# 50.002 Computation Structures
## C language, Procedures & Stacks

**Oliver Weeger**

**2018 Term 3, Week 5, Session 2**

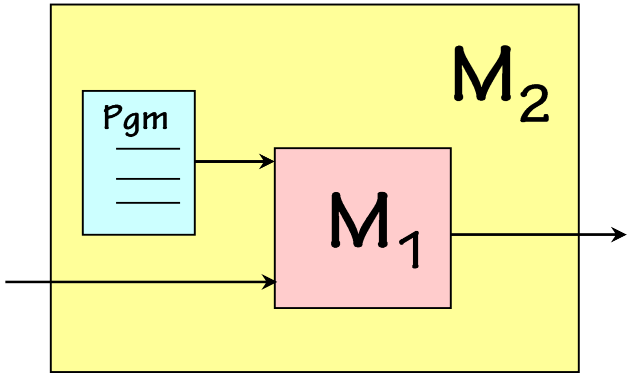
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

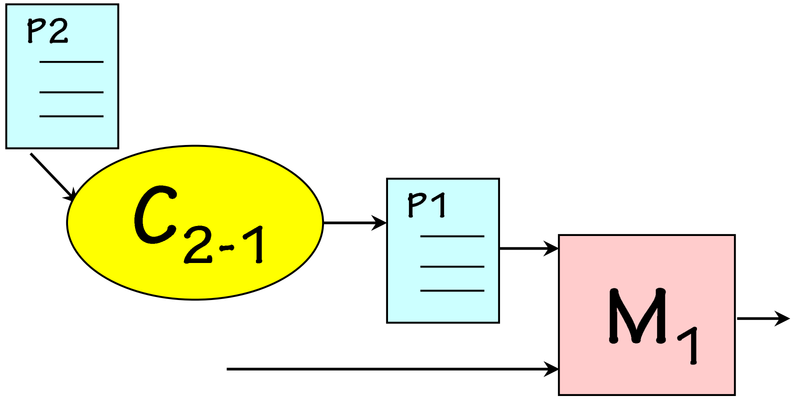
# Software Abstraction Strategy

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

**Interpretation:**




**Compilaton:**




- Initial steps: **compilation tools** `0x 80 61 10 00`
  - ✓ Assembler (**UASM**): symbolic representation of machine language `ADD(R1,R2,R3)`
  - ➢ High-level languages & Compiler (**C**): symbolic representation of algorithm `c=a+b;`
- Subsequent steps: **interpretive tools**
  - Operating system
  - Apps (e.g., Browser)

# C language overview: an example



```
int f(int a, int b) {
    int c=1;
    return a+b+c;
}
int g(int a, int b) {
    return a-b;
}
int main(int argc, char* argv[]) {
    int a,b;
    printf("a=");
    scanf("%d", &a);
    printf("b=");
    scanf("%d", &b);
    if (a>b)
        printf("%d\n", g(a,b));    // returns a-b
    else
        printf("%d\n", f(a,b));    /* returns a+b+1 */
    return 0;
}
```

procedure identifier

return type

procedures (functions)

programme entry point

output to console/screen

input from console/keyboard

conditional (control statement)

(local) variable identifier

literal

expression

operator

parameter

data type

statement

reference to address

comments

# C language overview

- Data model (i.e., data in von Neumann model): **values** (byte/word data) + **adresses**
- Control structure (i.e., code in vN model): **sequence**, **branch**, **iteration**
- Syntax:
  - C **programs** are made of **statements** (which may be marked by keywords)
  - **Values** are denoted by **expressions** (**literals**, **identifiers**, **operators**)
  - **Procedural** language: organize tasks into re-useable procedures
  - **No rich data model** like in object-oriented languages (only structs, no classes)

