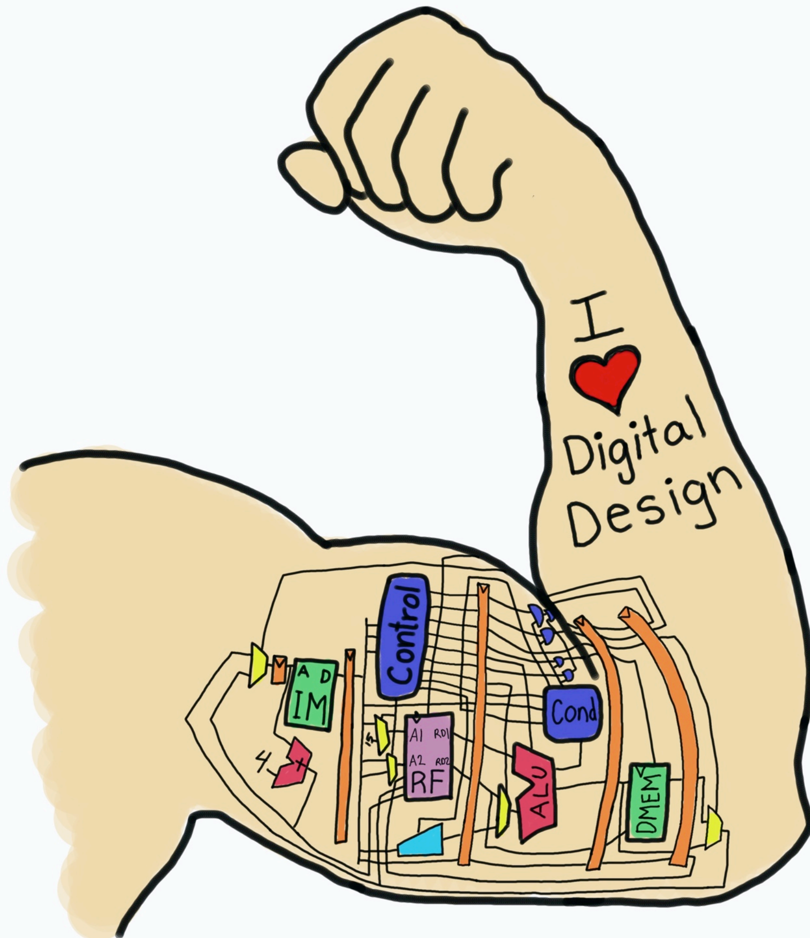
E85 Digital Design & Computer Engineering



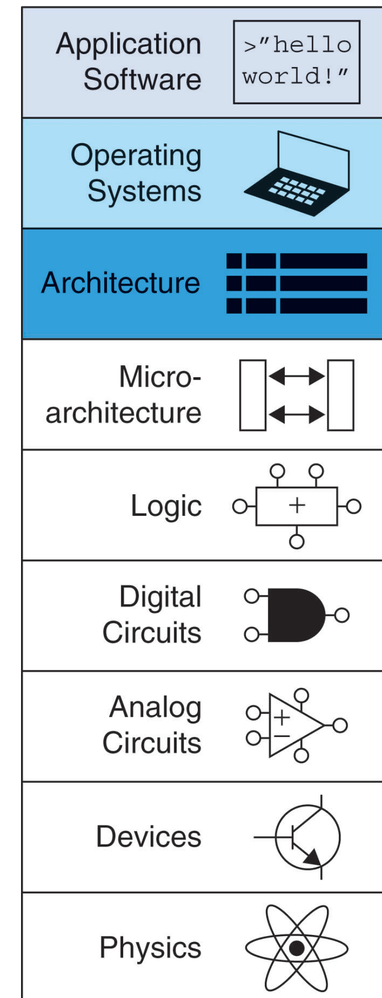
Lecture 18:

Function Calls & Machine Language

HARVEY MUDD COLLEGE

Lecture 18

- **Function Calls**
- **Machine Language**



Programming Building Blocks

- **Data-processing Instructions**
- **Conditional Execution**
- **Branches**
- **High-level Constructs:**
  - if/else statements
  - for loops
  - while loops
  - arrays
  - **function calls**

ELSEVIER

Function Calls

- **Caller:** calling function (in this case, `main`)
- **Callee:** called function (in this case, `sum`)

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

ELSEVIER

Function Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee

ELSEVIER

Function Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee
- **Callee:**
  - **performs** the function
  - **returns** result to caller
  - **returns** to point of call
  - **must not overwrite** registers or memory needed by caller

ELSEVIER

ARM Function Conventions

- **Call Function:** branch and link

  `BL`

- **Return** from function: move the link register to PC: `MOV PC, LR`
- **Arguments:** `R0-R3`
- **Return value:** `R0`

ELSEVIER

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

ARM Assembly Code

```
0x00000200 MAIN     BL   SIMPLE
0x00000204          ADD R4, R5, R6
...

0x00401020 SIMPLE   MOV PC, LR
```

`void` means that `simple` doesn't return a value

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

ARM Assembly Code

```
0x00000200 MAIN     BL  SIMPLE
0x00000204          ADD R4, R5, R6
...

0x00401020 SIMPLE   MOV PC, LR
```

BL branches to SIMPLE
LR = PC + 4 = 0x00000204

MOV PC, LR makes PC = LR
(the next instruction executed is at 0x00000200)

