Function Calls –Computer Organization–

Lionel Morel

Computer Science and Information Technologies - INSA Lyon

Fall-Winter 2024-25

Introduction

- SW we have looked at so far is basically one main function
- Exception with ITs
- Most programs we write use functions
- Goal of this chapter: Understand how functions work

Why Use functions?

- **Organize code**, name instruction blocks
- Avoid code duplication, encourage **code reuse**
- Improve **readbility**
- **Limit over-nesting** control structures
- Allow **local thinking** with local variables
- Allow building **libraries** (encourage code reuse ... more)
- Prepare for Object-Oriented-Programming

Example (cont'd)

```
int PP(int x){
  int z,p;
  z = x+1;
  p = z+2;
  return (p);
}

main(){
  int i,j,k;
  i = 0;
  j = i+3;
  j = PP(i+1);
  k = PP(2 * (i+5));
}
```

- `main` is the **caller**.
- it calls `PP` which is the **callee**.
- `PP` computes an integer output, the **result** of the function.
- variables `z` and `p` are **local variables** of `PP`.
- Every time `PP` is called it's the **same code** that is executed, but with **different instances** of `z` and `p` every time.

Example (cont'd)

```
int PP(int x){
  int z,p;
  z = x+1;
  p = z+2;
  return (p);
}

main(){
  int i,j,k;
  i = 0;
  j = i+3;
  j = PP(i+1);
  k = PP(2 * (i+5));
}
```

i==0 && j == 3 && k==??

i==0 && j == 4 && k=??

i==0 && j==4 && k == 13

Example (cont'd)

```c
int PP(int x){
    int z,p;
    z = x+1;
    p = z+2;
    return (p);
}

main(){
    int i,j,k;
    i = 0;
    j = i+3;
    j = PP(i+1);
    k = PP(2 * (i+5));
}
```

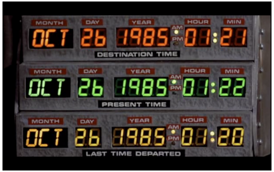
```
PP:
        SUB.W   #6, R1
        MOV.W   R12, @R1
        MOV.W   @R1, R12
        ADD.W   #1, R12
        MOV.W   R12, 4(R1)
        MOV.W   4(R1), R12
        ADD.W   #2, R12
        MOV.W   R12, 2(R1)
        MOV.W   2(R1), R12
        ADD.W   #6, R1
        RET
main:
        SUB.W   #6, R1
        MOV.W   #0, 4(R1)
        MOV.W   4(R1), R12
        ADD.W   #3, R12
        MOV.W   R12, 2(R1)
        MOV.W   4(R1), R12
        ADD.W   #1, R12
        CALL    #PP
        MOV.W   R12, 2(R1)
        MOV.W   4(R1), R12
        ADD.W   #5, R12
        ADD.W   R12, R12
        CALL    #PP
        MOV.W   R12, @R1
        MOV.B   #0, R12
        ADD.W   #6, R1
        RET
```

Problems we need to solve

- ☐ Problem #1: **Jump** from `main` to `PP` ... and then **come back**
- ☐ Problem #2: How do we make it work with **call cascades** (eg `main` calls `P`, `P` calls `Q`, etc)?
- ☐ Problem #3: How do we make it work with **recursive functions** (`P` calls itself)?
- ☐ Problem #4: Deal with **local variables** (the call to `PP` should not break anything in `main`)?
- ☐ Problem #5: How do we **pass parameters** from `main` to `PP`?
- ☐ Problem #6: How do we get **the return value** from `PP` to `main`?

Problem #1:
"Marty .... We have to go back ... to the future"

- Calling PP is "just" jumping to the address of `labelPP`.
- We need to keep track of the instruction immediatly following the call to PP
- This is called the **return address**



Call and Return

```
call labelPP
```

- Pre-condition:
  - `PC` contains the address of the `call labelPP` instruction
- Semantics:
  - Assume `PC` contains the address of the `call` instruction
  - save $\longleftarrow$ `PC`+δ for later on ... into a dedicated location
  - NB: $\delta =$ number of bytes taken by the call instruction itself
  - change `PC` to address of `labelPP`

Call and **Return**

```
ret
```

- Pre-condition:
  - at least one `call` has been executed
- Semantics:
  - copy the “saved return address” back to PC

Call and Return: example

0x20	some instruction
0x22	call myProcedure
0x24	some other instruction
0x26	yet another one