C online compiler: https://www.onlinegdb.com/online_c_compiler

# Data types

- Strictly aligned with byte code
- Signed (2's complement) and unsigned integers: `int a; unsigned int b;`
- Differently sized **integer types**, with typically at least x bits:
  8-bit `char`, 16-bit `short`/`int`, 32-bit `long`/`int`, 64-bit `long long`
  (for the **β** we assume `int` to be 32-bit)
- 8-bit ASCII **characters**: `char c = 'c'; // =0x63=99`
- Real/Floating point number data types: `float`, `double`, `long double` (not used here)
- **Arrays**: Packed lists of variables of the same data type: `int A[10]; A[0]=3;`
- **Strings**: Arrays of characters: `char str[12] = "Hello World!";`
- **Structures**: Packed forms of (heterogeneous) data:

```
struct rational {
    int numerator;
    int denominator;
}
```

```
struct rational r;
r.numerator   = 1;
r.denominator = 3;
```

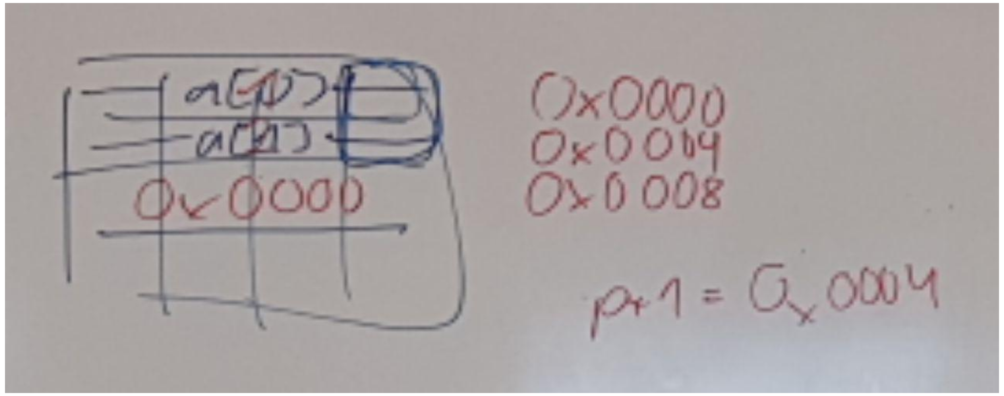
# Pointers



- **Reference operator** `&`: Gives the (32-bit) address of a variable: `&var`
- **Pointers**: Variables that store the address of a variable: `int var=17; int* p=&var;`
- **Dereference operator** `*`: Gives the value at the address of a pointer: `*p=17;`
- Array name is just a (fixed) pointer to the address of the first element!

```
int  var = 17;
int* p   = &var;
printf(" var =%d\n", var);    // 17
printf("&var =%u\n",&var);    // 0x1000
printf(" pvar=%d\n", p);      // 0x1000
printf("*pvar=%d\n",*p);      // 17
printf("&pvar=%u\n",&p);      // 0x1004
*p = 18;
printf(" var =%d\n", var);    // 18
```

```
int  a[2];
int* p = a;
a[0]   = 1;
*(p+1) = 2;
printf("a[0]=%d\n", *p);      // 1
printf("a[1]=%d\n", a[1]);    // 2
```




https://www.programiz.com/c-programming/c-pointers

# Operators and control structures

- **Operators** for integer expressions: `=,+,-,*,/,%,<<,>>,&&,||,++,--,+=,-=,…`
- Expressions can also include parenthesis: `d=(a+b)*c;`
- **Compounds/Blocks** of statements are include in curly brackets: `{…}`
- **Control structures** for conditional jumps of von Neumann machine
- **Branching** (`if-else`-statement):

```
if (<integer expression>)
   <true-statement>
else
   <false-statement>
```

also: `switch`-statement for multiple branches

- **Iterations/loops** (`while`-statement):

```
while (<integer expression>)
   <loop-statement>
```

also: `for`-statement and `do-while`-statement

# Compilation of C code into byte code

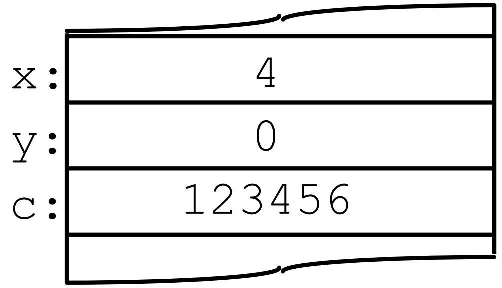
- **Variables** are assigned **memory locations** and accessed via `LD` or `ST`
- **Operators** translate to **ALU instructions**
- **Small constants** translate to “literal-mode” ALU instructions
- **Large constant** translate to initialized variables

