Fundamentals of Computer Architecture

ARM - STACK

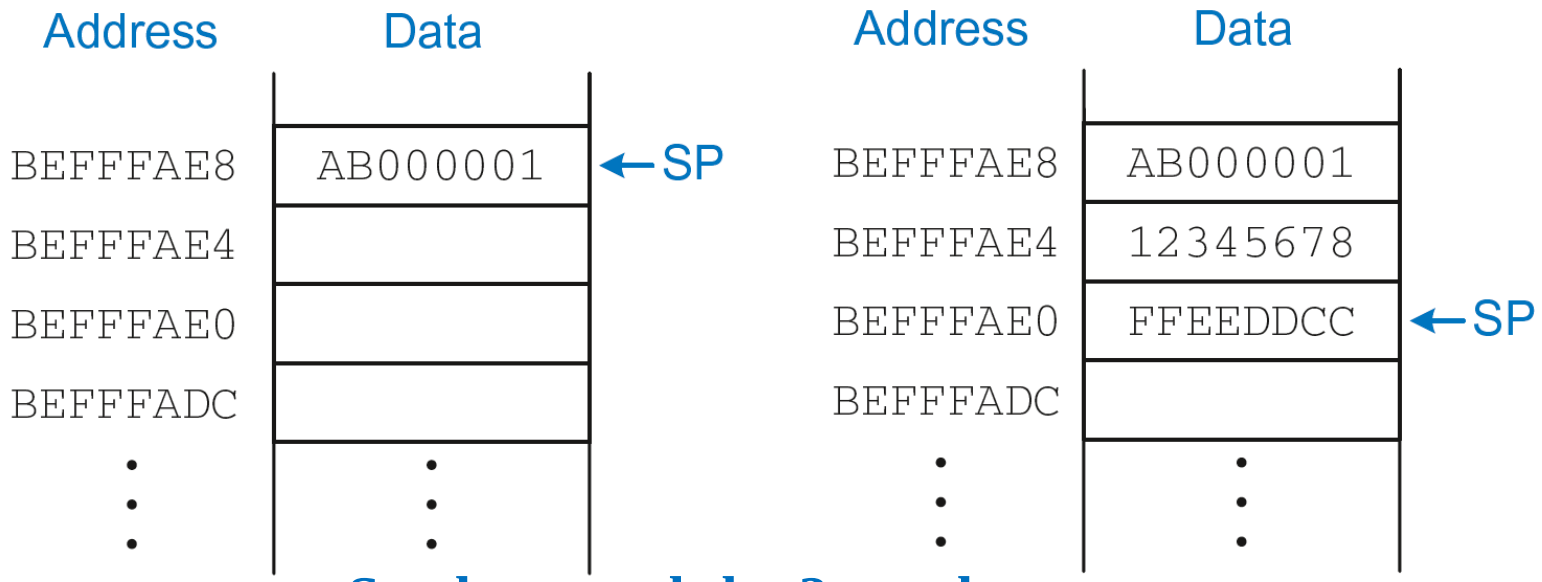
Teacher: Lobar Asretdinova

The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- *Expands*: uses more memory when more space needed
- *Contracts*: uses less memory when the space no longer needed

The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: `SP` points to top of the stack



Stack expands by 2 words

Input Arguments and Return Value

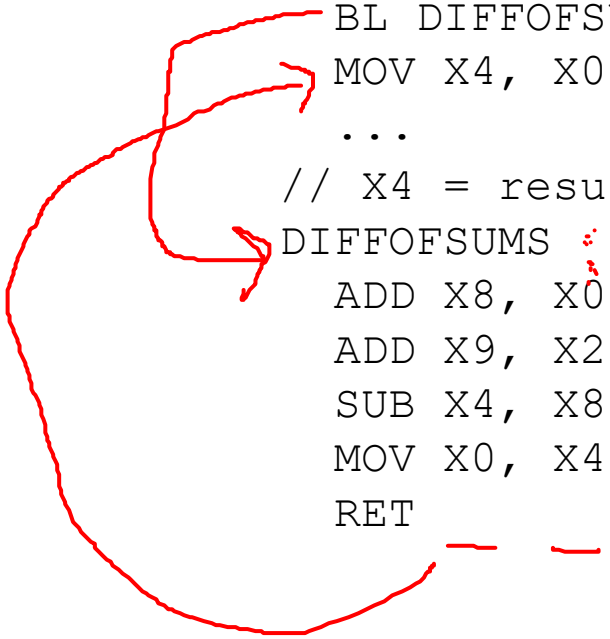
C Code

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;                // return value
}
```

Input Arguments and Return Value

ARM Assembly Code