ELSEVIER

Input Arguments and Return Value

ARM conventions:

- Argument values: `R0` - `R3`
- Return value: `R0`

ELSEVIER

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

```
; 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
```

ELSEVIER

Input Arguments and Return Value

ARM Assembly Code

```
; 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

ELSEVIER

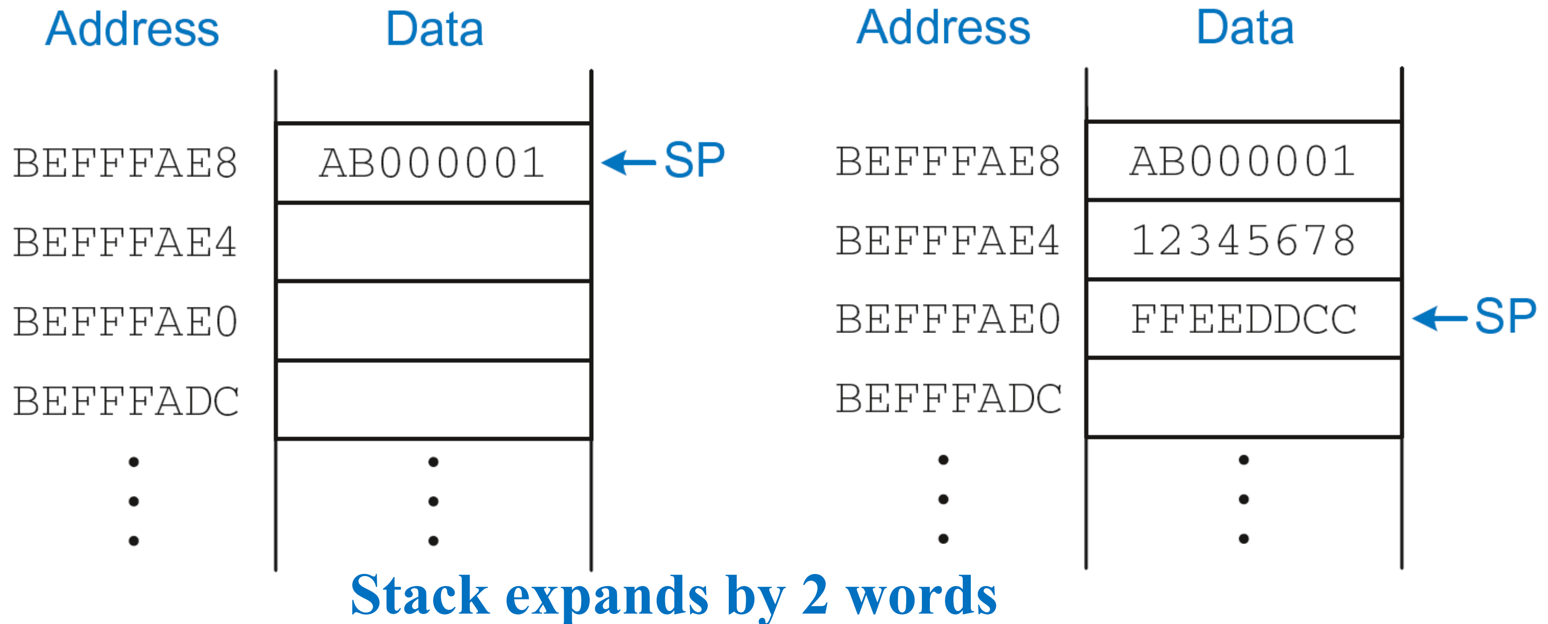
The Stack

- 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

ELSEVIER

The Stack

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



How Functions use the Stack

- Called functions must have no unintended side effects
- But `diffofsums` overwrites 3 registers: `R4`, `R8`, `R9`

ARM Assembly Code

```
; 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
```

Storing Register Values on the Stack

ARM Assembly Code

```
; R2 = result
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
```

The Stack during `diffofsums` Call

Before call

Address	Data	
BEF0F0FC	?	← SP
BEF0F0F8		
BEF0F0F4		
BEF0F0F0		
⋮	⋮	

During call

	Address	Data	
	BEF0F0FC	?	
Stack frame	BEF0F0F8	R9	
Stack frame	BEF0F0F4	R8	
Stack frame	BEF0F0F0	R4	← SP
	⋮	⋮	

After call

Address	Data	
BEF0F0FC	?	← SP
BEF0F0F8		
BEF0F0F4		
BEF0F0F0		
⋮	⋮	

Registers

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

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
  MOV PC, LR         ; return to caller
```

ELSEVIER

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
   MOV PC, LR         ; return to caller
```

Notice code optimization for expanding/contracting stack

Nonleaf Function

ARM Assembly Code

```
STR LR, [SP, #-4]!       ; store LR on stack
BL  PROC2                ; call another function
...
LDR LR, [SP], #4         ; restore LR from stack
MOV PC, LR               ; return to caller
```

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;
}
```

ELSEVIER

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

```
; R0=a, R1=b, R4=i, R5=x
F1
  PUSH {R4,  R5, LR}
  ADD   R5,  R0, R1
  SUB   R12, R0, R1
  MUL   R5,  R5, R12
  MOV   R4,  #0
FOR
  CMP   R4, R0
  BGE   RETURN
  PUSH {R0, R1}
  ADD   R0, R1, R4
  BL    F2
  ADD   R5, R5, R0
  POP  {R0, R1}
  ADD   R4, R4, #1
  B     FOR
RETURN
  MOV   R0, R5
  POP  {R4, R5, LR}
  MOV   PC, LR
```

```
; R0=p, R4=r
F2
  PUSH {R4}
  ADD   R4, R0, 5
  ADD   R0, R4, R0
  POP  {R4}
  MOV   PC, LR
```