0x80	myProcedure:
0x82	...
0x84	ret

call:
1/ save 0x24
2/ jump to 0x80

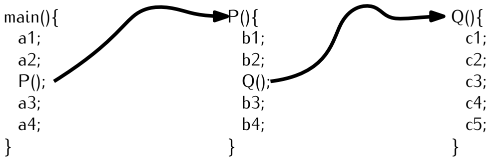
ret:
1/ jump back to 0x24

First implementation: save Return Address into **specific RA register**

Problems we need to solve

- ☑ Problem #1: **Jump** from `main` to `PP` ... and then **come back**
- ☐ Problem #2: How do we make it work with **call cascades** (eg `main` calls `P`, `P` calls `Q`, etc)?
- ☐ Problem #3: How do we make it work with **recursive functions** (`P` calls itself)?
- ☐ Problem #4: Deal with **local variables** (the call to `PP` should not break anything in `main`)?
- ☐ Problem #5: How do we **pass parameters** from `main` to `PP`?
- ☐ Problem #6: How do we get **the return value** from `PP` to `main`?

Problem #2: Call Cascades



- RA doesn't resolve the problem
- We would need several RA registers ...
- ... and we don't know how many exactly in general.

Problem #3: Recursive Calls

```
int fact(int x){
   if (x==0) { return 1;}
   else {return x * fact(x-1);}
}

int main(){
   int n,y;
   printf("give␣me␣a␣number,␣I'll␣give␣you␣its␣factorial\n");
   scanf("%d", n);
   y = fact(n);
   return 0;
}
```

- Even worse ...
- Number of calls to `fact` **depends on the value of n**
- Can't ever be predicted until user enters it, at **execution**;
- Code needs to be prepared before that, at **compilation**.

Solution: Use a stack

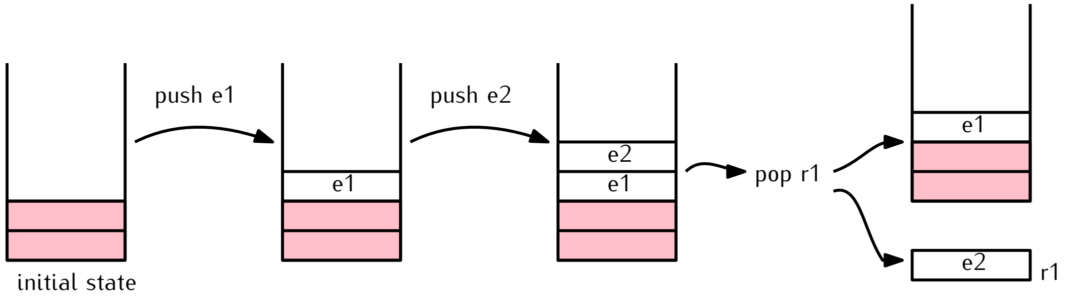
Definition (Stack)

A **Stack** is an abstract data type that allows to store items of data. It provides two main operations:

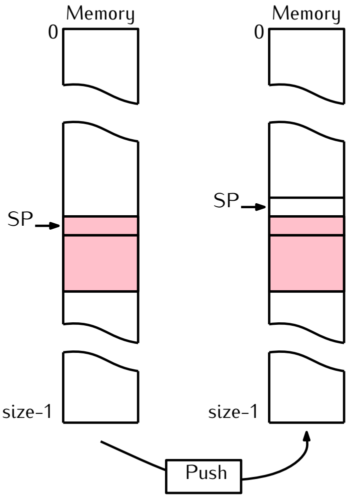
- **push**, which adds an element on TOP of the stack;
- **pop**, which removes an element from the TOP of the stack.

This is a LIFO (for Last In, First Out) data structure.

Stack: usage



The stack in hardware



- A dedicated area in memory
- a dedicated CPU register : **SP** for **Stack Pointer**
- two CPU instructions: `pop` and `push`
- **WARNING: often stack grows towards address `0x00`**

msp430: the stack

3.2.2 Stack Pointer (SP)

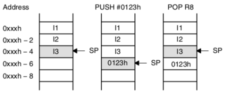
The stack pointer (SP/R1) is used by the CPU to store the return addresses of subroutine calls and interrupts. It uses a predecrement, postincrement scheme. In addition, the SP can be used by software with all instructions and addressing modes. Figure 3–3 shows the SP. The SP is initialized into RAM by the user, and is aligned to even addresses.

Figure 3–4 shows stack usage.

