Tutorial4.1

Stack , Load&Store instructions

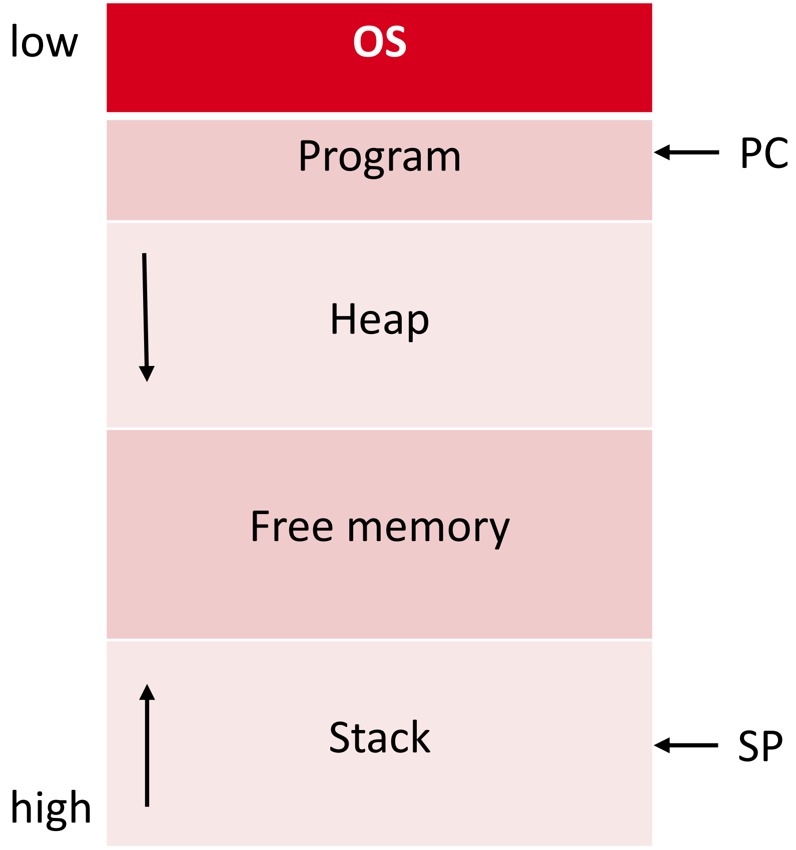
Lei Wang

lei.wang2@ucalgary.ca

UNIVERSITY OF CALGARY

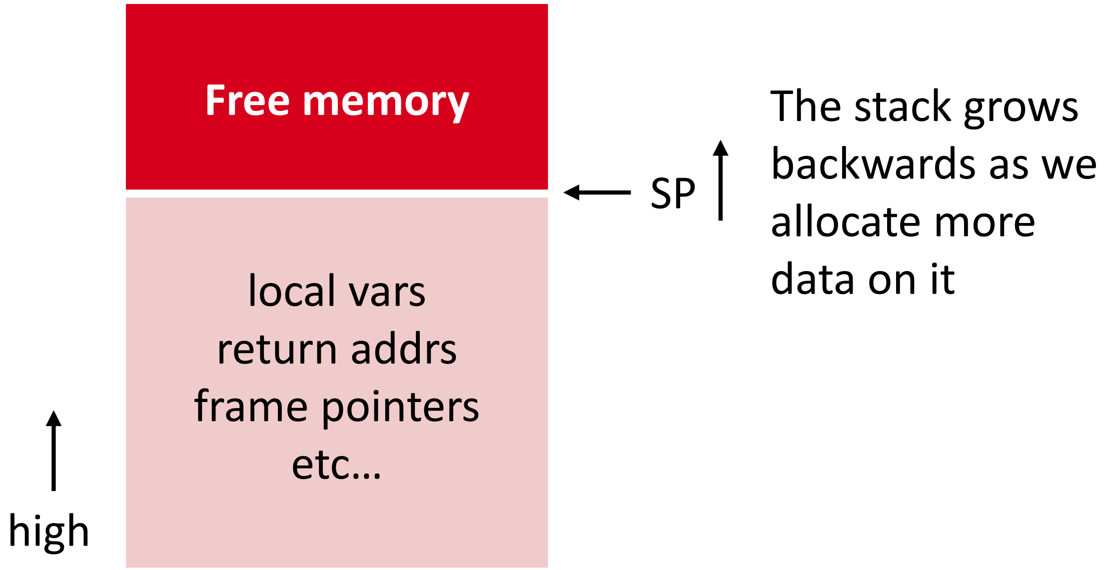
Stack Memory

- space in RAM to store data for functions
  - A stack frame is *pushed onto* the stack when the function is called
  - the frame is *popped* when the function returns
- Stack Uses high memory
  - It grows “backwards”(toward 0)
- *Programs* are loaded into low memory
- *Heap* is used for dynamically allocated memory in a program
  - Done in C using *malloc()* and *free()*
- *Stack* is used for local variables, static memory allocation, return address