<u>C code:</u>

```
int x = 4, y;
y = (x-3)*(y+123456)
```

<u>Beta assembly code:</u>

```
x:      LONG(4)
y:      LONG(0)
c:      LONG(123456)
        ...
        LD(x, r1)
        SUBC(r1,3,r1)
        LD(y, r2)
        LD(C, r3)
        ADD(r2,r3,r2)
        MUL(r2,r1,r1)
        ST(r1,y)
```



# Compilation of array and structs

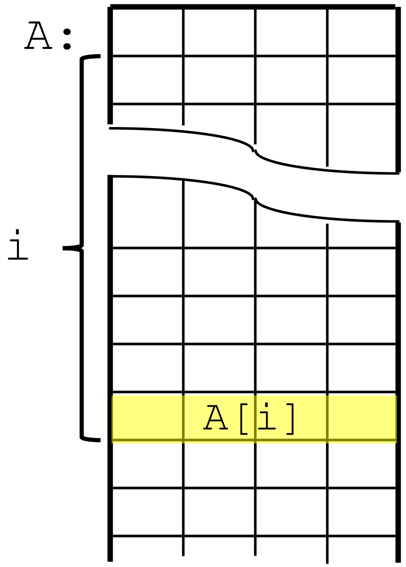
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

<u>C code:</u>

```
int A[100];
...
A[i] += 1;
```

<u>Beta assembly code:</u>

```
A:    .=.+4*100     | Leave room for 100 ints
...
LD(i,r1)
MULC(r1,4,r2)       | index -> byte offset
LD(r2,A,r0)         | A[i] -> R0
ADDC(r0,1,r0)       | increment
ST(r0,A,r2)         | A[i] <- R0
```

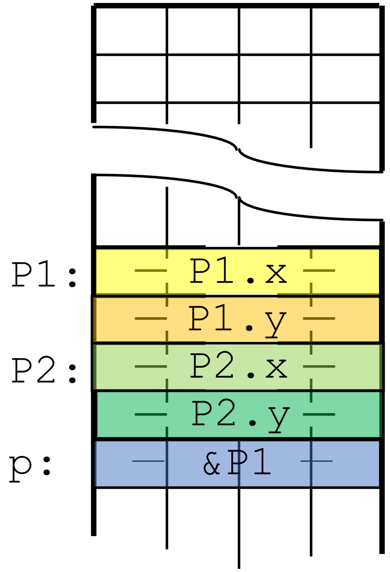



<u>C code:</u>

```
struct Point
{ int x, y; }
P1, P2, *p;

...
P1.x = 157;

...
p = &P1;

p->y = 157;
```

<u>Beta assembly code:</u>

```
P1: .=.+8
P2: .=.+8
p:  .=.+4
x=0                   | Offset for x component
y=4                   | Offset for y component
...
ADDC(r31,157,r0)      | r0 <- 157
ST(r0,P1+x)           | P1.x = 157
...
ADDC(r31,P1,r3)       | Put &P1 in r3
ST(r3,p)              | Store r3 in p
ST(r0,y,r3)           | p->y = 157;
```



# Conditionals

C code:

Beta assembly code:

```
if (expr)                <compile expr into rx>
{                        BF(rx, Lendif)
    STUFF                <compile STUFF>
}                      Lendif:
```

```
if (expr)                <compile expr into rx>
{                        BF(rx, Lelse)
    STUFF1               <compile STUFF1>
}                        BR(Lendif)
else                   Lelse:
{                        <compile STUFF2>
    STUFF2             Lendif:
}
```

Tricks & optimizations:

```
if (y > 32)              LD(y,R1)
{                        CMPLEC(R1,32,R1)
    x = x+1;             BT(R1,Lendif)
}                        ADDC(R2,1,R2)
                       Lendif:
```

