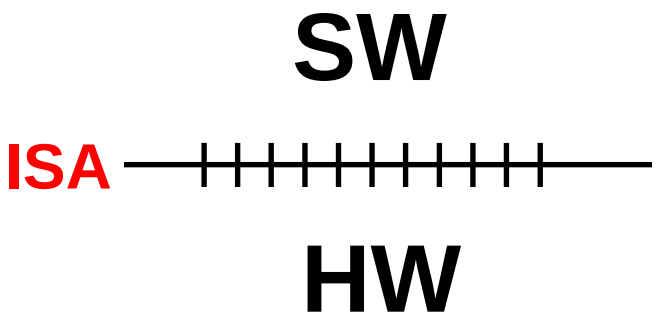
ENGN1640: Design of Computing Systems
Topic 03: Instruction Set Architecture Design

Professor Sherief Reda
http://scale.engin.brown.edu
School of Engineering
Brown University
Spring 2016

ISA is the HW/SW interface

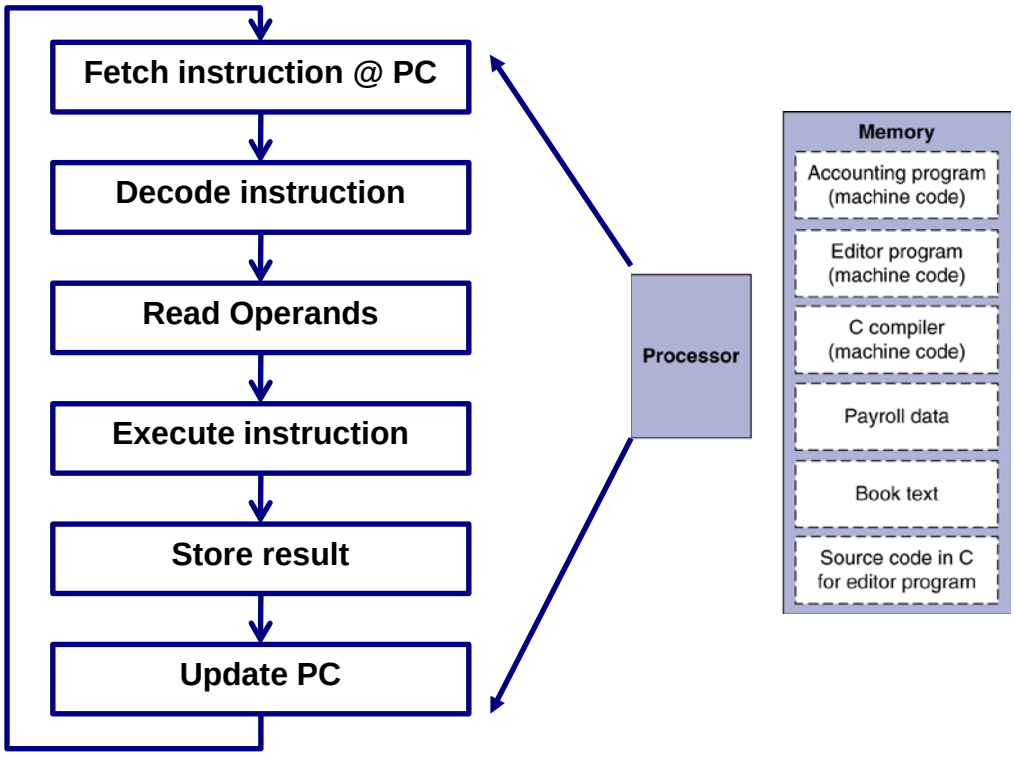


ISA choice determines:

- program size (& memory size)
- complexity of hardware (CPI and f)
- execution time for different applications and domains
- power consumption
- die area (cost)
- Backward compatibility

Stored program concept

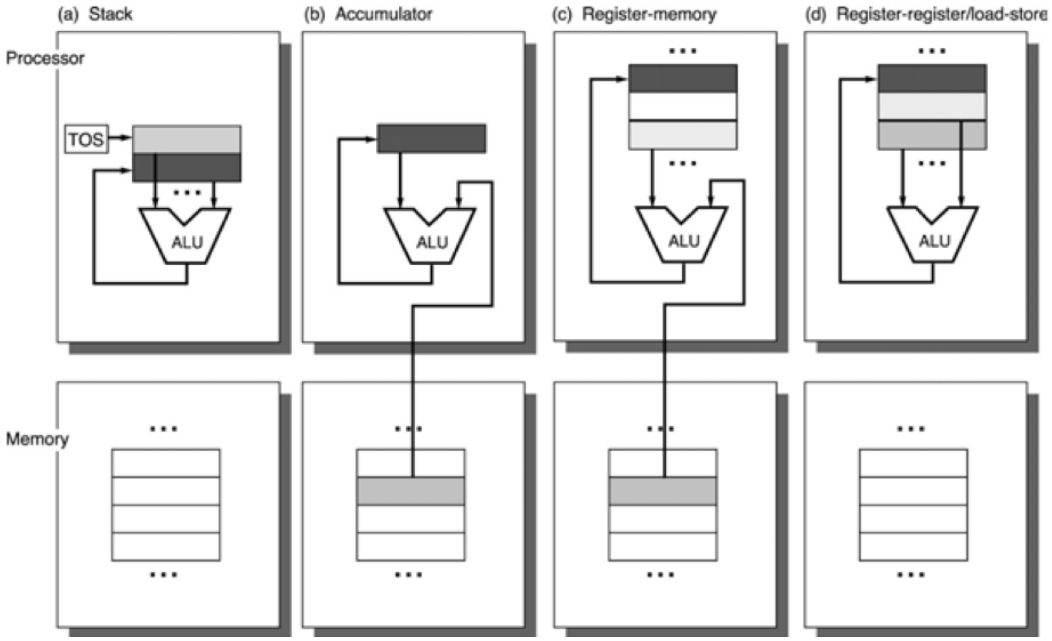
- Von Neumann model: Instructions represented in binary numbers, just like data → both in memory



ISA design issues:

- What is the instruction size?
- How is it encoded?
- Where are the operands located? What are their sizes and values?
- Where should the result be stored?
- How to determine the successor instruction?