Figure 3–3. Stack Pointer


```
MOV   2(SP),R6  ; Item I2 -> R6
MOV   R7,0(SP)  ; Overwrite TOS with R7
PUSH  #0123h    ; Put 0123h onto TOS
POP   R8        ; R8 = 0123h
```

Figure 3–4. Stack Usage



The special cases of using the SP as an argument to the PUSH and POP instructions are described and shown in Figure 3–5.

Figure 3–5. PUSH SP - POP SP Sequence

The stack pointer is changed after a PUSH SP instruction.

The stack pointer is not changed after a POP SP instruction. The POP SP instruction places SP1 into the stack pointer SP (SP2=SP1)

- `SP` dedicated register
- `SP` always points to the TOP of the stack (ie **last element that was pushed**)
- on push, `SP` moves towards address 0
- on pop, `SP` moves towards address `0xffffffff`
- NB1: 0xxxh means “address 0x0xxx”
- NB2: 0xxxh - 6 means “address 0x0xxx minus 2 ” (two bytes further down in memory)

msp430: the PUSH instruction

PUSH[.W]	Push word onto stack
PUSH.B	Push byte onto stack
Syntax	PUSH src or PUSH.W src PUSH.B src
Operation	SP – 2 → SP src → @SP
Description	The stack pointer is decremented by two, then the source operand is moved to the RAM word addressed by the stack pointer (TOS).
Status Bits	Status bits are not affected.
Mode Bits	OSCOFF, CPUOFF, and GIE are not affected.
Example	The contents of the status register and R8 are saved on the stack.

```
PUSH    SR      ; save status register
PUSH    R8      ; save R8
```

Example The contents of the peripheral TCDAT is saved on the stack.

```
PUSH.B  &TCDAT  ; save data from 8-bit peripheral module,
                ; address TCDAT, onto stack
```

Note: The System Stack Pointer

The system stack pointer (SP) is always decremented by two, independent of the byte suffix.

msp430: the POP instruction

*** POP[.W]**	Pop word from stack to destination
*** POP.B**	Pop byte from stack to destination
Syntax	POP dst POP.B dst
Operation	@SP –> temp SP + 2 –> SP temp –> dst
Emulation	MOV @SP+,dst or MOV.W @SP+,dst
Emulation	MOV.B @SP+,dst
Description	The stack location pointed to by the stack pointer (TOS) is moved to the destination. The stack pointer is incremented by two afterwards.
Status Bits	Status bits are not affected.
Example	The contents of R7 and the status register are restored from the stack.

```
POP     R7      ; Restore R7
POP     SR      ; Restore status register
```

Example The contents of RAM byte LEO is restored from the stack.

```
POP.B   LEO     ; The low byte of the stack is moved to LEO.
```

Example The contents of R7 is restored from the stack.

```
POP.B   R7      ; The low byte of the stack is moved to R7,
                ; the high byte of R7 is 00h
```

Example The contents of the memory pointed to by R7 and the status register are restored from the stack.

```
POP.B   0(R7)   ; The low byte of the stack is moved to the
                ; the byte which is pointed to by R7
                : Example:  R7 = 203h
                ;           Mem(R7) = low byte of system stack
                : Example:  R7 = 20Ah
                ;           Mem(R7) = low byte of system stack
POP     SR      ; Last word on stack moved to the SR
```

Note: The System Stack Pointer

The system stack pointer (SP) is always incremented by two, independent of the byte suffix.

msp430: the CALL instruction

CALL	Subroutine		
Syntax	CALL	dst	
Operation	dst	–> tmp	dst is evaluated and stored
	SP – 2	–> SP	
	PC	–> @SP	PC updated to TOS
	tmp	–> PC	dst saved to PC

Description A subroutine call is made to an address anywhere in the 64K address space. All addressing modes can be used. The return address (the address of the following instruction) is stored on the stack. The call instruction is a word instruction.

Status Bits Status bits are not affected.

Example Examples for all addressing modes are given.

```
CALL   #EXEC   ; Call on label EXEC or immediate address (e.g. #0A4h)
               ; SP–2 → SP, PC+2 → @SP, @PC+ → PC

CALL   EXEC    ; Call on the address contained in EXEC
               ; SP–2 → SP, PC+2 → @SP, X(PC) → PC
               ; Indirect address

CALL   &EXEC   ; Call on the address contained in absolute address
               ; EXEC
               ; SP–2 → SP, PC+2 → @SP, X(0) → PC
               ; Indirect address

CALL   R5      ; Call on the address contained in R5
               ; SP–2 → SP, PC+2 → @SP, R5 → PC
               ; Indirect R5

CALL   @R5     ; Call on the address contained in the word
```

msp430: the RET instruction

*** RET**	Return from subroutine
Syntax	RET
Operation	@SP → PC SP + 2 → SP
Emulation	MOV @SP+,PC
Description	The return address pushed onto the stack by a CALL instruction is moved to the program counter. The program continues at the code address following the subroutine call.
Status Bits	Status bits are not affected.