# Iterations/Loops

C code:

```
while (expr)
{
    <STUFF>
}
```

Beta assembly code:

```
Lwhile:
    <compile expr into rx>
    BF(rx,Lendwhile)
    <compile STUFF>
    BR(Lwhile)
Lendwhile:
```

Slightly optimized beta assembly code:

```
    BR(Ltest)
Lwhile:
    <compile STUFF>
Ltest:
    <compile expr into rx>
    BT(rx,Lwhile)
Lendwhile:
```

Compilers spend a lot of time optimizing in and around loops:

- moving all possible computations outside of loops
- *"unrolling"* loops to reduce branching overhead
- simplifying expressions that depend on "loop variables"

# Example: Factorial

C code:

```
int n = 3, r;

r = 1;

while (n > 0)
{
    r = r * n;

    n = n - 1;
}
```

Beta assembly code:

```
n:      INT32(3)
r:      INT32(0)
start:  ADDC(r31,1,r0)
        ST(r0,r)
loop:   LD(n,r1)
        CMPLT(r31,r1,r2)
        BF(r2,done)
        LD(r,r3)
        LD(n,r1)
        MUL(r1,r3,r3)
        ST(r3,r)
        LD(n,r1)
        SUBC(r1,1,r1)
        ST(r1,n)
        BR(loop)
done:
```

11 instructions in loop

Optimized beta assembly:

```
n:      INT32(3)
r:      INT32(0)
start:  ADDC(r31,1,r0)
        ST(r0,r)
        LD(n,r1)
        LD(r,r3)
loop:   CMPLT(r31, r1, r2)
        BF(r2, done)
        MUL(r1, r3, r3)
        SUBC(r1, 1, r1)
        BR(loop)
done:   ST(r3,r)
        ST(r1,n)
```

- Moved `LD` & `ST` outside
- Use only registers in loop

→ 5 instructions in loop

Fully optimized assembly:

```
n:      INT32(3)
r:      INT32(0)
start:  LD(n,r1)
        ADDC(r31,1,r3)
        BF(r1, done)
loop:   MUL(r1, r3, r3)
        SUBC(r1, 1, r1)
        BT(r1,loop)
done:   ST(r3,r)
        ST(r1,n)
```

- Avoid initializing `r`
- Avoid overhead of conditional

→ 3 instructions in loop

# Procedures & functions

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

- Reusable code fragments that are called as needed
- Single named entry point (identifier)
- Parameterizable (arguments passed by value, not by reference)
- Local state (variables)
- Upon completion control is transferred back to caller
- Procedures can be recursive!

```
<return type>  <identifier>(<parameters>)
   <statement>
```

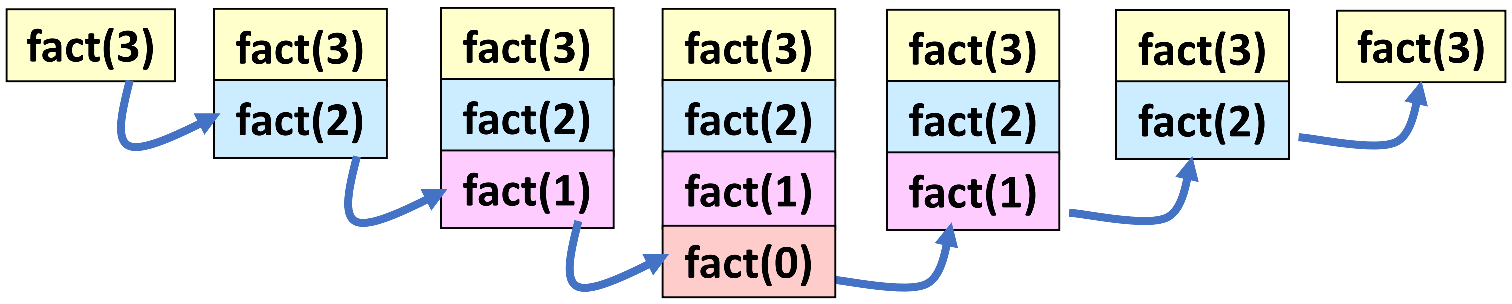
```
int fact(int n)
{
   int r = 1;
   while (n>0) {
      r = r*n;
      n--;
   }
   return r;
}
```

```
int fact(int n)
{
   if (n>0)
      return n*fact(n-1);
   else
      return 1;
}
```