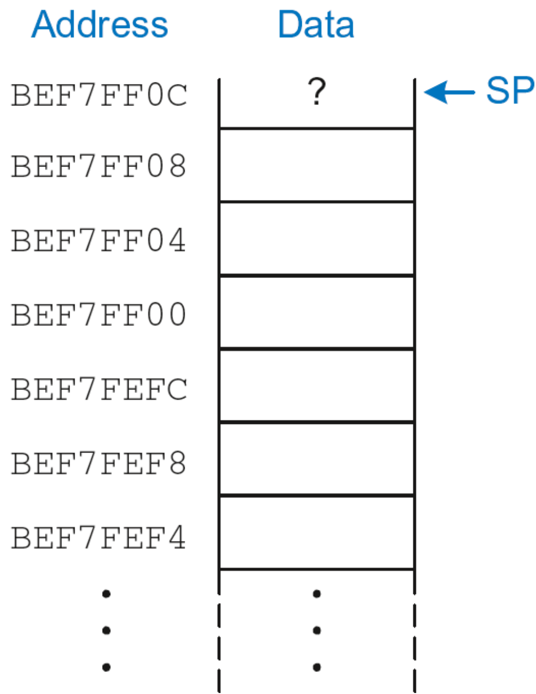
Nonleaf Function Example

ARM Assembly Code

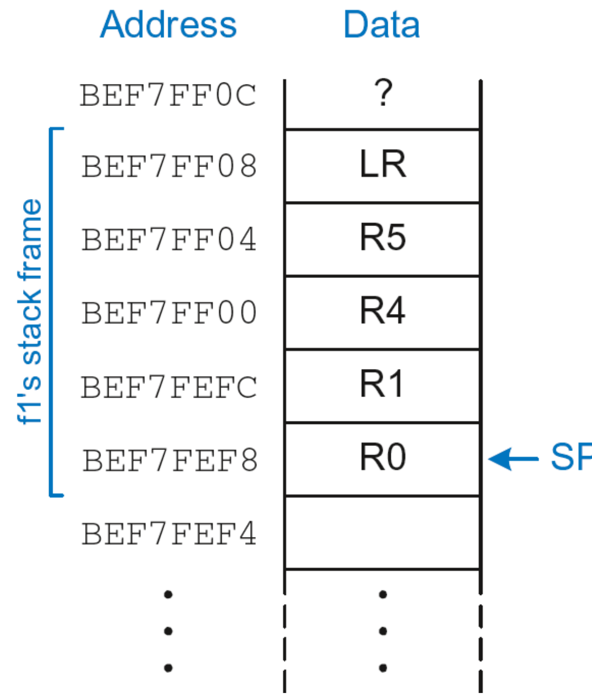
```
; R0=a, R1=b, R4=i, R5=x
F1
  PUSH {R4,  R5, LR} ; save regs
  ADD   R5,  R0, R1  ; x = (a+b)
  SUB   R12, R0, R1  ; temp = (a-b)
  MUL   R5,  R5, R12 ; x = x*temp
  MOV   R4,  #0      ; i = 0
FOR
  CMP   R4, R0       ; i < a?
  BGE   RETURN       ; no: exit loop
  PUSH {R0, R1}      ; save regs
  ADD   R0, R1, R4   ; arg is b+i
  BL    F2           ; call f2(b+i)
  ADD   R5, R5, R0   ; x = x+f2(b+i)
  POP  {R0, R1}      ; restore regs
  ADD   R4, R4, #1   ; i++
  B     FOR          ; repeat loop
RETURN
  MOV   R0, R5       ; return x
  POP  {R4, R5, LR}  ; restore regs
  MOV   PC, LR       ; return
```

```
; R0=p, R4=r
F2
  PUSH {R4}          ; save regs
  ADD   R4, R0, 5    ; r = p+5
  ADD   R0, R4, R0   ; return r+p
  POP  {R4}          ; restore regs
  MOV   PC, LR       ; return
```