DEMO: let's follow the call to PP

Problems we need to solve

- ☑ Problem #1: **Jump** from `main` to `PP` ... and then **come back**
- ☑ Problem #2: How do we make it work with **call cascades** (eg `main` calls `P`, `P` calls `Q`, etc)?
- ☑ Problem #3: How do we make it work with **recursive functions** (`P` calls itself)?
- ☐ Problem #4: Deal with **local variables** (the call to `PP` should not break anything in `main`)?
- ☐ Problem #5: How do we **pass parameters** from `main` to `PP`?
- ☐ Problem #6: How do we get **the return value** from `PP` to `main`?

Problem #4, #5 and #6: Local variables, Parameter and function results

Now that we have a stack:

- We can use it to store all these variables and values
- But not always: we may use registers for that too
- This is defined in the **ABI** (Application Binary Interface)
- The ABI is defined :
  - partly by CPU designers,
  - partly by the compiler
  - (and sometimes also by the OS)

Stack Frame & the Frame Pointer

- A **Stack Frame** is a piece of the frame that is used to store and acces all information relating to the local environment of **one** function:
  - local variables,
  - parameters
  - return values
- The **Frame Pointer** can be used to designate a limit to this frame.
- In some architectures, there even is a **dedicated register**, called `FP`

msp430: the “frame pointer”

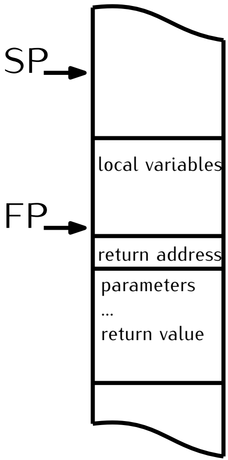
- The processor documentation doesn’t explicitely define one register for that
- This convention is left to the compiler to define
- Example, with gcc:

mspgcc’s ABI[a]
Register usage

• *If you intend to interface assembly routines with your C code, you need to know how GCC uses the registers. This section describes how registers are allocated and used by the compiler. (You can override GCC’s settings by issuing -ffixed-regs=...)*
• ***r0, r2, and r3*** *- are fixed registers and* ***not used by the compiler in any way****. They* ***cannot be used for temporary register arguments*** *either.*
• ***r1 - is the stack pointer****. The compiler modifies it only in the function prologues and epilogues, and when a function call with a long argument list occurs.* ***Do not modify it yourself under any circumstances!!!***
• ***r4 - is the frame pointer****. This can be used by the compiler, when va_args is used. When va_args is not used, and optimization is switched on, this register is eliminated by the stack pointer.*

[a] `http://mspgcc.sourceforge.net/manual/c1225.html`

The frame pointer



- Exact shape and order depends on convention
- See demo for gcc and msp430

msp430 - calling convention

Function calling conventions Fixed argument lists
Function arguments are allocated left to right. They are assigned from r15 to r12. If more parameters are passed than will fit in the registers, the rest are passed on the stack. This should be avoided since the code takes a performance hit when using variables residing on the stack.
[...]
Return values
The various functions types return the results as follows:

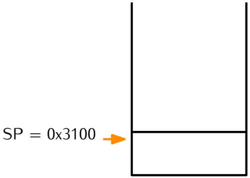
- char, int and pointer functions return their values r15
- long and float functions return their values in r15:r14
- long long functions return their values r15:r14:r13:r12

If the returned value wider than 64 bits, it is returned in memory.

Problems we need to solve

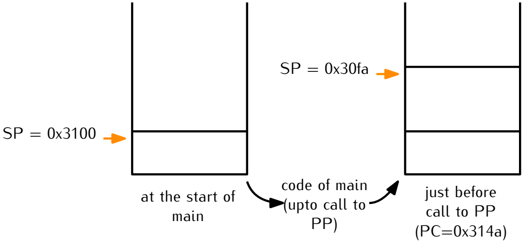
- ☑ Problem #1: **Jump** from `main` to `PP` ... and then **come back**
- ☑ Problem #2: How do we make it work with **call cascades** (eg `main` calls `P`, `P` calls `Q`, etc)?
- ☑ Problem #3: How do we make it work with **recursive functions** (`P` calls itself)?
- ☑ Problem #4: Deal with **local variables** (the call to `PP` should not break anything in `main`)?
- ☑ Problem #5: How do we **pass parameters** from `main` to `PP`?
- ☑ Problem #6: How do we get **the return value** from `PP` to `main`?