```
// X4 = y
MAIN
   ...
   MOV X0, #2          ; argument 0 = 2
   MOV X1, #3          ; argument 1 = 3
   MOV X2, #4          ; argument 2 = 4
   MOV X3, #5          ; argument 3 = 5
   BL DIFFOFSUMS       ; call function
   MOV X4, X0          ; y = returned value
   ...
// X4 = result
DIFFOFSUMS
   ADD X8, X0, X1      ; X8 = f + g
   ADD X9, X2, X3      ; X9 = h + i
   SUB X4, X8, X9      ; result = (f + g) - (h + i)
   MOV X0, X4          ; put return value in X0
   RET                 ; return to caller
```



Input Arguments and Return Value

ARM Assembly Code

```
// X4 = result
DIFFOFSUMS
  ADD X8, X0, X1      ; X8 = f + g
  ADD X9, X2, X3      ; X9 = h + i
  SUB X4, X8, X9      ; result = (f + g) - (h + i)
  MOV X0, X4          ; put return value in X0
  RET                  ; return to caller
```

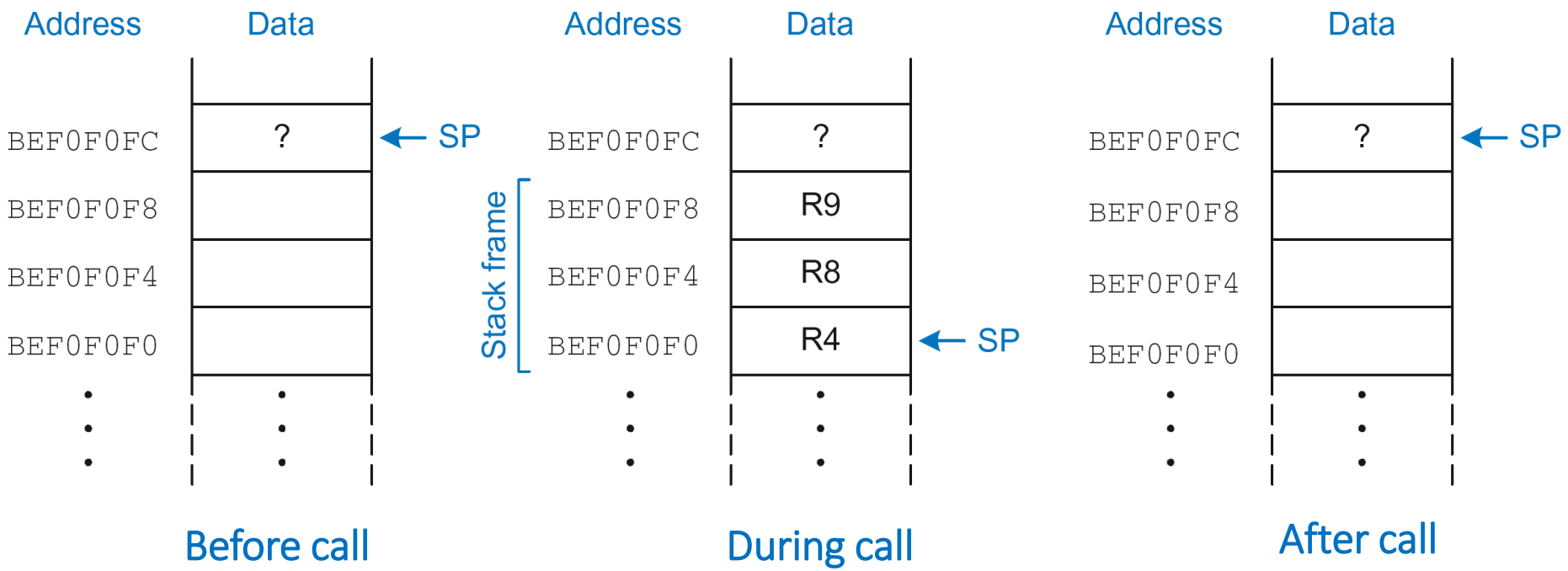
- `diffofsums` overwrote 3 registers: `X4`, `X8`, `X9`
- `diffofsums` can use *stack* to temporarily store registers

Storing Register Values on the Stack

ARM Assembly Code