1. Operand storage choices



C=A+B

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

- Memory is slow → need fast internal (but smaller) storage
- The international storage is one of the basic differentiation between ISAs.
- For register machines, how many registers are sufficient?
- What are the pros and cons of each method?

1. Pros and Cons of different register ISAs

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, TM32
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see Appendix C).	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs.
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

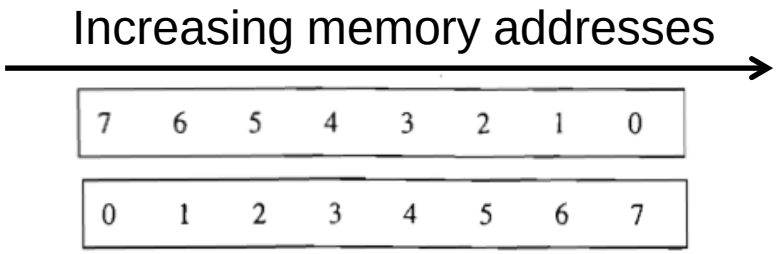
2. Memory addressing choices

- Data addressing modes

Addressing mode	Example instruction
Register	Add R4, R3
Immediate	Add R4, #3
Register Indirect	Add R4, (R1)
Displacement	Add R4, 100(R1)
Indexed	Add R3, (R1+R2)
Direct or absolute	Add R1, (1001)
Memory indirect	Add R4, @(R1)

What is the impact on instruction size, decoding and execution time?

- Big Endian
  vs. little Endian



3. Type and size of operands

- Character
- integer
- single-precision floating point
- double-precision floating point.
- Scalar / vector
- Types supported lead to variations of individual instructions

4. Operations in ISA

Operations	Examples
Arithmetic & logical	Integer arithmetic, logical operations: add, and, multiply, etc
Data transfer	Load-stores
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating point operations: add, multiple, divide, compare
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression, etc
Signal processing	MAC units, vector (SIMD) processing

4. Operations supported

Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

- **Simple instructions dominate instruction frequency**

- In Intel x86 ISA, 10 simple instructions account for 96% of integer programs → make the common case fast

What makes a good ISA?

- Efficiency of hardware implementation
- Convenience of programming / compiling
- Matches target applications (or alternatively generality)
- Compatibility and portability

Four design principles for ISA

1. Simplicity favors regularity
2. Smaller is faster
3. Make the common case fast
4. Good design demands good compromises

*ISA design is an **art**!*

Example: ARM ISA

- Billions of devices (e.g., smart phones, tables, etc) use ARM architecture.
- Example of register-register / load-store ISA
- 32 bit and 64 bit available
- All instructions are word aligned.
- All instructions could be conditionally executed.
- Most instructions execute in 1 cycle
- Will not cover all options in ISA but rather pick most used ones.

1. Register file

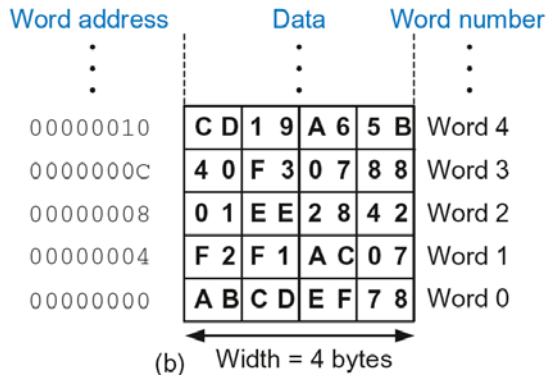
- ARM has 16 32-bit integer register file *(smaller is faster)*

Name	Use
R0	Argument / return value / temporary variable
R1-R3	Argument / temporary variables
R4-R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

In addition is also a status register (CPSR) → holds flags: results of arithmetic and logical operations.

2. Memory instructions

- Each data byte has unique address
- 32-bit word = 4 bytes, so word address increments by 4 and aligned
- Use little endian numbering
- Instructions:
  - Loads: LDR, LDRB, LDRH
  - Stores: STR, STRB, STRW
- Addresses are in bytes: can be written in decimal or hexadecimal (prefix address with 0x)



Addressing offset and indexing methods

	ARM Assembly	Memory Address
Rd←[Rn]	LDR R0, [R9]	R9
Rd←[Rn,#±imm]	LDR R0, [R3, #4]	R3 + 4
	LDR R0, [R5, #-16]	R5 – 16
Rd←[Rn,±Rm]	LDR R1, [R6, R7]	R6 + R7
	LDR R2, [R8, -R9]	R8 – R9
Rd←[Rn,±Rm,shft]	LDR R3, [R10, R11, LSL #2]	R10 + (R11 << 2)
	LDR R4, [R1, -R12, ASR #4]	R1 – (R12 >>> 4)