Demo - the stack

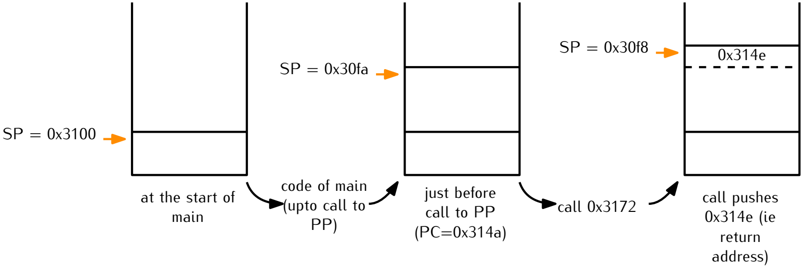


at the start of
main

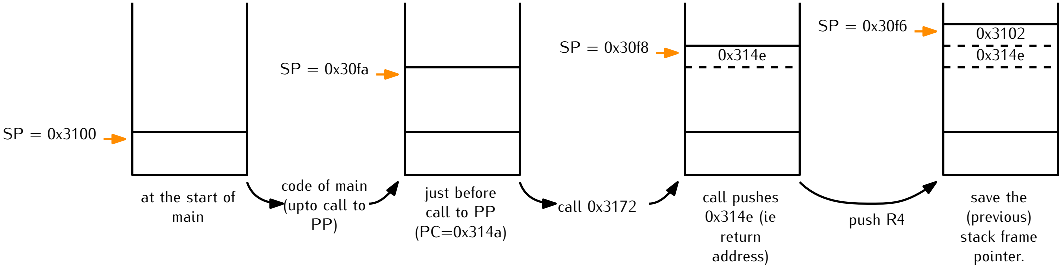
Demo - the stack



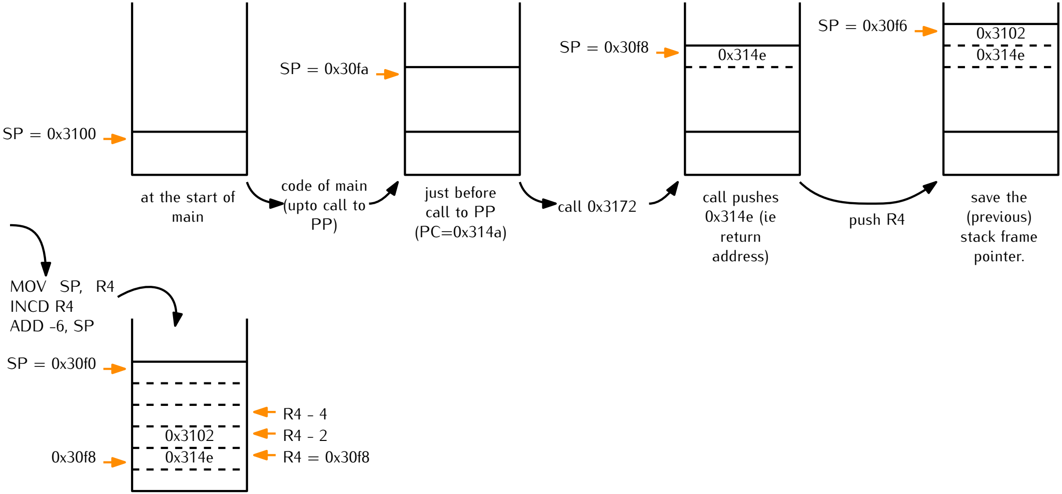
Demo - the stack



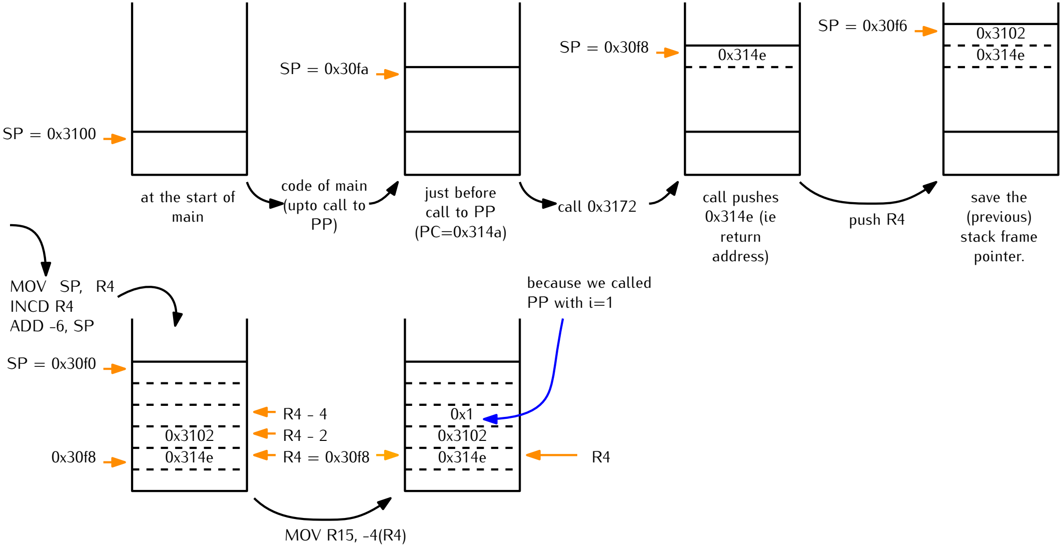
Demo - the stack



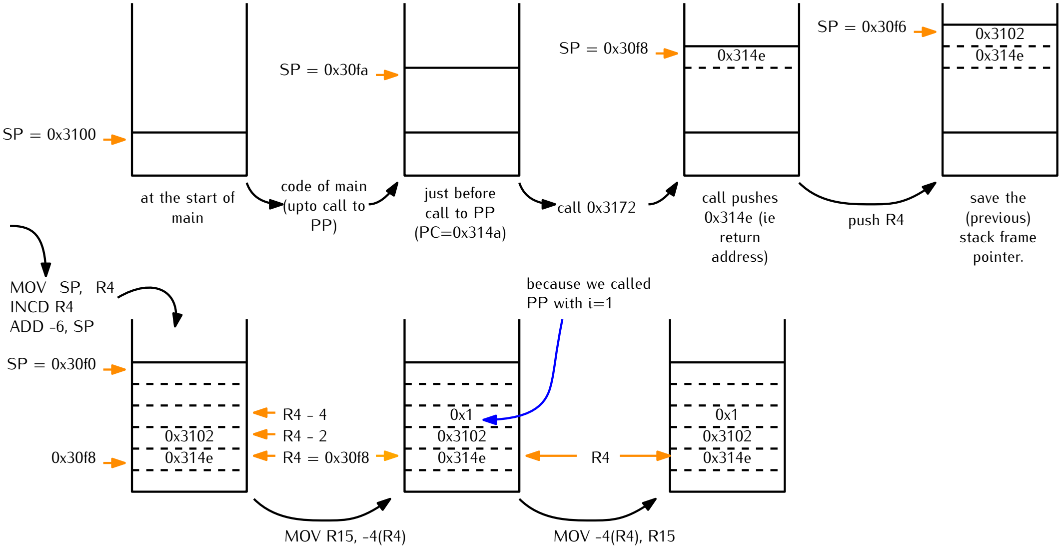
Demo - the stack