```
; X2 = result
DIFFOFSUMS
  SUB SP, SP, #24     ; make space on stack for 3 registers
  STR X4, [SP, #-16]  ; save X4 on stack
  STR X8, [SP, #-8]   ; save X8 on stack
  STR X9, [SP]        ; save X9 on stack
  ADD X8, X0, X1      ; X8 = f + g
  ADD X9, X2, X3      ; X9 = h + i
  SUB X4, X8, X9      ; result = (f + g) - (h + i)
  MOV X0, X4          ; put return value in X0
  LDR X9, [SP]        ; restore X9 from stack
  LDR X8, [SP, #-8]   ; restore X8 from stack
  LDR X4, [SP, #-16]  ; restore X4 from stack
  ADD SP, SP, #       ; deallocate stack space
  RET                 ; return to caller
```

The Stack during `diffofsums` Call



Register Usage

- X9 to X17: temporary registers
  - Not preserved by the callee

- X19 to X28: saved registers
  - If used, the callee saves and restores them

Storing Saved Registers only on Stack

ARM Assembly Code

```
; X2 = result
DIFFOFSUMS
   STR X4, [SP, #-8]! ; save X4 on stack
   ADD X8, X0, X1     ; X8 = f + g
   ADD X9, X2, X3     ; X9 = h + i
   SUB X4, X8, X9     ; result = (f + g) - (h + i)
   MOV X0, X4         ; put return value in X0
   LDR X4, [SP], #8   ; restore X4 from stack
   RET                ; return to caller
```

Storing Saved Registers only on Stack

ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
  STR R4, [SP, #-4]! ; save R4 on stack
  ADD R8, R0, R1     ; R8 = f + g
  ADD R9, R2, R3     ; R9 = h + i
  SUB R4, R8, R9     ; result = (f + g) - (h + i)
  MOV R0, R4         ; put return value in R0
  LDR R4, [SP], #4   ; restore R4 from stack
  RET          ; return to caller
```

Notice code optimization for expanding/contracting stack

Nonleaf Function

ARM Assembly Code

```
STR LR, [SP, #-8]!      ; store LR on stack
BL  PROC2               ; call another function
...
LDR LR, [SP], #8        ; restore LR from stack
ret                     ; return to caller
```

Link Register → LR → x30

Nonleaf Function Example

C Code

```
int f1(int a, int b) {
  int i, x;

  x = (a + b)*(a - b);

  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}

int f2(int p) {
  int r;

  r = p + 5;
  return r + p;
}
```