Stack

- used to store local variables and arrays whose sizes are known at run time
- The register $sp keeps track of what address is currently at the top of the stack



Stack Variables

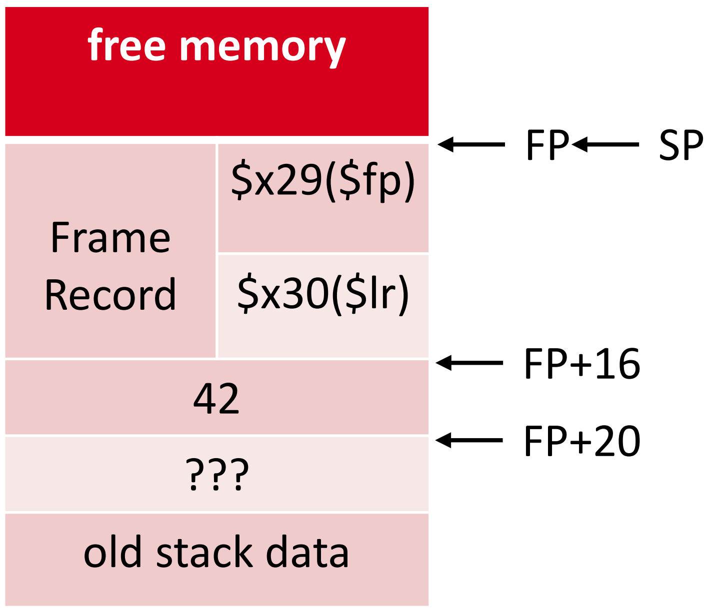
- Created by allocating extra space in the stack frame when entering a function
  - Is in addition to the frame record's 16 bytes
- Stack variables are addressed by adding an offset to the base address in FP
- Addressed are specified inside the [ ] of load and stored
  - [base address, offset]
  - Eg: Write 42 to the first stack variable

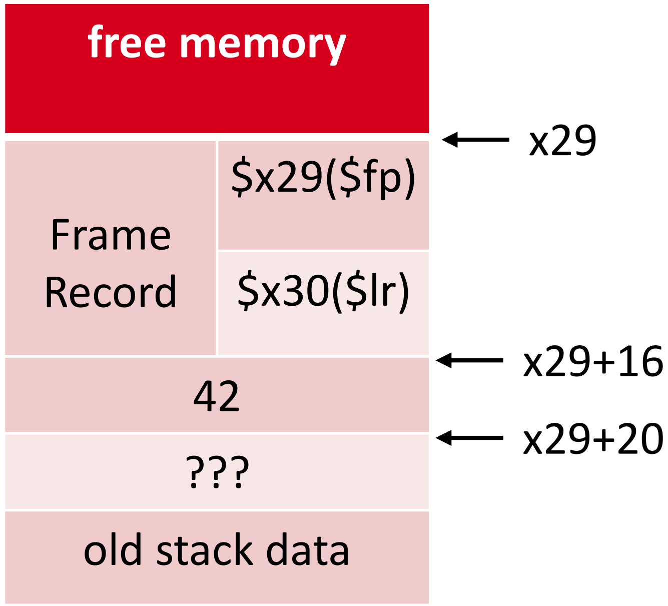
```
mov     w20, 42
str     w20, [x29, 16]
```

  - Eg:Read from the second stack variable

```
ldr     w21, [x29, 20]
```

Stack variable





Basic CPU Arthitectures

- Only *load* and *store* instructions can access RAM
- Other instructions operate on specified registers, not on RAM
  - Registers are more quickly accessed than RAM, so this is fast
- Typical program sequence:
  - load registers from memory
  - Execute an instruction using two source registers, putting the result into a destination register
  - store the result back into memory
- access memory by address
- register is limited, mostly variables are stored in memory.