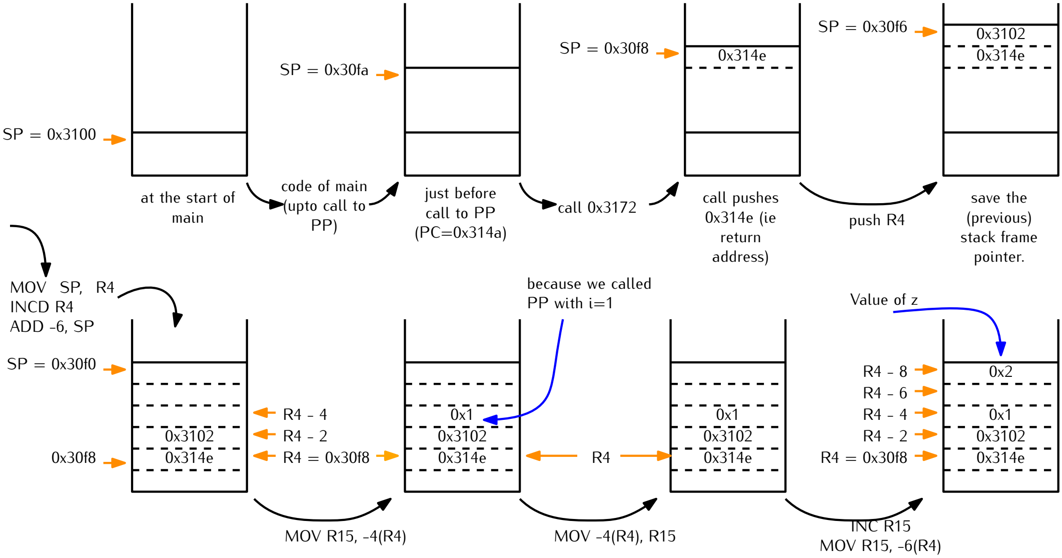
Demo - the stack



Demo - the stack

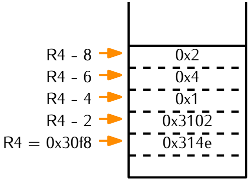


Demo - the stack



Demo - the stack (cont'd)

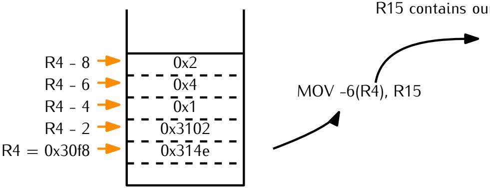
```
MOV -8(R4), R15 // copy z to R15
INCD R15 // R15 <– R15 + 2
MOV R15, -6(R4) // copy R15 to stack
```



Demo - the stack (cont'd)