Nonleaf Function Example

C Code

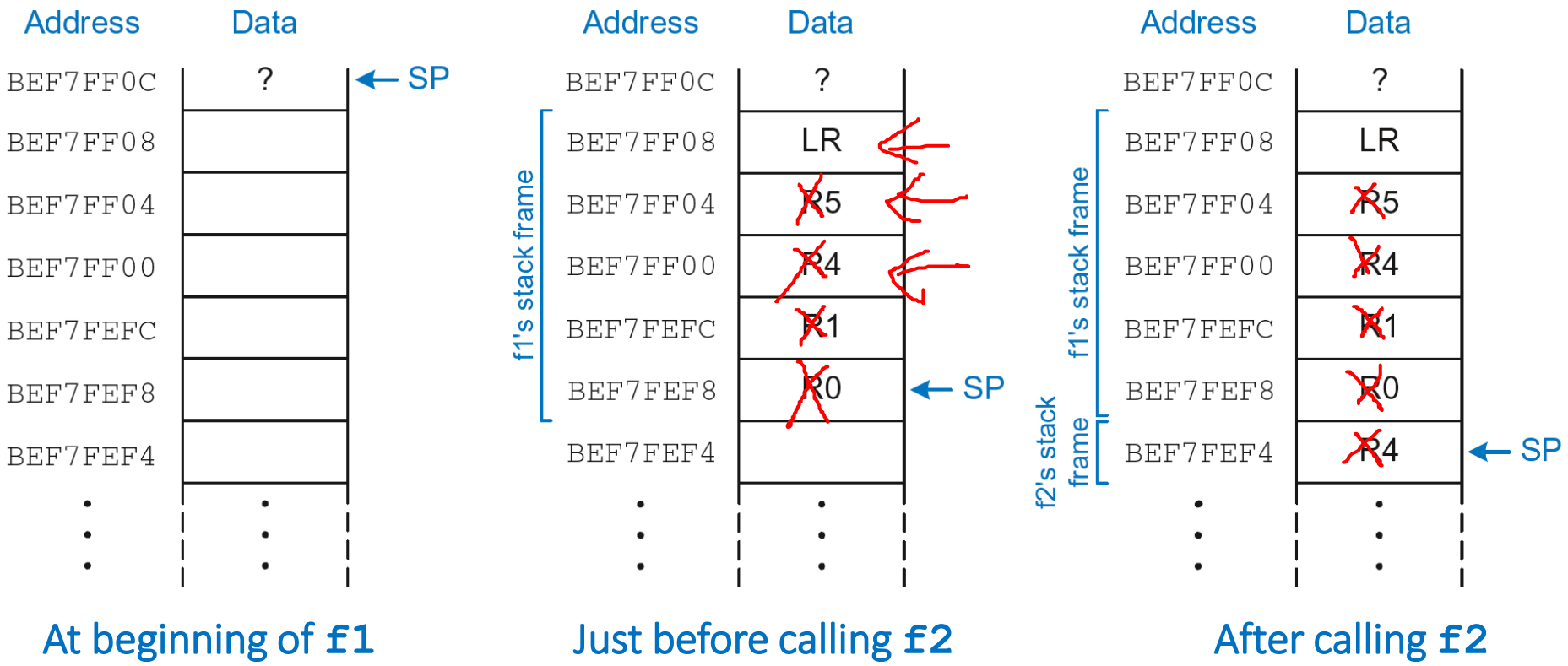
```
int f1(int a, int b) {
  int i, x;
  x = (a + b)*(a - b);
  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}
int f2(int p) {
  int r;
  r = p + 5;
  return r + p;
}
```

ARM Assembly Code

```
; X0=a, X1=b, X4=i, X5=x
F1:
  SUB SP, SP, #24
  STP X4, X5, [SP, #16]
  STR X30, [SP]
  ADD   X5,  X0, X1
  SUB   X12, X0, X1
  MUL   X5,  X5, X12
  MOV   X4,  #0
FOR:
  CMP   X4, X0
  BGE   RETURN
  PUSH {X0, X1}
  ADD   X0, X1, X4
  BL    F2
  ADD   X5, X5, X0
  POP  {X0, X1}
  ADD   X4, X4, #1
  B     FOR
RETURN:
  MOV   X0, X5
  LDR X30, [SP]
  LDP X4, X5, [SP, #16]
  ADD SP, SP, #24
  RET
```

```
; X0=p, X4=r
F2:
  SUB SP, SP, #8
  STR X4, [SP]
  ADD   X4, X0, 5
  ADD   X0, X4, X0
  LDR X4, [SP]
  ADD SP, SP, #8
  RET
```