# Compiling procedures

- Call sequence: jump/branch into procedure, pass arguments, allocate local variables, return value, jump back to caller statement

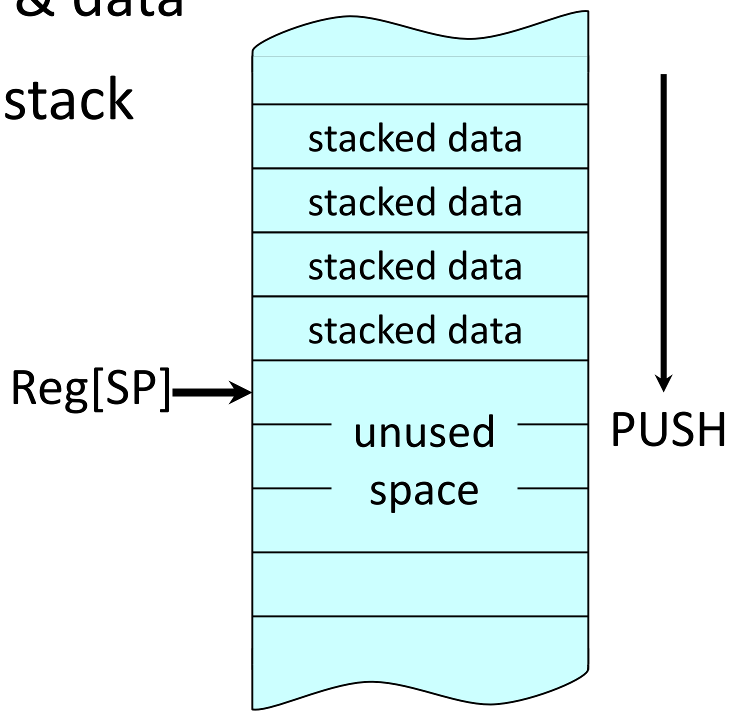
```
int fact(int n) {
   if (n > 0)
      return n*fact(n-1);
   else
      return 1;
}
```




- Need memory locations for dynamically allocated local variables: **heap**
- Need memory locations for overhead (**activation record**) of procedure: **stack**
  (arguments, values in registers, local variables, return address, return value)

# Stack Implementation

- Stack: sequential, last-in-first-out (LIFO) data structure
- Block of memory is reserved "well away" from the program code & data
- **Stack pointer (SP)**: Register R29 contains first unused location of stack
- Stack management macros:
  - `PUSH(Rc)`: Push value in Reg[Rc] onto stack
    - Reg[SP] = Reg[SP] + 4 `ADDC(R29,4,R29)`
    - Mem[Reg[SP]-4] = Reg[Rc] `ST(Rc,-4,R29)`
  - `POP(Rc)`: Pop value on top of stack into Reg[Rc]
    - Reg[Rc] = Mem[Reg[SP]-4] `LD(R29,-4,Rc)`
    - Reg[SP] = Reg[SP] - 4 `SUBC(R29,4,R29)`
  - `ALLOCATE(k)`: Reserve k words of stack
    - Reg[SP] = Reg[SP] + 4*k `ADDC(R29,4*k,R29)`
  - `DEALLOCATE(k)`: Release k words of stack
    - Reg[SP] = Reg[SP] - 4*k `SUBC(R29,4*k,R29)`