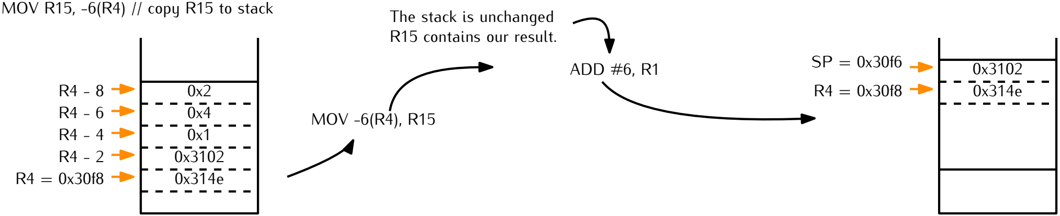
MOV -8(R4), R15 // copy z to R15
INCD R15 // R15 <– R15 + 2
MOV R15, -6(R4) // copy R15 to stack

The stack is unchanged
R15 contains our result.

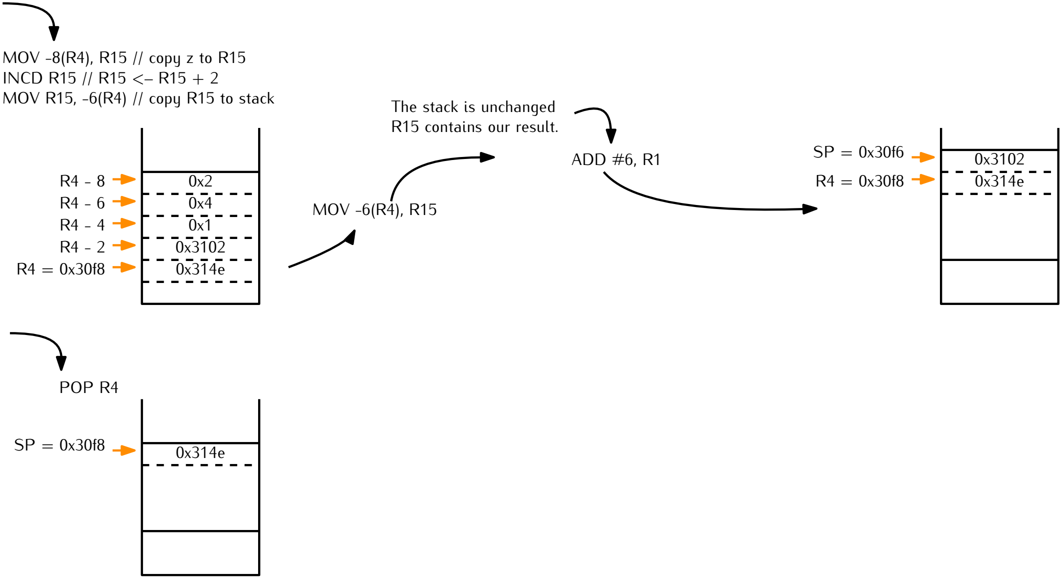


Demo - the stack (cont'd)

MOV -8(R4), R15 // copy z to R15
INCD R15 // R15 <– R15 + 2
MOV R15, -6(R4) // copy R15 to stack