Stack during Nonleaf Function



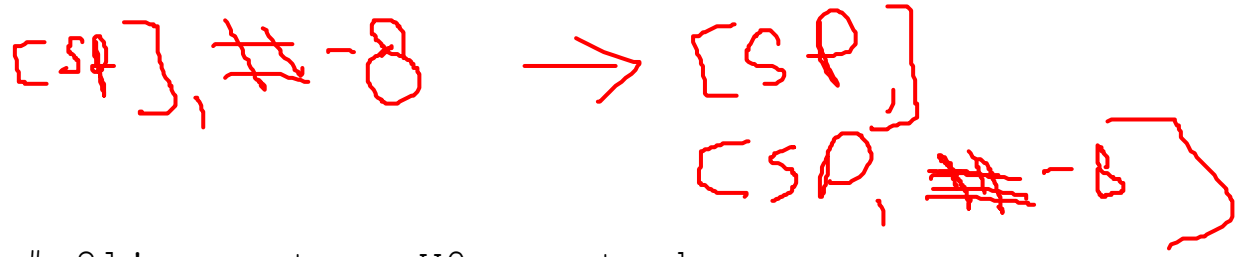
Recursive Function Call

C Code

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```

Recursive Function Call

ARM Assembly Code



```
0x94 FACTORIAL: STR X0, [SP, #-8]!    ;store X0 on stack
0x98            STR LR, [SP, #-8]!    ;store LR on stack
0x9C            CMP X0, #2            ;set flags with X0-2
0xA0            BHS ELSE              ;if (x0>=2) branch to else
0xA4            MOV X0, #1            ; otherwise return 1
0xA8            ADD SP, SP, #16       ; restore SP 1
0xAC            RET                   ; return
0xB0 ELSE:      SUB X0, X0, #1        ; n = n - 1
0xB4            BL  FACTORIAL         ; recursive call
0xB8            LDR LR, [SP], #8      ; restore LR
0xBC            LDR X1, [SP], #8      ; restore X0 (n) into X1
0xC0            MUL X0, X1, X0        ; X0 = n*factorial(n-1)
0xC4            RET                   ; return
```

Recursive Function Call

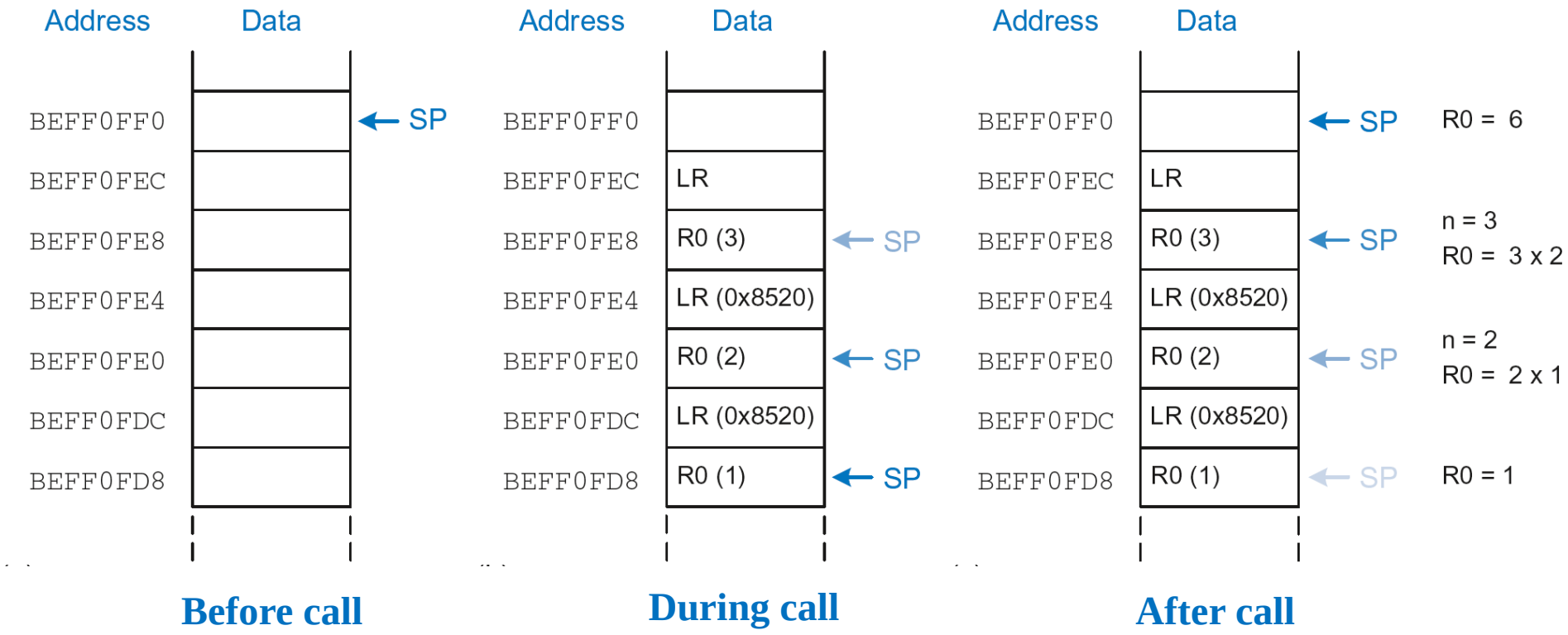
C Code

```
int factorial(int n) {

  if (n <= 1)
    return 1;




  else
    return (n * factorial(n-1));
}
```