- **Indexing:**

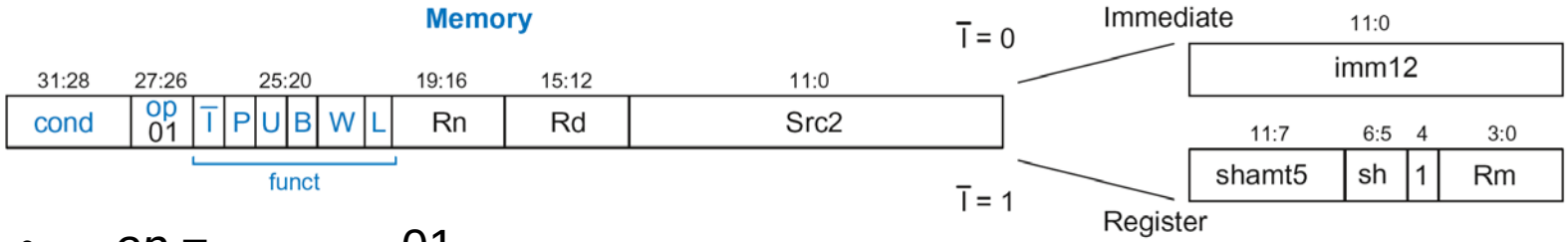
1. Preindex: Use ! to update Rn with memory access.

→ `LDR R3, [R5, #16]!  ; R3 = mem[R5 + 16]; R5 = R5 + 16;`

2. Postindex: Use [] on Rn to update after memory access.

→ `LDR R8, [R1], #8  ; R8 = mem[R1] ; R1 = R1 + 8`

Memory instruction format



- *op* = 01_2
- *Rn* = base register
- *Rd* = destination (load), source (store)
- *Src2* = offset: register (optionally shifted) or immediate
- *funct* = 6 control bits

L	*B*	Instruction
0	0	STR
0	1	STRB
1	0	LDR
1	1	LDRB

P	*W*	Indexing Mode
0	1	Not supported
0	0	Postindex
1	0	Offset
1	1	Preindex

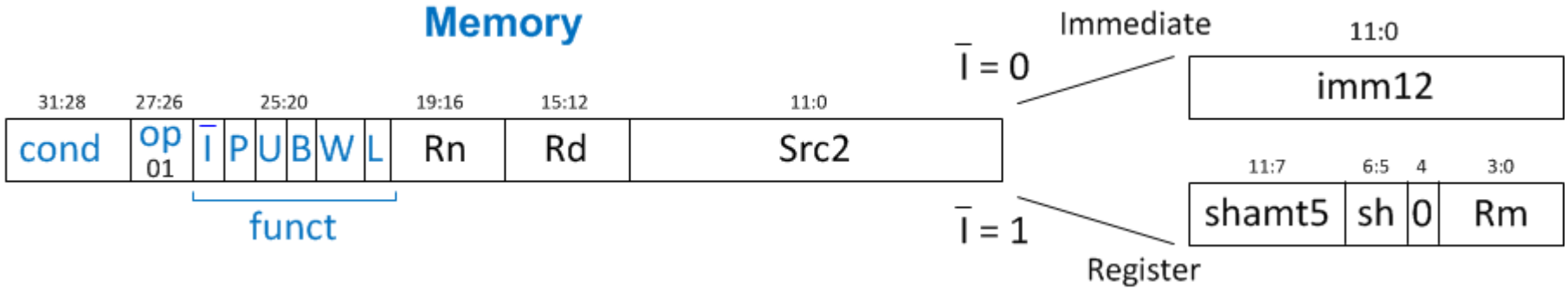
Value	*I*	*U*
0	**Immediate** offset in *Src2*	**Subtract** offset from base
1	**Register** offset in *Src2*	**Add** offset to base

Example 1 (register):

```
LDR R3, [R4, R5]
```

- **Operation:** R3 ← mem[R4 + R5]
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 01_2 (1) for memory instruction
- ***funct*** = 111001_2 (57)
  I = 1 (register offset), ***P*** = 1 (offset indexing),
  U = 1 (add), ***B*** = 0 (load **word**),
  W = 0 (offset indexing), ***L*** = 1 (load)
- ***Rd*** = 3, ***Rn*** = 4, ***Rm*** = 5 (***shamt5*** = 0, ***sh*** = 0)

1110 01 111001 0100 0011 00000 00 0 0101 = **0xE7943005**

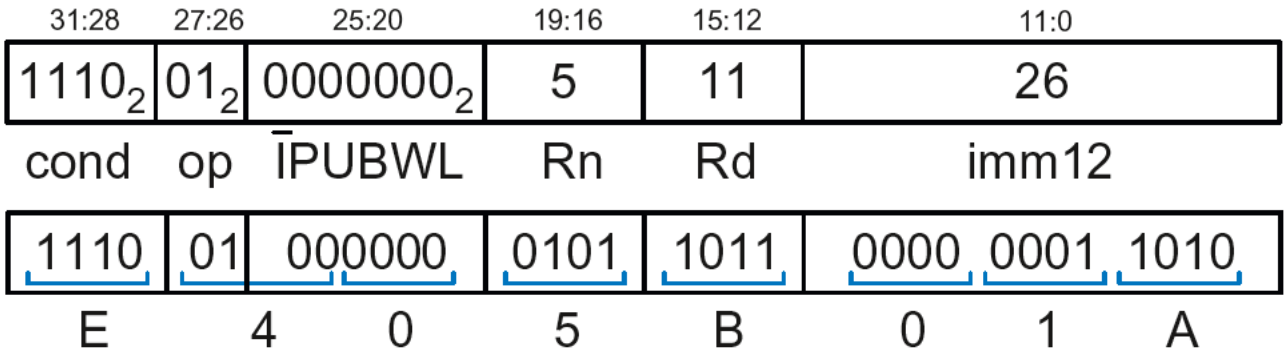


Example 2 (immediate):

```
STR R11, [R5], #-26
```

- **Operation:** R11←mem[R5]; R5 ← R5 - 26
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 01_2 (1) for memory instruction
- ***funct*** = 0000000_2 (0)
  $\overline{I}$ = 0 (immediate offset), ***P*** = 0 (postindex),
  U = 0 (subtract), ***B*** = 0 (store word),
  W = 0 (postindex), ***L*** = 0 (store)
- ***Rd*** = 11, ***Rn*** = 5, ***imm12*** = 26