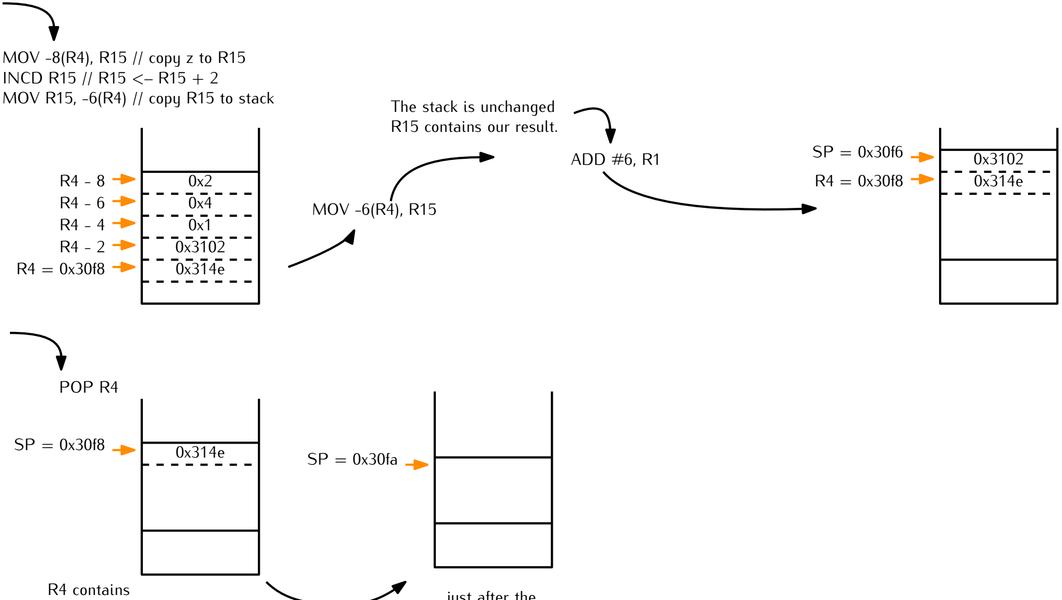


Demo - the stack (cont'd)



R4 contains

Demo - the stack (cont'd)



Demo (factorial) - the code

```
int fact(int x){
  if (x==0) {return 1;}
  else {return x * fact(x-1);}
}


int main(){
  int n,y;
  int z;
  n = 5;
  y = fact(n);
  z = y;
  return 0;
}
```

Demo (factorial) - the code

```
0000312c <main>:
  312c: 04 41        mov r1,r4
  312e: 24 53        incd r4
  3130: 31 50 fa ff  add #-6,r1 ;#0xfffa
  3134: b4 40 05 00  mov #5,-8(r4) ;#0x0005, 0xfff8(r4)
  3138: f8 ff
  313a: 1f 44 f8 ff  mov -8(r4),r15 ;0xfff8(r4)
  313e: b0 12 5c 31  call #0x315c
  3142: 84 4f fa ff  mov r15,-6(r4) ;0xfffa(r4)
  3146: 94 44 fa ff  mov -6(r4),-4(r4) ;0xfffa(r4), 0xfffc(r4)
  314a: fc ff
  314c: 0f 43        clr r15
  314e: 31 50 06 00  add #6,r1 ;#0x0006
```

Demo (factorial) - the code

```
0000315c <fact>:
    315c: 04 12         push r4
    315e: 04 41         mov r1,r4
    3160: 24 53         incd r4
    3162: 21 83         decd r1
    3164: 84 4f fc ff   mov r15,-4(r4) ;0xfffc(r4)
    3168: 84 93 fc ff   tst -4(r4) ;0xfffc(r4)
    316c: 02 20         jnz $+6        ;abs 0x3172
    316e: 1f 43         mov #1,r15 ;r3 As==01
    3170: 10 3c         jmp $+34       ;abs 0x3192
    3172: 1f 44 fc ff   mov -4(r4),r15 ;0xfffc(r4)
    3176: 3f 53         add #-1,r15 ;r3 As==11
    3178: b0 12 5c 31   call #0x315c
    317c: 02 12         push r2
    317e: 32 c2         dint
```

Conclusions

- We have seen **how a simple CPU works**, how it interprets instructions, how it deals with peripherals, and how we can use it to program with functions.
- We have discussed **Instruction Set Architectures**, ie the interface provided by the CPU to programmers, compilers and operating systems.
- Many topics cannot be covered which have been explored by researchers and industry to try and make these machines more and more efficient:
  - Memory hierarchy
  - Parallelism
  - Energy consumption