ARM Assembly Code

```
0x94 FACTORIAL: STR X0, [SP, #-8]!
0x98            STR LR, [SP, #-8]!
0x9C            CMP R0, #2
0xA0            BHS ELSE
0xA4            MOV R0, #1
0xA8            ADD SP, SP, #16
0xAC            MOV PC, LR
0xB0 ELSE:      SUB X0, X0, #1
0xB4            BL  FACTORIAL
0xB8            LDR LR, [SP], #8
0xBC            LDR X1, [SP], #8
0xC0            MUL X0, X1, R0
0xC4            RET
```

Stack during Recursive Call



Function Call Summary

- **Caller**
  - Puts arguments in **x0-x7** for the first eight arguments. If there are more arguments, the rest go on the stack.
  - Saves any needed registers on the stack. This may include the callee-saved registers **(x19-x29)** if they are used by the caller.
  - Calls function: `BL CALLEE`
  - Restores registers: if necessary
  - Looks for result in **x0** (for integer return types) or **v0** (for floating-point return types).
- **Callee**
  - Saves registers that might be disturbed: **(x19-x29)** if they are used in the function.
  - Performs function
  - Puts result in **x0**
  - Restores registers
  - Returns: **RET**

Leaf Procedure Example

- C code:

```
long long int leaf_example (long long int g, long long
int h, long long int i, long long int j)
{ long long int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, …, j in X0, …, X3
  - f in X19 (hence, need to save $s0 on stack)

Leaf Procedure Example

- ARMv8 code:

```
leaf_example:
SUB SP,SP,#24
STR X10,[SP,#16]
STR X9,[SP,#8]
STR X19,[SP,#0]
ADD X9,X0,X1
ADD X10,X2,X3
SUB X19,X9,X10
ADD X0,X19,XZR
LDR X10,[SP,#16]
LDR X9,[SP,#8]
LDR X19,[SP,#0]
ADD SP,SP,#24
BR LR
```

Save X10, X9, X19 on stack

X9 = g + h

X10 = i + j

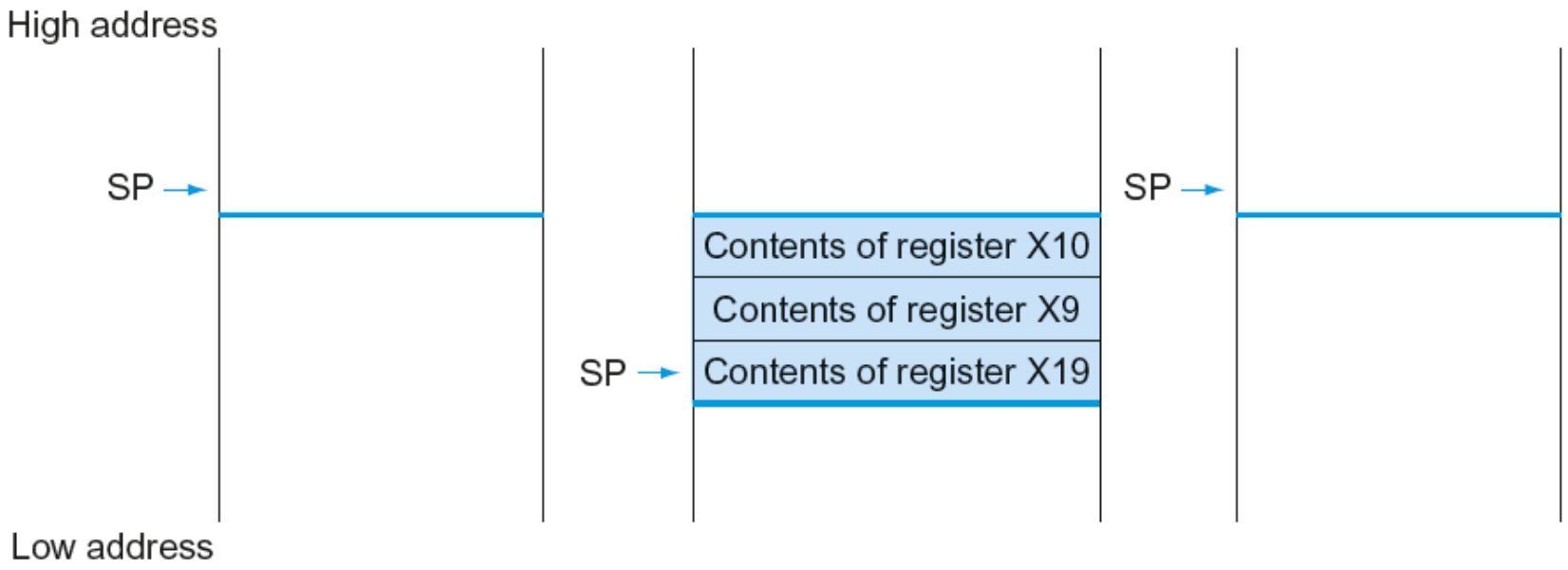
f = X9 – X10

copy f to return register

Resore X10, X9, X19 from stack

Return to caller

Local Data on the Stack



Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
  if (n < 1) return f;
  else return n * fact(n - 1);
}
```