Field Values

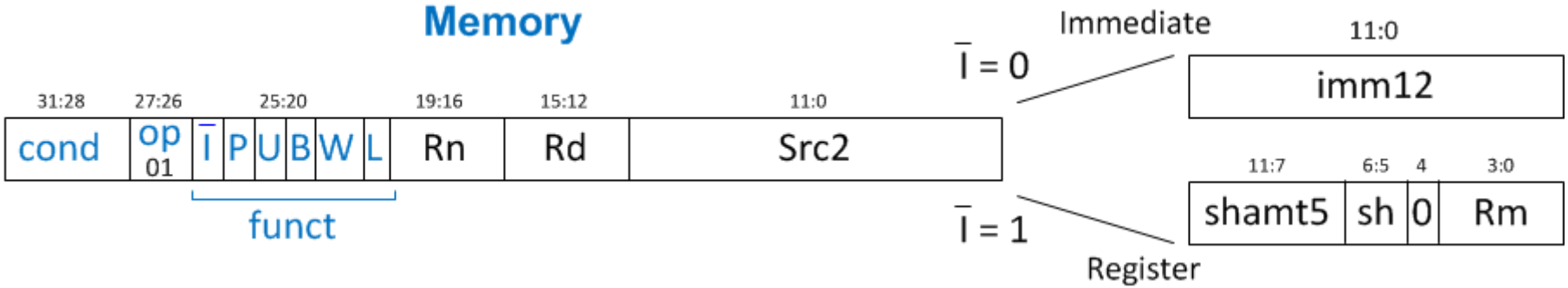


Example 3 (register shifted):

```
STR R9, [R1, R3, LSL #2]
```

- **Operation:** R9←mem[R1 + (R3 << 2)]
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 01_2 (1) for memory instruction
- ***funct*** = 111000_2 (0)
  $\overline{I}$ = 1 (register offset), ***P*** = 1 (offset indexing),
  U = 1 (add), ***B*** = 0 (store **word**), ***W*** = 0 (offset indexing),
  L = 0 (store)
- ***Rd*** = 9, ***Rn*** = 1, ***Rm*** = 3, ***shamt*** = 2, ***sh*** = 00_2 (LSL)

1110 01 111000 0001 1001 00010 00 0 0011 = **0xE7819103**



3.A Data processing operations

- **Movement**

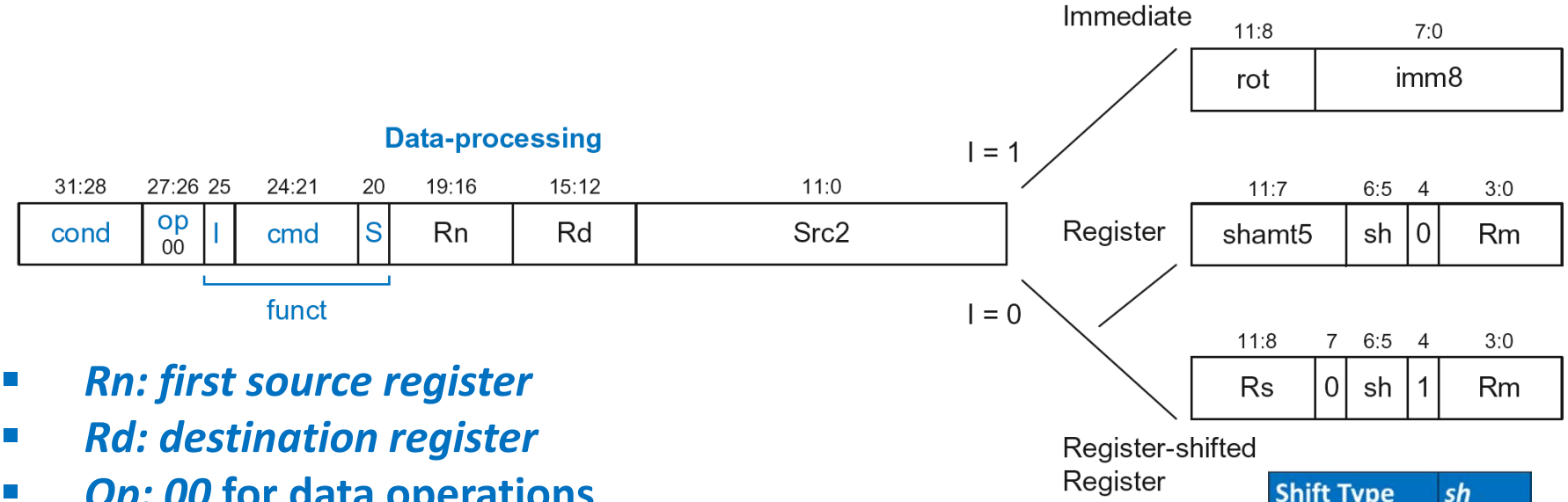
```
MOV R1, #0x45
MOV R1, #0xFF0
MOV R1, R0, LSL #3; R1 ← (R0 << 3)
MVN R7, R2
```