UNIVERSITY OF CALGARY

Load

- Load Register
  - 64 bit form: *ldr Xt, addr*
    - Loads register with 8 bytes read from RAM
    - Eg: ldr x22, [x29, 56]
  - 32 bit form: *ldr Wt, addr*
    - loads register with 4 bytes from RAM
- Load Byte
  - Form: *ldrb Wt, addr*
  - Loads 1 bytes from RAM into low-order part of *Wt*, Zero-extending high-order bits
- Load Signed Byte
  - Form(32-bit): *ldrsb Wt, addr*
  - Form(64-bit): *ldrsb Xt, addr*
  - Loads 1 byte from RAM into low-order, sign-extending high-order bits

Load

- Load Halfword
  - Form(32-bit only): *ldrh wt, addr*
- Load Signed Halfword
  - Form(32-bit): *ldrsh Wt, addr*
  - Form(64-bits): *ldrsh Xt, addr*
- Load Signed Word
  - Form(64-bit only): *ldrsw Xt, addr*

Store

- Store Register
  - 64-bit form: *str Xt, addr*
  - Stores doubleword(8 bytes) in *Xt* to RAM
  - Eg: *str x20, [x29, 56]*
  - 32-bit form: *str Wt, addr*
- Store Byte
  - Form(32-bit only): *strb Wt, addr*
  - stores low-order byte in *Wt* to RAM
- Store Halfword
  - Form(32-bit only): *strh Wt, addr*
  - stores low-order halfword(2 bytes) in *Wt* to RAM

Load/Store Addressing modes

- Base plus immediate offset
  - Form: [base, #imm]
- Base plus 64-bit register offset
  - Form: [base, Xm]
- Base plus 32-bit register offset
  - sign extended form: [base, Wm, SXTW]
  - Zero extended form: [base, Wm, UXTW]
- Pre-indexed by immediate offset
  - Form: [base, #imm]!→base is first updated and then used for the load/store
- Post-indexed by immediate offset
  - Form: [base], #imm →base is used for load/store and then updated

Local Variables

- In C, can be declared in a block of code
  - i.e. in any construct delimited by{…}
  - Eg:

```
int main()
 {
    int a = 5, b = 7;   <-- local to main()


    if (a < b) {
       int c;           <-- local to if-construct
       c = 10;
       ...
    }
    ...
 }
```

- Local variables are implemented as stack variables in assembly
  - allocated when entering the block
    - Done by directly decrementing SP
    - Are read/written using FP plus a offset
  - Deallocated when leaving the block
    - Done by incrementing SP

equivalent assembly code:

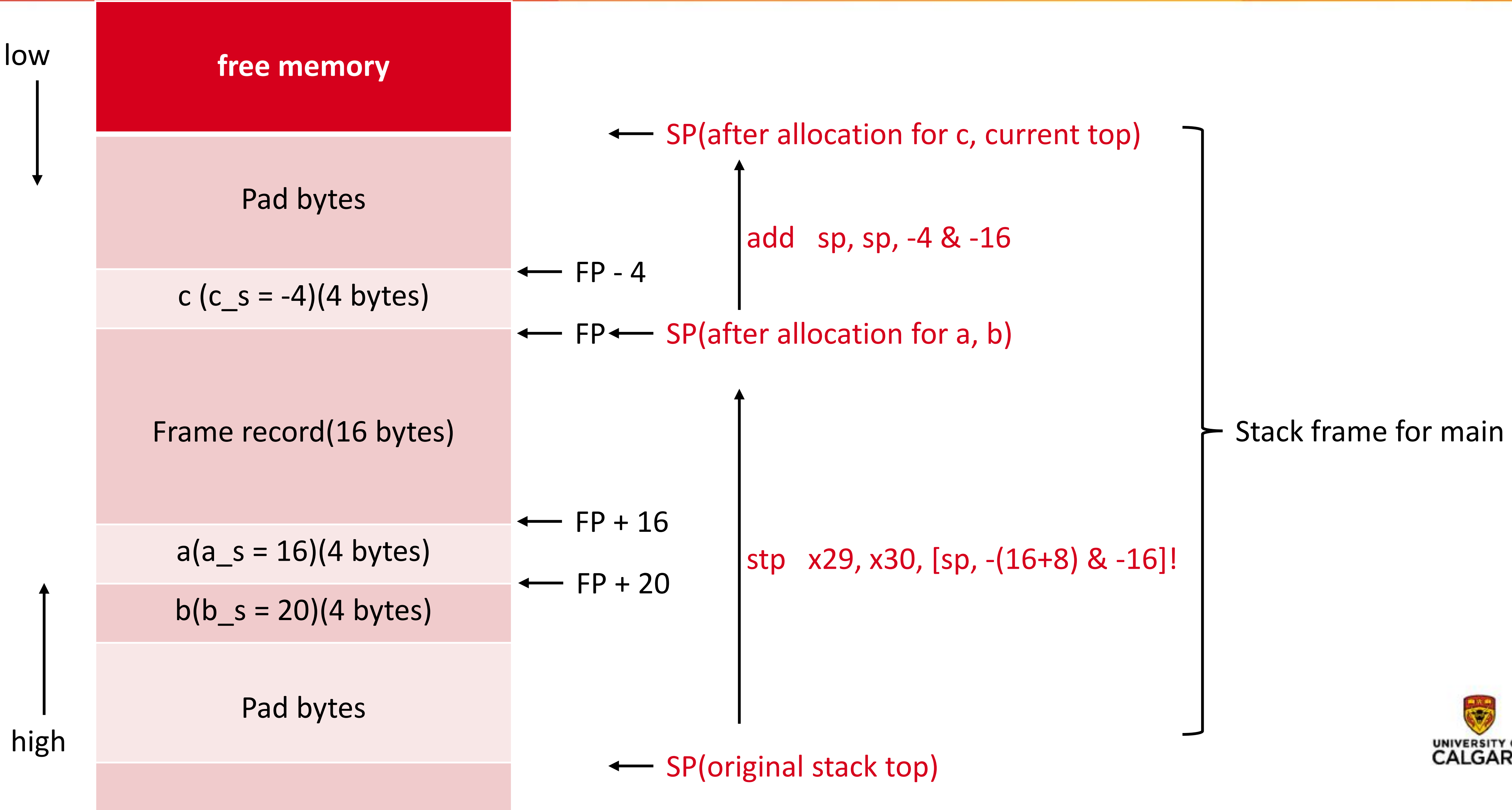
```
      a_s = 16
      b_s = 20
      c_s = -4
      ...
main: stp   x29, x30, [sp, -(16+8) & -16]!// alloc RAM for a,b
      mov   fp, sp
      mov   w19, 5           // init a to 5
      str   w19, [x29, a_s]
      mov   w20, 7           // init b to 7
      str   w20, [x29, b_s]

      cmp   w19, w20

      b.ge  next

      // start of block of code for if-else
      add   sp, sp, -4 & -16 // alloc RAM for c

      mov   w21, 10          // init c to 10
      str   w21, [x29, c_s]
      ...
      // end of block of code
      add   sp, sp, 16       // dealloc RAM for c
next: ...


       ldp   x29, x30, [sp], -((16+8)&-16)// dealloc RAM for a,b
       ret
```


UNIVERSITY OF CALGARY

how to allocate ?

- stp   x29, x30, [sp, alloc]!
  - *alloc* = -(16 + [memory needed for local/stack variables])&-16
  - memory is allocated by decrementing the sp, form high to low
  - 16 bytes for the Frame Record
  - &-16 clears the last 4 bits of alloc ensuring it is divisible by 16
- example
  - 160 bytes needed for variables: -(16 + 160)&-16 = -176, so we need 176 bytes
  - 170 bytes needed for variables: -(16 + 170)&-16 = -192, so we need 192, the last unused 6 bytes are simply padded
- SP can be moved many times when saving variables
- ldp x29, x30, [sp], dealloc   //dealloc =-alloc

what happened after allocation?



Coding practice

- write assembly equivalent of this program

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i, sum = 0;

    for(i = 1; i < 100; i++) {
        sum += i;
    }
    printf("1 + 2 + ... + 98 + 99 = %d", sum);
    return 0;
}
```

```
// define register aliases
fp              .req x29
lr              .req x30
//define format strings
print_string:   .string "1 + 2 + ...+ 98 + 99 = %d\n"
                .balign 4
                .global main
main:
                //save FP and LR to the stack, alloc 8 bytes for sum and i
                stp fp, lr, [sp, -(16+8)&-16]!
                mov fp, sp// set fp to the stack addr(sp)

                mov w19, 0
                mov w20, 1

                str w19, [fp, 16]
                str w20, [fp, 20]

//optimization
                b loop_test

loop:           ldr w20, [fp, 20]
                ldr w19, [fp, 16]
                add w19, w19, w20
                add w20, w20, 1
                str w20, [fp, 20]
                str w19, [fp, 16]

loop_test:
                cmp w20, 100
                b.lt loop

//print result
loop_exit:      ldr x0, =print_string
                ldr w1, [fp, 16]
                bl printf
                mov w0, 0
                ldp fp, lr, [sp], -(-(16+8)&-16)
                ret
```

reference

- http://edwinckc.com/cpsc355/70-stack
- slides from Prof. Manzara's lecture.