  - Argument n in X0
  - Result in X1

Leaf Procedure Example

- ARMv8 code:

```
fact:
   SUB SP,SP,#16
   STR LR,[SP,#8]
   STR X0,[SP,#0]
   CMP X0, #1
   BGE L1
   ADD X1,XZR,#1
   ADD SP,SP,#16
   BR LR
L1: SUB X0,X0,#1
   BL fact
   LDR X0,[SP,#0]
   LDR LR,[SP,#8]
   ADD SP,SP,#16
   MUL X1,X0,X1
   BR LR
```

Save return address and n on stack

compare n and 1

if n >= 1, go to L1

Else, set return value to 1

Pop stack, don't bother restoring values

Return

n = n - 1

call fact(n-1)

Restore caller's n

Restore caller's return address

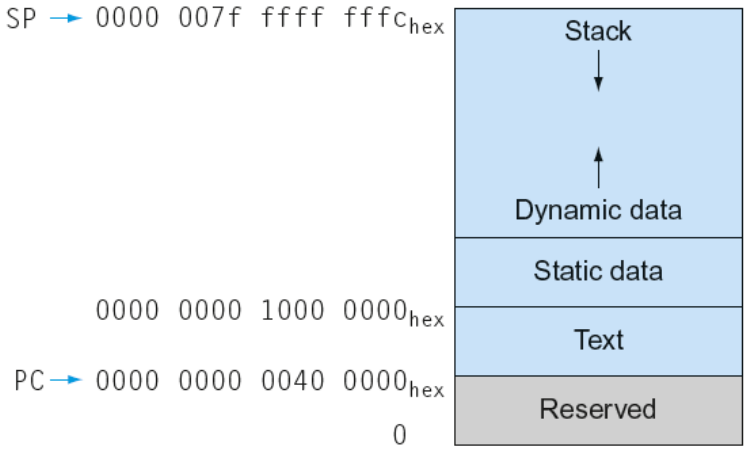
Pop stack

return n * fact(n-1)

return

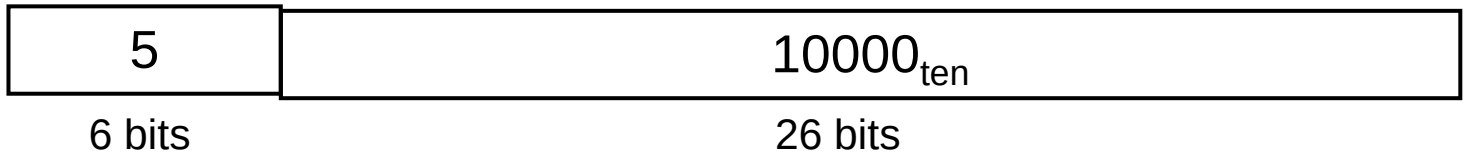
Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Branch Addressing