- **Arithmetic:** ADD, SUB, MUL
- **Logical** : AND, ORR, EOR, BIC (bit clear)
- **Shift/Rotation:** LSL, LSR, ASR, ROR

```
LSL R0, R7, #5                  ; R0 ← R7 << 5
ADD R0, R1, R2                  ; R0 ← R1 + R2
ASR R9, R11, R4                 ; R9 ← R11 >>> R4[7:0]
ORR R9, R5, R3, LSR #2          ; R9 ← R5 + (R3 >> 5)
EOR R8, R9, R10, ROR R12        ; R8←R9+(R10 ROR R12)
```

- First operand must be a register.
- Second operand may be: immediate or a register (just register, register shifted by immediate or register shifted *a register).*
- Immediate can only be 8 bits.

Encoding of data operation instructions



Shift Type	*sh*
LSL	00_2
LSR	01_2
ASR	10_2
ROR	11_2

- ***Rn: first source register***
- ***Rd: destination register***
- ***Op: 00* for data operations**
- ***cmd:*** is code of operation. (e.g., 0100_2 for ADD, 0010_2 for SUB, 1100_2 for ORR, 0001_2 for EOR, 1101 for all shift operations)
- ***S***-bit: 1 if sets condition flags
  - ***S*** = 0: `SUB R0, R5, R7`
  - ***S*** = 1: `ADDS R8, R2, R4`

Notice similarity to memory format: simplicity require regularity!

Example 1 (immediate):

```
ADD R0, R1, #42
```

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0100_2 (4) for ADD
- ***Src2*** is an immediate so ***I*** = 1
- ***Rd*** = 0, ***Rn*** = 1
- ***imm8*** = 42, ***rot*** = 0

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0100_2	0	1	0	0	42
cond	op	I	cmd	S	Rn	Rd	shamt5	sh Rm
1110	00	1	0100	0	0001	0000	0000	00101010

0xE281002A

Example 2 (immediate rotated):

```
SUB R2, R3, #0xFF0
```

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0010_2 (2) for SUB
- ***Src2*** is an immediate so ***I***=1
- ***Rd*** = 2, ***Rn*** = 3
- ***imm8*** = 0xFF
- ***imm8*** must be rotated right by 28 to produce 0xFF0, so ***rot*** = 14

ROR by 28 =
ROL by (32-28) = 4

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0010_2	0	3	2	14	255
cond	op	I	cmd	S	Rn	Rd	rot	imm8

0xE2432EFF

Example 3 (register):

```
ADD R5, R6, R7
```

- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0100_2 (4) for ADD
- ***Src2*** is a register so ***I***=0
- ***Rd*** = 5, ***Rn*** = 6, ***Rm*** = 7
- ***shamt*** = 0, ***sh*** = 0

Field Values

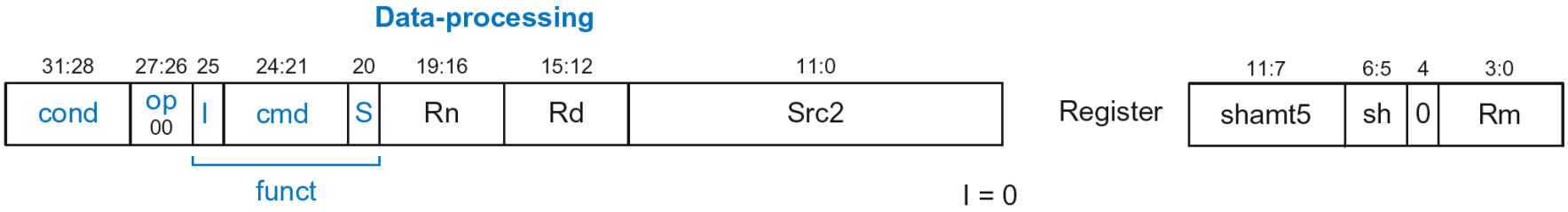
31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110_2	00_2	0	0100_2	0	6	5	0	0	0	7
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
1110	00	0	0100	0	0110	0101	00000	00	0	0111

0xE0865007

Example 3 (register shifted):

```
ORR R9, R5, R3, LSR #2
```

- **Operation:** R9 = R5 OR (R3 >> 2)
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 1100_2 (12) for ORR
- ***Src2*** is a register so ***I***=0
- ***Rd*** = 9, ***Rn*** = 5, ***Rm*** = 3
- ***shamt5*** = 2, ***sh*** = 01_2 (LSR)



1110 00 0 1100 0 0101 1001 00010 01 0 0011

0xE1859123

3.B Control flow operations

- Branches enable out of sequence instruction execution
- Types of branches:
  - **Branch (`B`)**
    - branches to another instruction
  - **Branch and link (`BL`)**
    - discussed later
- Both can be conditional or unconditional

Unconditional branching

ARM assembly

```
   MOV R2, #17          ; R2 = 17
   B   TARGET           ; branch  to  target
   ORR R1, R1, #0x4     ; not executed

TARGET:
      SUB R1, R1, #78       ; R1 = R1 + 78
```

Labels (like TARGET) indicate instruction location.
Labels can't be reserved words (like ADD, ORR, etc.)