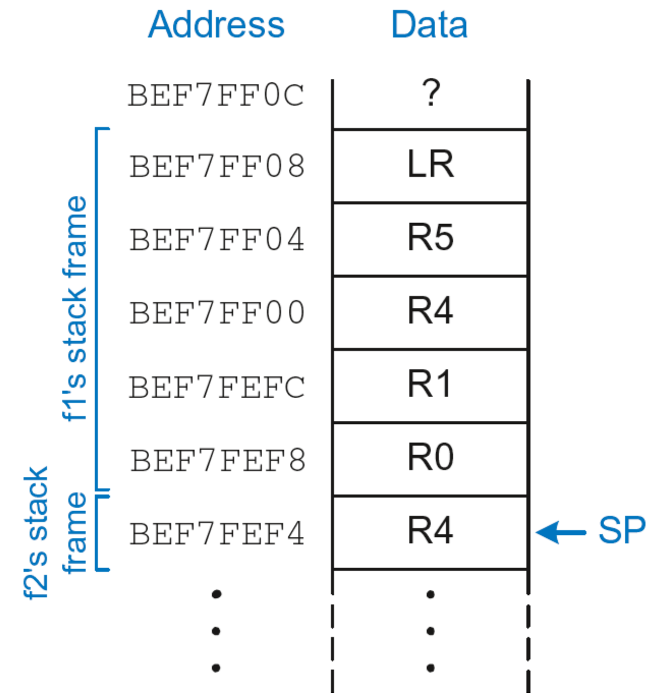
Stack during Nonleaf Function



At beginning of f1



Just before calling f2



After calling f2

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 R0, [SP, #-4]!    ;store R0 on stack
0x98            STR LR, [SP, #-4]!    ;store LR on stack
0x9C            CMP R0, #2            ;set flags with R0-2
0xA0            BHS ELSE              ;if (r0>=2) branch to else
0xA4            MOV R0, #1            ; otherwise return 1
0xA8            ADD SP, SP, #8        ; restore SP 1
0xAC            MOV PC, LR            ; return
0xB0 ELSE       SUB R0, R0, #1        ; n = n - 1
0xB4            BL  FACTORIAL         ; recursive call
0xB8            LDR LR, [SP], #4      ; restore LR
0xBC            LDR R1, [SP], #4      ; restore R0 (n) into R1
0xC0            MUL R0, R1, R0        ; R0 = n*factorial(n-1)
0xC4            MOV PC, LR            ; 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 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 Call

Before call

Address	Data	
BEFF0FF0		← SP
BEFF0FEC		
BEFF0FE8		
BEFF0FE4		
BEFF0FE0		
BEFF0FDC		
BEFF0FD8		

During call

Address	Data	
BEFF0FF0		
BEFF0FEC	LR	
BEFF0FE8	R0 (3)	← SP
BEFF0FE4	LR (0x8520)	
BEFF0FE0	R0 (2)	← SP
BEFF0FDC	LR (0x8520)	
BEFF0FD8	R0 (1)	← SP

After call

Address	Data		
BEFF0FF0		← SP	R0 = 6
BEFF0FEC	LR		
BEFF0FE8	R0 (3)	← SP	n = 3 R0 = 3 x 2
BEFF0FE4	LR (0x8520)		
BEFF0FE0	R0 (2)	← SP	n = 2 R0 = 2 x 1
BEFF0FDC	LR (0x8520)		
BEFF0FD8	R0 (1)	← SP	R0 = 1

Function Call Summary

- **Caller**
  - Puts arguments in `R0-R3`
  - Saves any needed registers (`LR`, maybe `R0-R3, R8-R12`)
  - Calls function: `BL CALLEE`
  - Restores registers
  - Looks for result in `R0`
- **Callee**
  - Saves registers that might be disturbed `(R4-R7)`
  - Performs function
  - Puts result in `R0`
  - Restores registers
  - Returns: `MOV PC, LR`

ELSEVIER

How to Encode Instructions?

How to Encode Instructions?

- **Design Principle 1: Regularity supports design simplicity**
  – 32-bit data, 32-bit instructions
  – For design simplicity, would prefer a single instruction format but…

How to Encode Instructions?

- **Design Principle 1: Regularity supports design simplicity**
  - 32-bit data, 32-bit instructions
  - For design simplicity, would prefer a single instruction format but…
  - Instructions have different needs

ELSEVIER

Design Principle 4

Good design demands good compromises

- Multiple instruction formats allow flexibility
  - `ADD`, `SUB`: use 3 register operands
  - `LDR`, `STR`: use 2 register operands and a constant
- Number of instruction formats kept small
  - to adhere to design principles 1 and 3 (regularity supports design simplicity and smaller is faster)

ELSEVIER

Machine Language

- **Binary representation of instructions**
- Computers only understand **1's and 0's**
- **32-bit instructions**
  - Simplicity favors regularity: 32-bit data & instructions
- **3 instruction formats:**
  - Data-processing
  - Memory
  - Branch

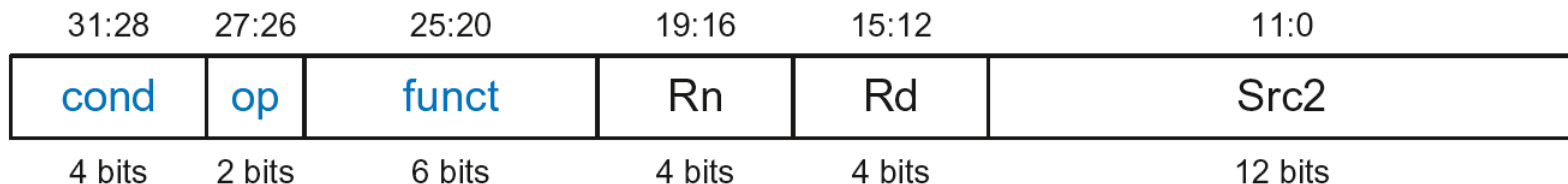
Instruction Formats

- **Data-processing**
- Memory
- Branch

ELSEVIER

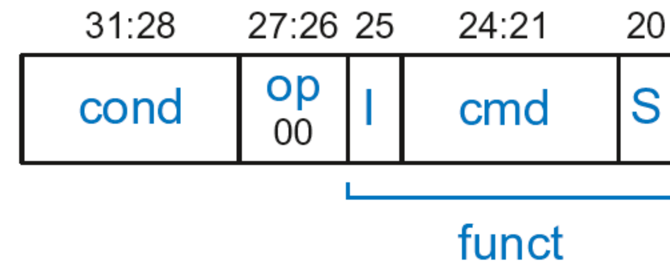
Data-processing Instruction Format

- **Operands:**
  - ***Rn*:** first source register
  - ***Src2*:** second source – register or immediate
  - ***Rd*:** destination register
- **Control fields:**
  - ***cond*:** specifies conditional execution
  - ***op*:** the *operation code* or *opcode*
  - ***funct*:** the *function*/operation to perform