# Compiling Procedues continued



- Pointers for procedure call overhead:
  - `R29=SP` **Stack pointer**
  - `R28=LP` **Linkage pointer**: return address to caller
  - `R27=BP` **Base pointer**: points into stack to local variables of callee
- Procedure:
  1. Caller pushes arguments onto stack, in reverse order
  2. Caller branches to callee, putting return address (current PC) into LP
  3. Callee pushes LP & BP, sets BP=SP, allocates local vars, pushes any used registers
  4. Callee performes computation, leaving return value in R0
  5. Callee pops registers, (deallocates local vars,) sets SP=BP, pops BP & LP
  6. Callee branches to return address (LP)
  7. Caller pops/deallocates arguments from stack

```
    PUSH(arg_n)
    ...
    PUSH(arg_1)
    BR(f,LP)
    DEALLOCATE(n)
    ...

f:  PUSH(LP)
    PUSH(BP)
    MOVE(SP,BP)
    ALLOCATE(nlocals)
    (push other regs)

    ...

    (pop other regs)
    MOVE(val,R0)
    MOVE(BP,SP)
    POP(BP)
    POP(LP)
    JMP(LP,R31)
```

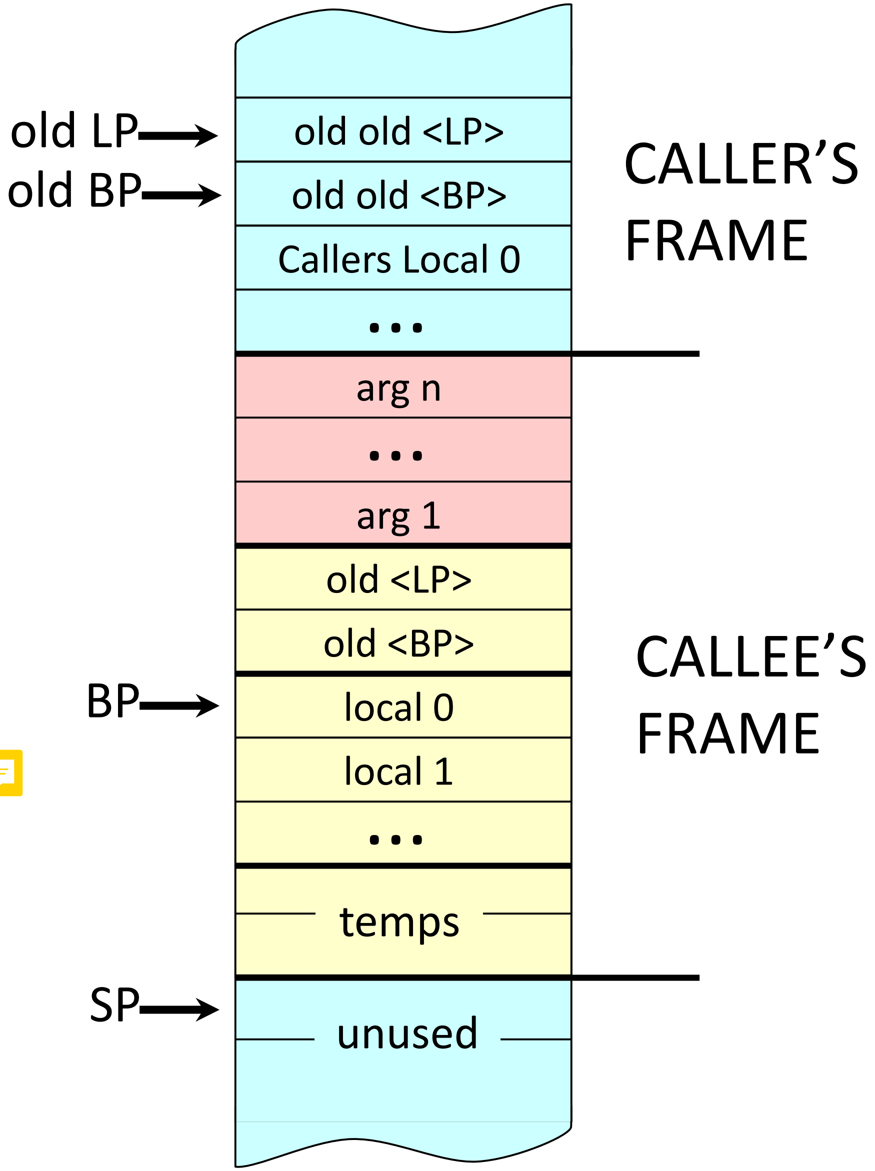
# Stack frames

**caller**

```
      PUSH(argn)
      ...
      PUSH(arg1)
      BR(f,LP)
      DEALLOCATE(n)
      ...
```

**callee**

```
f:    PUSH(LP)
      PUSH(BP)
      MOVE(SP,BP)
      ALLOCATE(nlocals)
      (push other regs)

      ...

      (pop other regs)
      MOVE(val,R0)
      MOVE(BP,SP)
      POP(BP)
      POP(LP)
      JMP(LP,R31)
```




caller pushes

callee pushes

# Example: Factorial

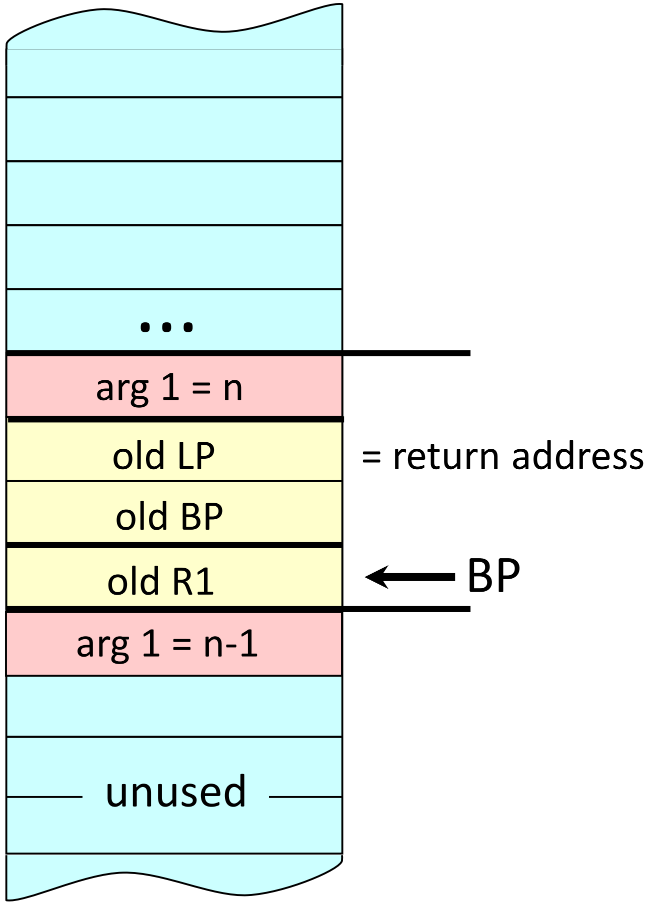
<u>C code:</u>

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

BP-12 is always the first argument of the function.
(BP-16 is the second argument)