Conditional branches

ARM Assembly

```
  MOV  R0, #4        ; R0 = 4
  ADD  R1, R0, R0    ; R1 = R0+R0 = 8
  CMP  R0, R1        ; sets flags with R0-R1
  BEQ  THERE         ; branch not taken (Z=0)
  ORR  R1, R1, #1    ; R1 = R1 OR R1 = 9
THERE:
  ADD R1, R1, 78     ; R1 = R1 + 78 = 87
```

Example if-else code

C Code

```
if (i == j)
  f = g + h;

else
  f = f – i;
```

ARM Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP R3, R4      ; set flags with R3-R4
 BNE L1          ; if i!=j, skip if block
 ADD R0, R1, R2  ; f = g + h
 B   L2          ; branch past else block
L1
 SUB R0, R0, R2  ; f = f – i
L2
```

Example: while loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;


while (pow != 128) {

  pow = pow * 2;
  x = x + 1;
}
```

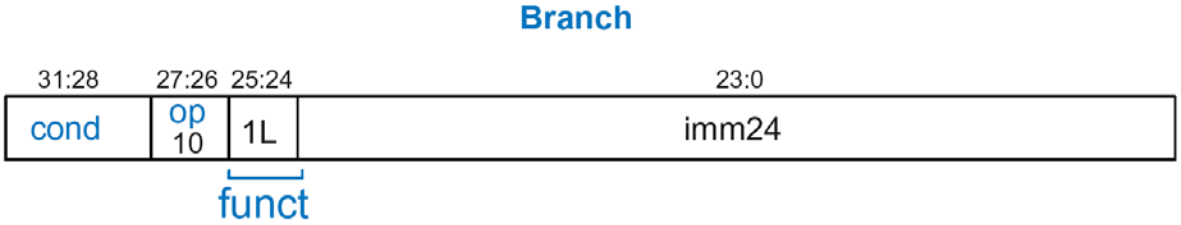
ARM Assembly Code

```
; R0 = pow, R1 = x
  MOV   R0, #1          ; pow = 1
  MOV   R1, #0          ; x = 0

WHILE:
  CMP R0, #128          ; R0-128
  BEQ DONE              ; if (pow==128)
                        ; exit loop
  LSL R0, R0, #1        ; pow=pow*2
  ADD R1, R1, #1        ; x=x+1
  B   WHILE             ; repeat loop

DONE:
```

Branch format

Encodes B and BL

- *op* = 10_2
- *imm24*: 24-bit immediate**; # of words BTA is away from PC+8**
- *funct* = $1L_2$: *L* = 0 for B, *L* = 1 for BL (branch & link)



Some condition codes:

0000: EQ: equal
0001: NE: not equal
0100: MI: negative
0101: PL: positive or zero
1010: GE: greater than or equal
1011: LT: less than
1100: GT: greater than
1101: LE: less than or equal

Conditional execution

Encode in *cond* bits of machine instruction

C Code

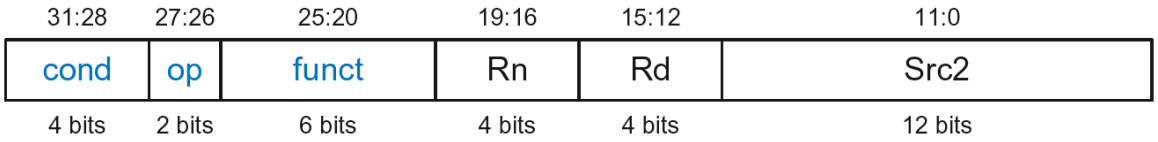
```
if (i == j)
  f = g + h;
else
   f = f – i;
```

ARM Assembly Code

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP   R3, R4      ; set flags with R3-R4
 ADDEQ R0, R1, R2  ; if (i==j) f = g + h
 SUBNE R0, R0, R2  ; else f = f - i
```

When to use?

Data-processing



EORLT R9, R5, R6	11	0	0	1	0	5	9	0	0	0	6
ADDEQ R4, R5, R6	0	0	0	4	0	5	4	0	0	0	6

Procedure/ function calls

- **Caller:**
  - passes **arguments** to callee (R0-R3)
  - branches to callee using branch and link (BL)
- **Callee:**
  - **performs** the function
  - **returns** result to caller in R0
  - **returns** to point of call by moving the link register to PC: MOV PC, LR
  - **must not overwrite** registers or memory needed by caller

Example

High-level 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
}
```

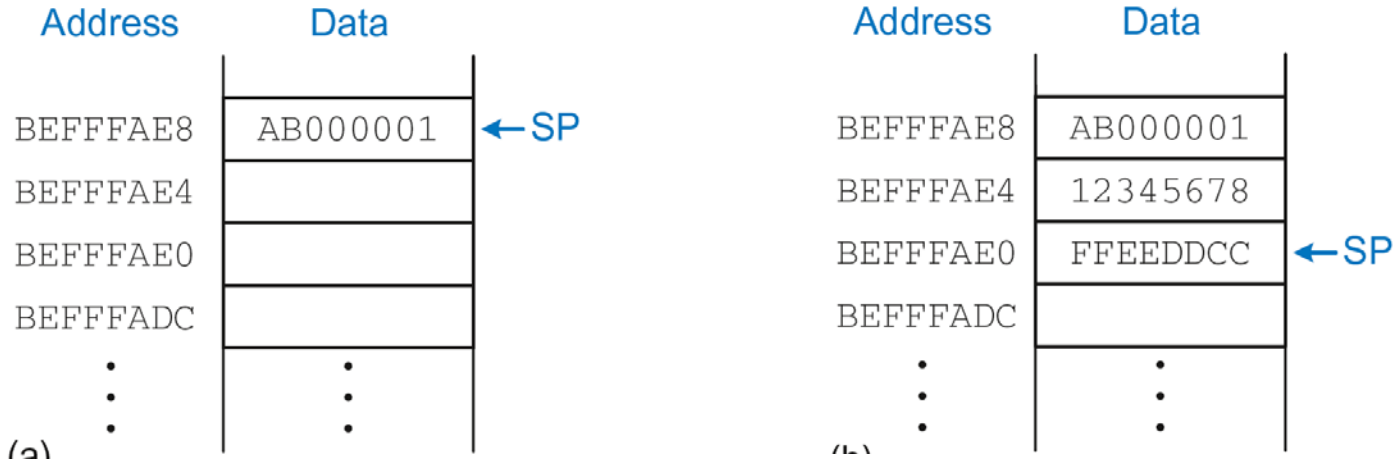
Example

```
; R4 = y
MAIN
  ...
  MOV R0, #2          ; argument 0 = 2
  MOV R1, #3          ; argument 1 = 3
  MOV R2, #4          ; argument 2 = 4
  MOV R3, #5          ; argument 3 = 5
  BL DIFFOFSUMS       ; call function
  MOV R4, R0          ; y = returned value
  ...
; R4 = result
DIFFOFSUMS:
  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
  MOV PC, LR          ; return to caller
```