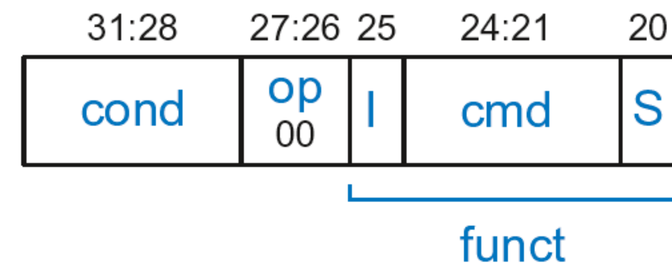
Data-processing Control Fields

- ***op*** **= 00_2** for data-processing (DP) instructions
- ***funct*** is composed of ***cmd***, ***I***-bit, and ***S***-bit



Data-processing Control Fields

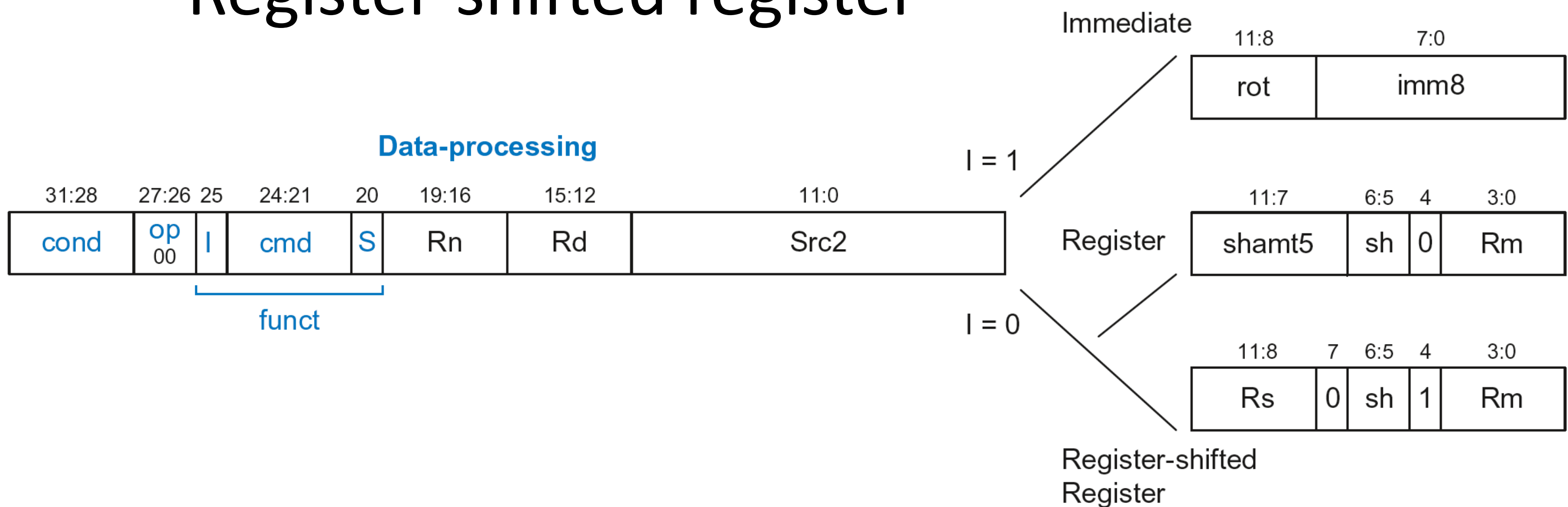
- ***op*** **= 00_2** for data-processing (DP) instructions
- ***funct*** is composed of ***cmd***, ***I***-bit, and ***S***-bit
  - ***cmd:*** specifies the specific data-processing instruction. For example,
    - ***cmd*** = 0100_2 for `ADD`
    - ***cmd*** = 0010_2 for `SUB`
  - ***I***-bit
    - ***I*** = 0: *Src2* is a register
    - ***I*** = 1: *Src2* is an immediate
  - ***S***-bit: 1 if sets condition flags
    - ***S*** = 0: `SUB R0, R5, R7`
    - ***S*** = 1: `ADDS R8, R2, R4` or `CMP R3, #10`