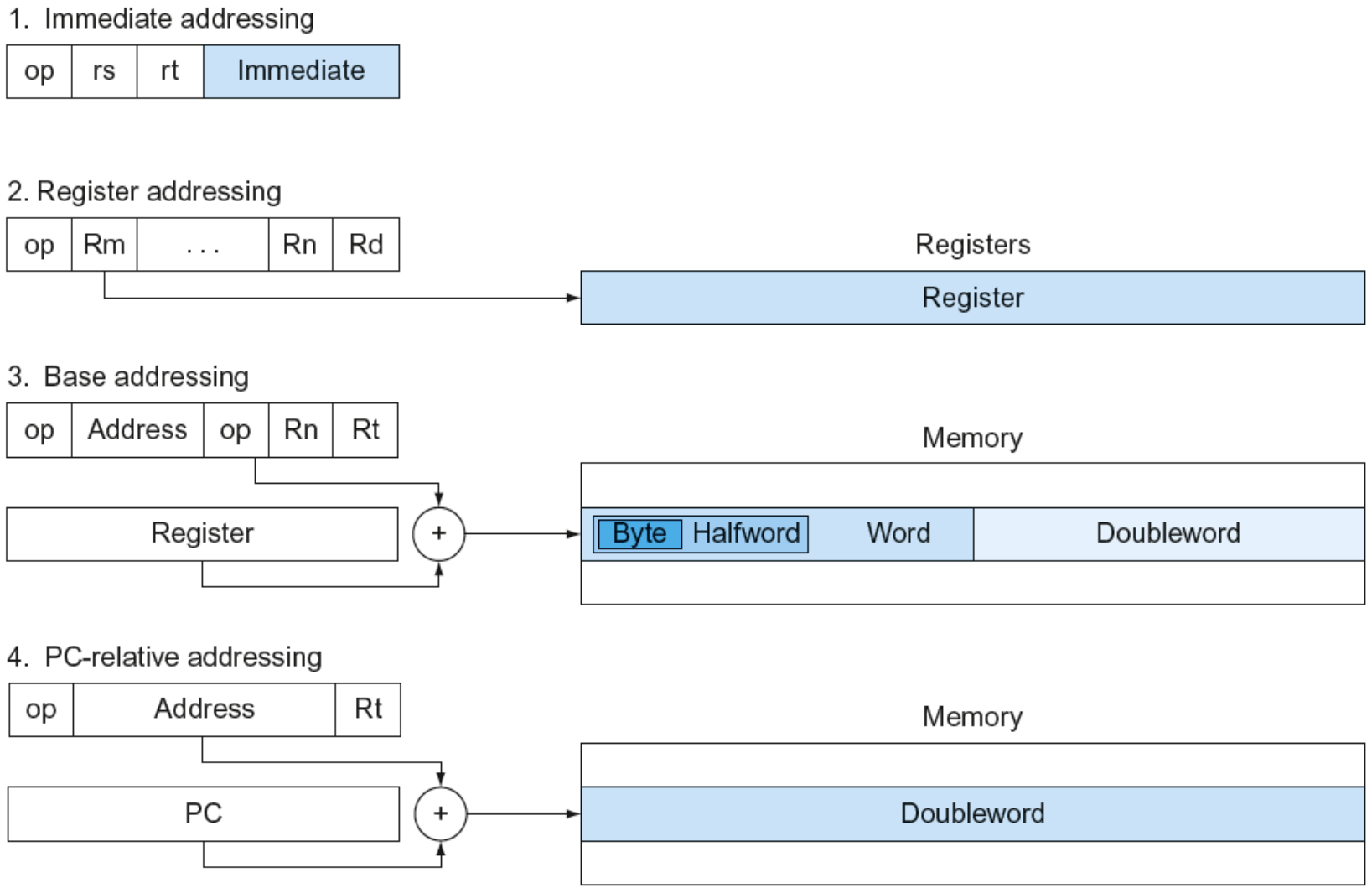
- B-type
  - `B 1000 // go to location 10000`$_{ten}$



- CB-type
  - `CBNZ X19, Exit // go to Exit if X19 != 0`

- Both addresses are PC-relative
  - Address = PC + offset (from instruction)

ARMv8 Addressing Summary



ARMv8 Encoding Summary

Name	Fields							Comments
Field size		6 to 11 bits	5 to 10 bits	5 or 4 bits	2 bits	5 bits	5 bits	All LEGv8 instructions are 32 bits long
R-format	R	opcode	Rm	shamt		Rn	Rd	Arithmetic instruction format
I-format	I	opcode	immediate			Rn	Rd	Immediate format
D-format	D	opcode	address		op2	Rn	Rt	Data transfer format
B-format	B	opcode	address					Unconditional Branch format
CB-format	CB	opcode	address				Rt	Conditional Branch format
IW-format	IW	opcode	immediate				Rd	Wide Immediate format