- `diffofsums` overwrote 3 registers: R4, R8, R9
- `diffofsums` can use *stack* to temporarily store registers

The stack

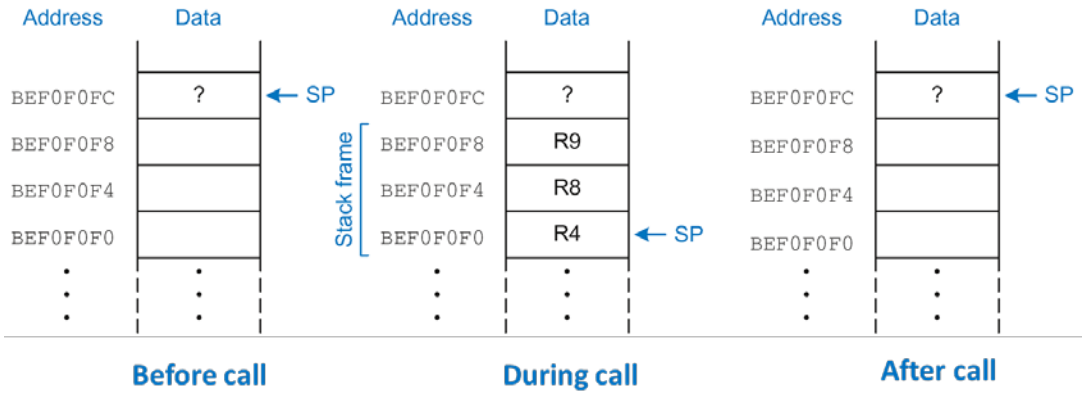
- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- ***Expands*:** uses more memory when more space needed
- ***Contracts*:** uses less memory when the space no longer needed
- Grows down (from higher to lower memory addresses)
- Stack pointer: SP points to top of the stack



Stack expands by 2 words

Storing register values on the stack

```
DIFFOFSUMS
  SUB SP, SP, #12      ; make space on stack for 3 registers
  STR R4, [SP, #-8]    ; save R4 on stack
  STR R8, [SP, #-4]    ; save R8 on stack
  STR R9, [SP]         ; save R9 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 R9, [SP]         ; restore R9 from stack
  LDR R8, [SP, #-4]    ; restore R8 from stack
  LDR R4, [SP, #-8]    ; restore R4 from stack
  ADD SP, SP, #12      ; deallocate stack space
  MOV PC, LR           ; return to caller
```



Can code be reduced?

Protocol for preserving registers

Preserved *Callee-Saved*	**Nonpreserved** *Caller-Saved*
R4-R11	**R12**
R14 (LR)	**R0-R3**
R13 (SP)	**CPSR**
stack above SP	**stack below SP**

Recursive procedure calls

High-level code

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

What is the potential problem with recursive or nested function calls?