ELSEVIER

Data-processing *Src2* Variations

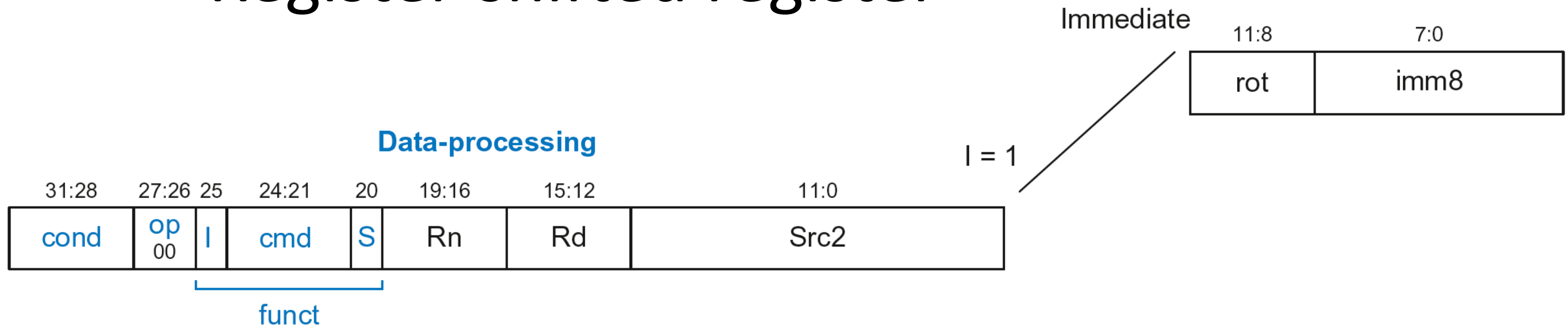
- *Src2* can be:
  - Immediate
  - Register
  - Register-shifted register



ELSEVIER

Data-processing *Src2* Variations

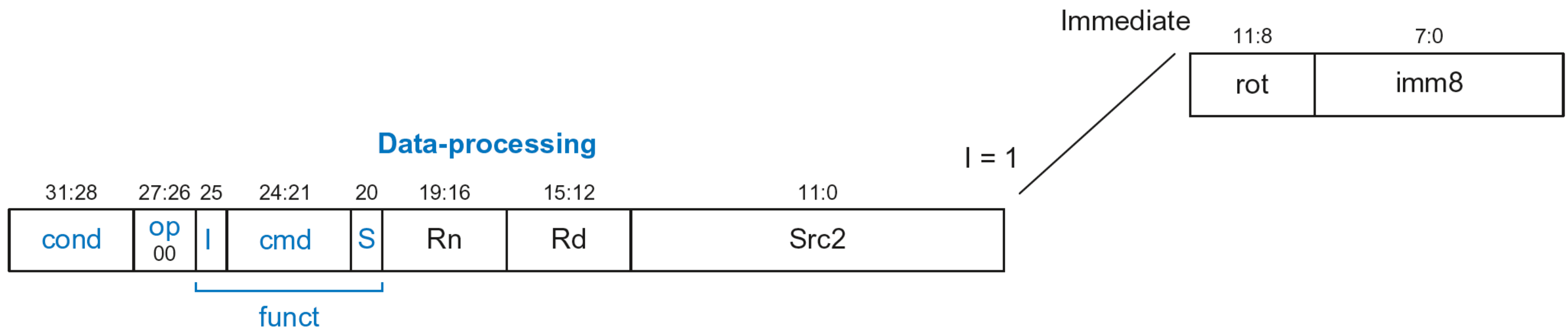
- *Src2* can be:
  - **Immediate**
  - Register
  - Register-shifted register



ELSEVIER

Immediate *Src2*

- **Immediate encoded as:**
  - *imm8*: 8-bit unsigned immediate
  - *rot*: 4-bit rotation value
- **32-bit constant is:** *imm8* **ROR** (*rot* × 2)



ELSEVIER

Immediate *Src2*

- **Immediate encoded as:**
  - *imm8*: 8-bit unsigned immediate
  - *rot*: 4-bit rotation value
- **32-bit constant is:** *imm8* **ROR** (*rot* × 2)
- **Example:** *imm8* = abcdefgh

rot	32-bit constant
0000	0000 0000 0000 0000 0000 0000 abcd efgh
0001	gh00 0000 0000 0000 0000 0000 00ab cdef
…	…
1111	0000 0000 0000 0000 0000 00ab cdef gh00

Immediate *Src2*

- **Immediate encoded as:**
  - *imm8*: 8-bit unsigned immediate
  - *rot*: 4-bit rotation value

ROR by X = ROL by (32-X)
Ex: ROR by 30 = ROL by 2

- **32-bit constant is:** *imm8* **ROR** (*rot* × 2)
- **Example:** *imm8* = abcdefgh

rot	32-bit constant
0000	0000 0000 0000 0000 0000 0000 abcd efgh
0001	gh00 0000 0000 0000 0000 0000 00ab cdef
…	…
1111	0000 0000 0000 0000 0000 00ab cdef gh00

DP Instruction with Immediate *Src2*

```
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

ELSEVIER

DP Instruction with Immediate *Src2*