<u>Beta assembler code:</u>

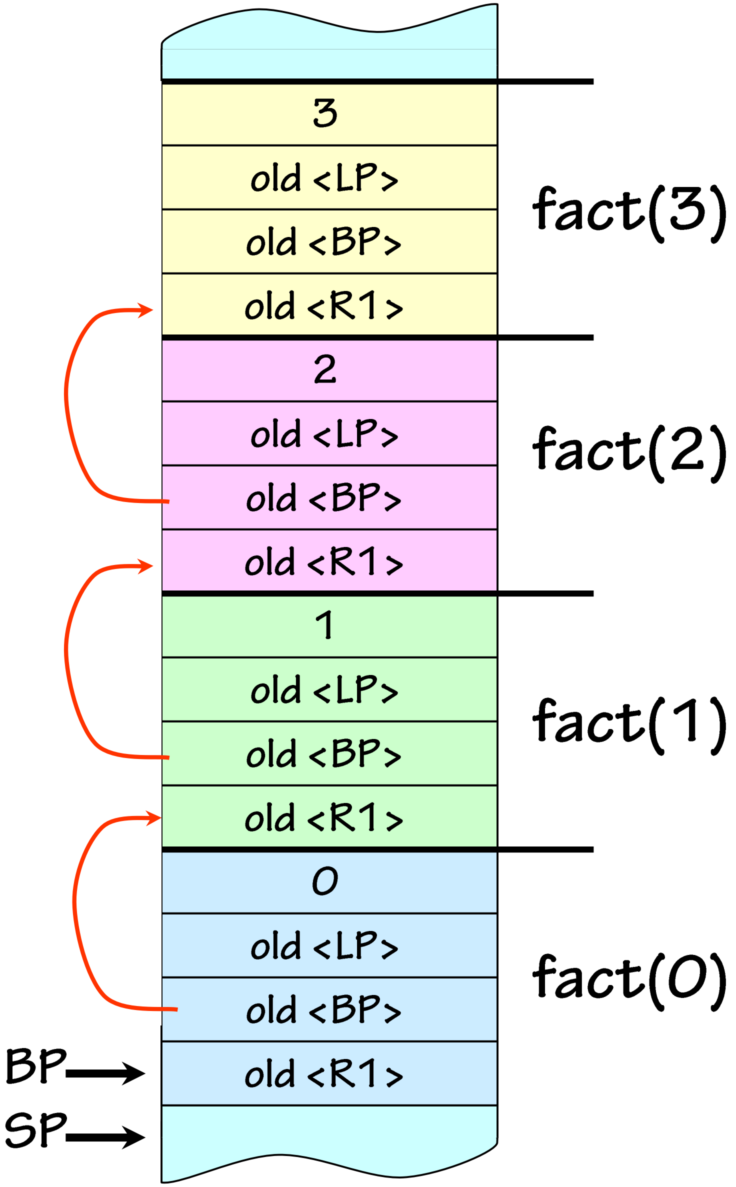
```
fact:   PUSH(LP)          | save linkages
        PUSH(BP)
        MOVE(SP,BP)       | new frame base
        PUSH(r1)          | preserve regs

        LD(BP,-12,r1)     | r1 ← n
        BNE(r1,big)       | if (n != 0)
        ADDC(r31,1,r0)    | else return 1;
        BR(rtn)

big:    SUBC(r1,1,r1)     | r1 ← (n-1)
        PUSH(r1)          | push arg1
        BR(fact,LP)       | fact(n-1)
        DEALLOCATE(1)     | pop arg1
        LD(BP,-12,r1)     | r1 ← n
        MUL(r1,r0,r0)     | r0 ← n*fact(n-1)

rtn:    POP(r1)           | restore regs
        MOVE(BP,SP)       | Why?
        POP(BP)           | restore links
        POP(LP)
        JMP(LP,R31)       | return.
```



# Example: Factorial

Beta assembler code:

```
fact:   PUSH(LP)            | save linkages
        PUSH(BP)
        MOVE(SP,BP)         | new frame base
        PUSH(r1)            | preserve regs

        LD(BP,-12,r1)       | r1 ← n
        BNE(r1,big)         | if (n != 0)
        ADDC(r31,1,r0)      | else return 1;
        BR(rtn)

big:    SUBC(r1,1,r1)       | r1 ← (n-1)
        PUSH(r1)            | push arg1
        BR(fact,LP)         | fact(n-1)
        DEALLOCATE(1)       | pop arg1
        LD(BP,-12,r1)       | r1 ← n
        MUL(r1,r0,r0)       | r0 ← n*fact(n-1)

rtn:    POP(r1)             | restore regs
        MOVE(BP,SP)         | Why?
        POP(BP)             | restore links
        POP(LP)
        JMP(LP,R31)         | return.
```



# Summary

- Overview of C language
- Compilation of C code into assembly code
- Compilation of procedures → stacks

- Quiz 2: Friday, October 19, 09:00-10:00
  - W3: Sequential logic, Finite state machines, Synchronization
  - W4: Computation Models, Programmable Machines, Beta architecture & Instruction Set
  - W5: Software Abstraction, Assembly Language, C language, Procedures & Stacks