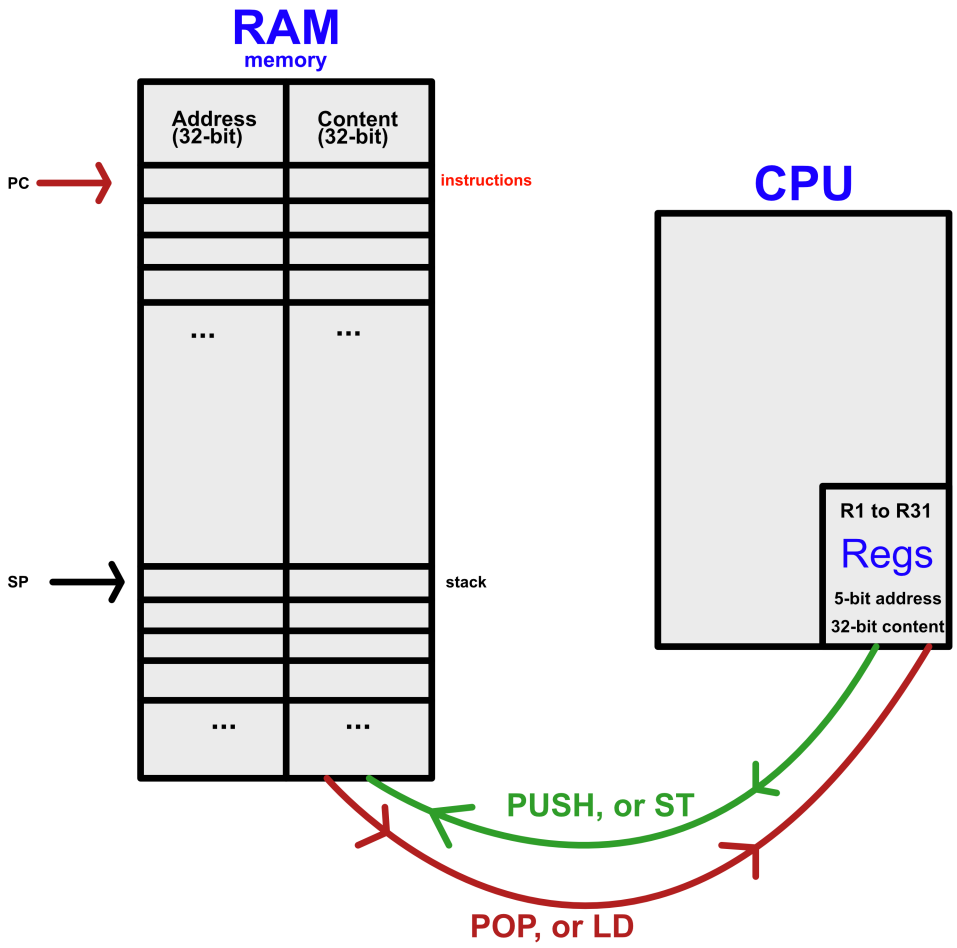

Figure 1

Take a look at the β hardware structure above. The logic unit that performs all logic operations is located inside the CPU. However the ALU can only perform these logic operations on the registers. In the β architecture, there are only 32 of them. We definitely require more space to perform logic operations and execute instructions than the space offered by these 32 registers, therefore we have the RAM (memory) to store both instructions (code like what is shown in the previous section) and data.

We can move data from registers to the memory using β instruction: STORE or PUSH, and move data from memory to registers using β instruction: LOAD or POP.

6 Main Memory Sectioning

The main memory of the β architecture can hold 32 bits contents per line. The main memory can store both data values or instructions, both in 32 bits. Typically, the assembler will arbitrarily section the main memory into two parts as follows. The 'instruction' section is where you would want to store the 32-bit β instructions. The 'data' section is where you would want to store your 'stack' during code running time.