```
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

ELSEVIER

DP Instruction with Immediate *Src2*

```
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

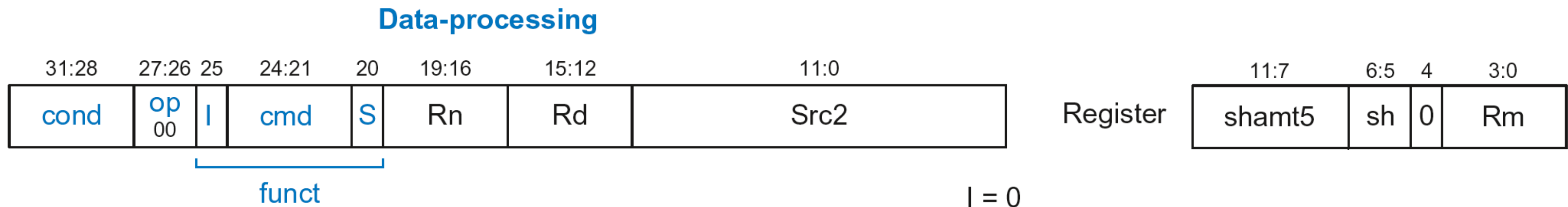
Field Values

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
1110	00	1	0010	0	0011	0010	1110	11111111

0xE2432EFF

DP Instruction with Register *Src2*

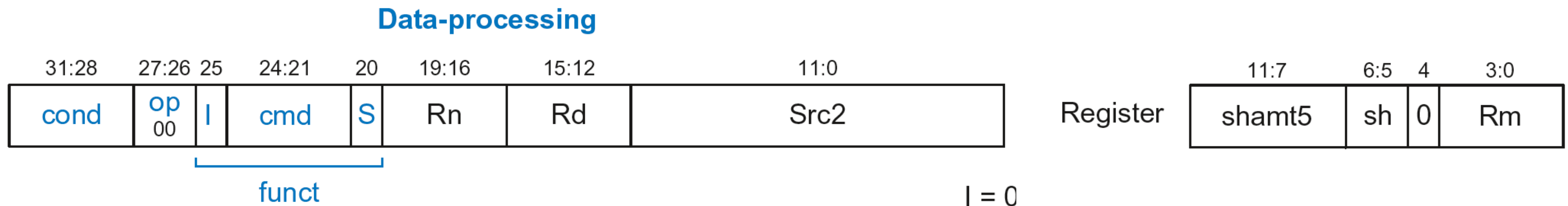
- *Src2* can be:
  - Immediate
  - **Register**
  - Register-shifted register



ELSEVIER

DP Instruction with Register *Src2*

- ***Rm***: the second source operand
- ***shamt5***: the amount Rm is shifted
- ***sh***: the type of shift (i.e., >>, <<, >>>, ROR)

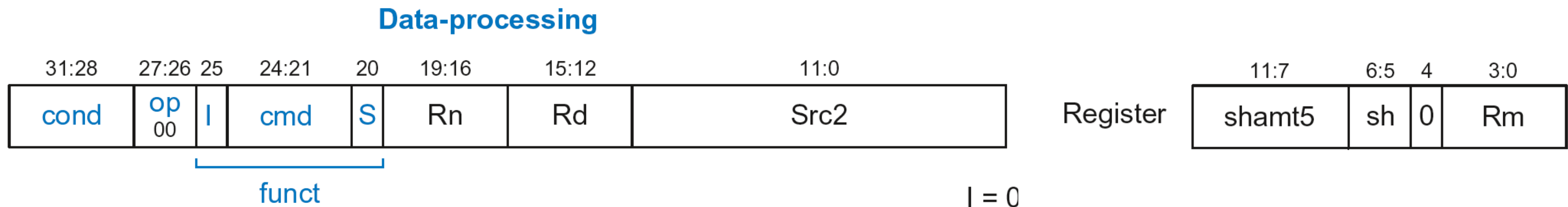


ELSEVIER

DP Instruction with Register *Src2*

- ***Rm***: the second source operand
- ***shamt5***: the amount Rm is shifted
- ***sh***: the type of shift (i.e., >>, <<, >>>, ROR)

First, consider unshifted versions of *Rm* (*shamt5*=0, *sh*=0)



ELSEVIER

DP Instruction with Register *Src2*

```
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

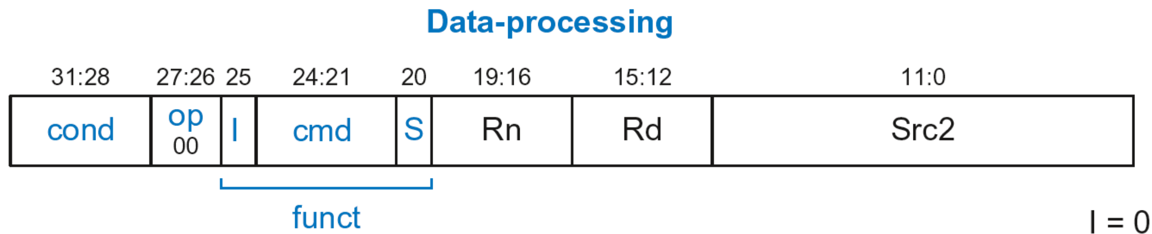
0xE0865007

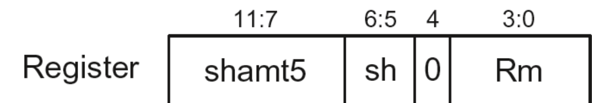
DP Instruction with Register *Src2*

- ***Rm***: the second source operand
- ***shamt5***: the amount Rm is shifted
- ***sh***: the type of shift

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

Now, consider shifted versions.