Recursive procedure calls

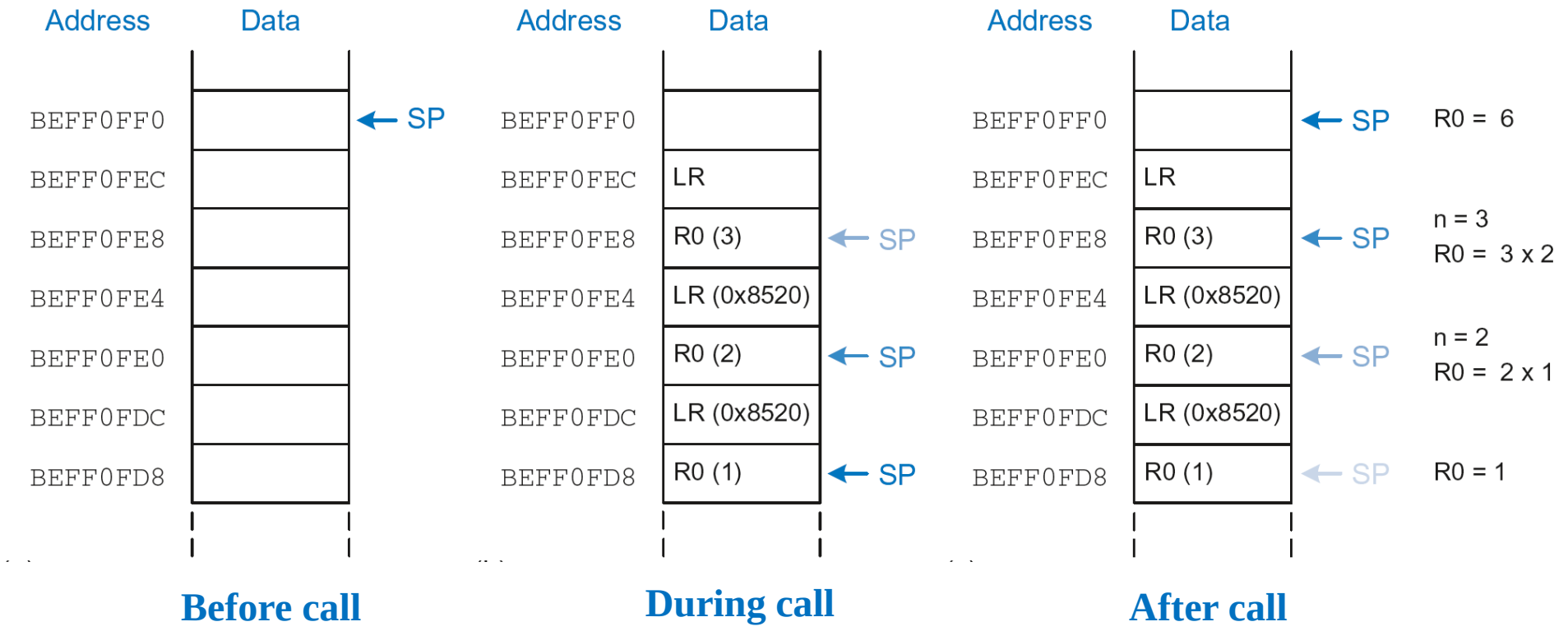
C Code

```
int factorial(int n) {

  if (n <= 1)
    return 1;




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

ARM Assembly Code

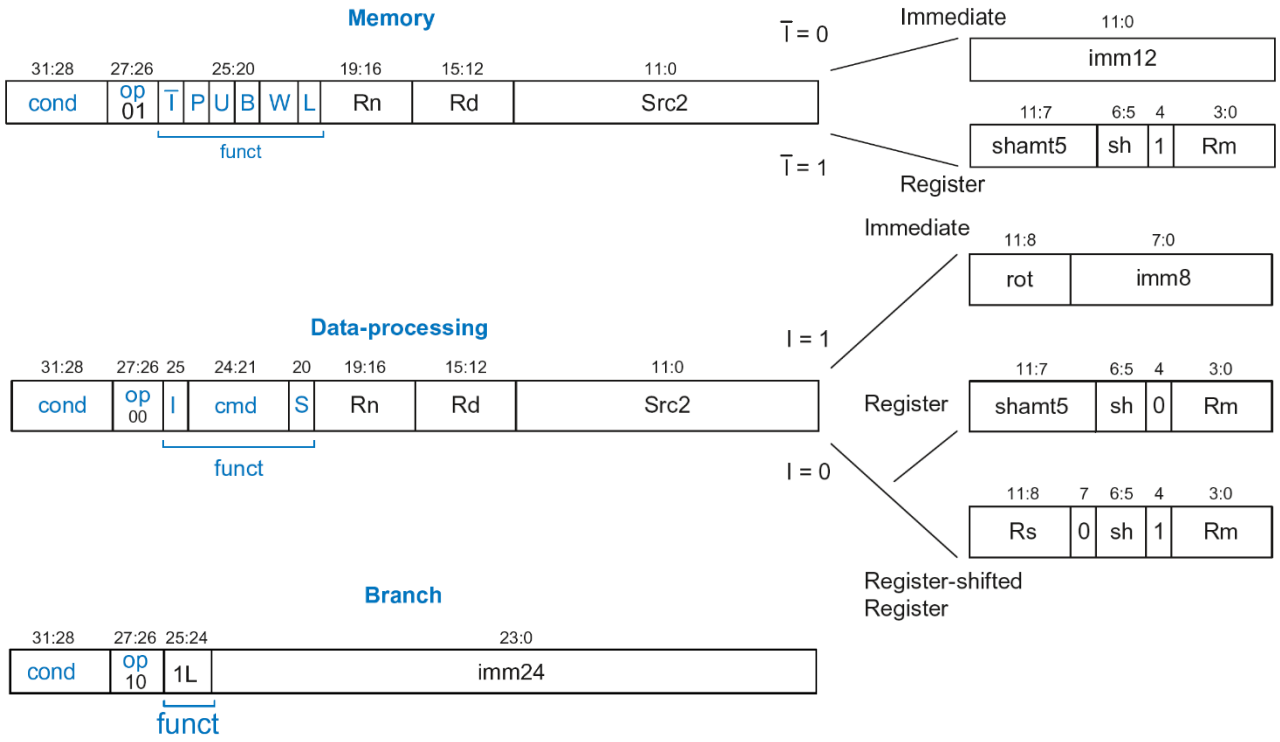
```
0x94 FACTORIAL  STR R0, [SP, #-4]!
0x98            STR LR, [SP, #-4]!
0x9C            CMP R0, #2
0xA0            BHS ELSE
0xA4            MOV R0, #1
0xA8            ADD SP, SP, #8
0xAC            MOV PC, LR
0xB0 ELSE       SUB R0, R0, #1
0xB4            BL  FACTORIAL
0xB8            LDR LR, [SP], #4
0xBC            LDR R1,[SP], #4
0xC0            MUL R0, R1, R0
0xC4            MOV PC, LR
```

Stack during recursive calls



Summary

- Overviewed most relevant instructions in ARM ISA.



- Additional instructions exist for various versions: Thumb extensions, floating point and SIMD NEON extensions.