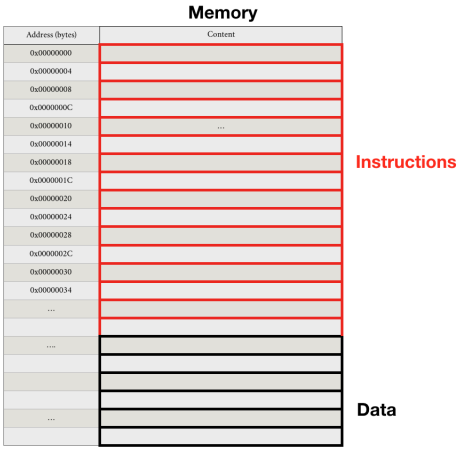

Figure 2

7 Stack and Pointer

To understand how β interprets programs and execute it in the memory block, we first must understand the notion of stack and pointers. **A pointer** is 32-bit long and it tells the machine which 32-bit address in the memory to write or read on. Take a look at Figure 1 below on how a **stack** is implemented [1]:

1. A stack is a **concept**, a data structure on how we organize i.e: store or remove data in the memory
2. You can only add one item at a time **to the top of the stack** by PUSH, and remove one item at a time **from the top of the stack** by POP
3. A stack has two pointers: SP and BP
4. The stack pointer (SP) **points to the available memory location to write to** (first unused stack space). SP is actually the content of R29. In the diagram it is drawn as the 'pointer'.
5. The base pointer (BP) **points to the base of the stack, or equivalently first item pushed to the stack**. BP is actually the content of R27. Based on the diagram, BP is 0x0000 1000.
6. There are **only two operations to modify a stack**, PUSH(Rx) and POP(Rx). PUSH grows a stack, and POP reduces a stack.
7. In the 'original state' of the diagram, the SP is set to address 0x0000 1000 at first, which is the first "empty" space to write to.
8. For example in the diagram below, we want to PUSH 3 different words (32-bit data) into the stack.
9. We first **PUSH**(red data), which is basically a two-step procedure : (1) Increase the value of SP by 4, to 0x0000 1004, and then (2) writing the red data to address 0x0000 1000 as pointed by SP - 4
10. We then similarly **PUSH**(blue data), and after this SP is now increased to 0x0000 1008
11. Finally we **PUSH**(green data), and after this SP is now increased to 0x0000 100C.
12. In β assembly language, **PUSH(Rx)** means a **two-step procedure**: (1) to write (add) the data in Reg Rx into the content of the address pointed by the SP, and then (2) increment SP by 4 bytes (to the next line). Therefore after pushing three 32-bits data, the pointer is at address 100C (12 bytes later than the original state at address 1000).

[1] In this diagram, for illustration convenience sake, the higher 4-bit of the address is omitted

13. To ”remove” the data from the memory, we perform a **POP(Rx)** operation. **POP is also a two step procedure**: we (1) store the data at the address pointed by SP - 4 (the green data) to some register Rx (not written in diagram), and then (2) reduce the value of the SP by 4 bytes (to the previous line at 1008).

14. Note that the green data is actually still there but since the pointer points to the available memory location to write to, we basically will overwrite this green data the next time we push something to the stack (as good as empty memory).

15. Summary: A stack is basically a first-in last-out procedure (just like how people enter a lift or train cabin, the first that enters the area is far back from the door, and will be the last one who exit the area).

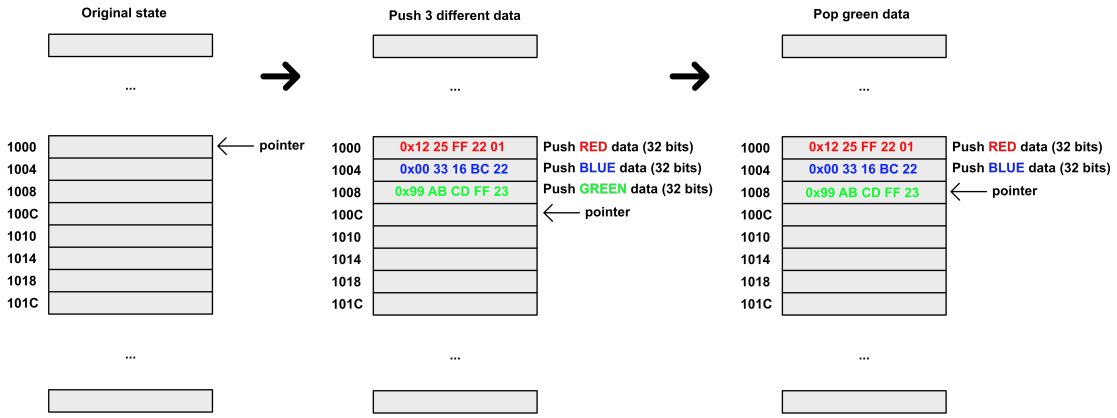

Figure 3