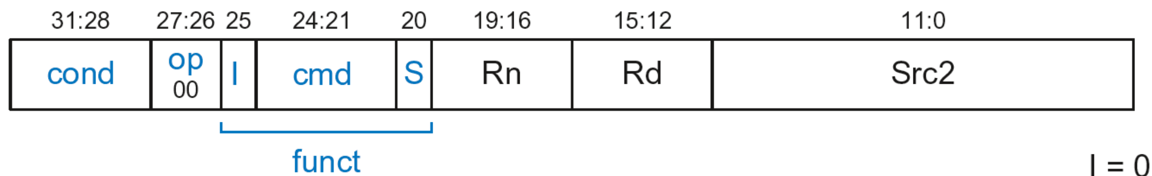


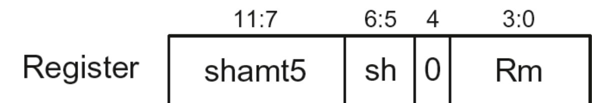
ELSEVIER

DP Instruction with Register *Src2*

```
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)



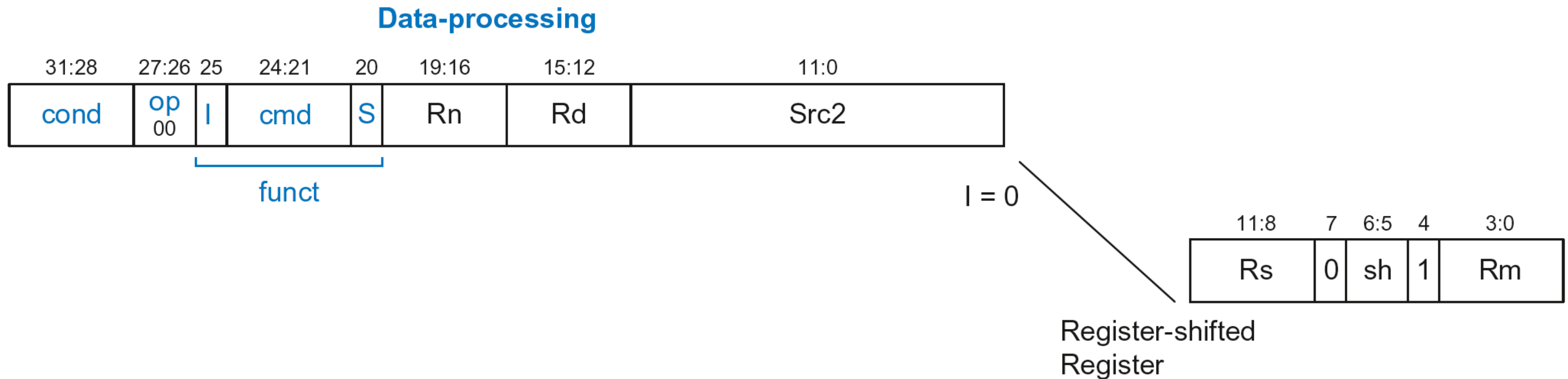


1110 00 0 1100 0 0101 1001 00010 01 0 0011

0xE1859123

DP with Register-shifted Reg. *Src2*

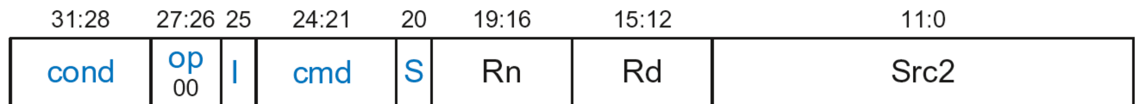
- *Src2* can be:
  - Immediate
  - Register
  - **Register-shifted register**



DP with Register-shifted Reg. *Src2*

```
EOR R8, R9, R10, ROR R12
```

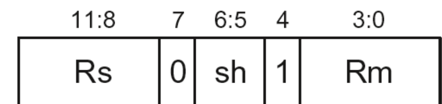
- **Operation:** R8 = R9 XOR (R10 ROR R12)
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 0001_2 (1) for `EOR`
- ***Src2*** is a register so ***I***=0
- ***Rd*** = 8, ***Rn*** = 9, ***Rm*** = 10, ***Rs*** = 12
- ***sh*** = 11_2 (ROR)



I = 0

1110 00 0 0001 0 1001 1000 1100 0 11 1 1010

0xE0298C7A



Register-shifted Register

Shift Instructions Encoding

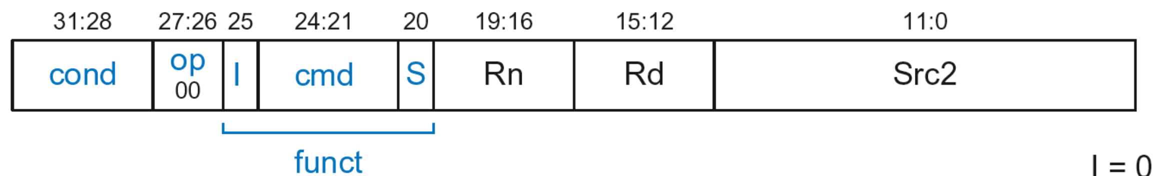
Shift Type	*sh*
LSL	00_2
LSR	01_2
ASR	10_2
ROR	11_2

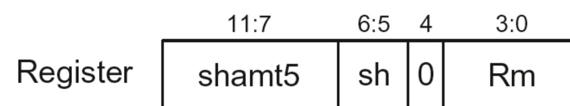
ELSEVIER

Shift Instructions: Immediate shamt

```
ROR R1, R2, #23
```

- **Operation:** R1 = R2 ROR 23
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 1101_2 (13) for all shifts (LSL, LSR, ASR, and ROR)
- ***Src2*** is an immediate-shifted register so ***I***=0
- ***Rd*** = 1, ***Rn*** = 0, ***Rm*** = 2
- ***shamt5*** = 23, ***sh*** = 11_2 (ROR)





1110 00 0 1101 0 0000 0001 10111 11 0 0010

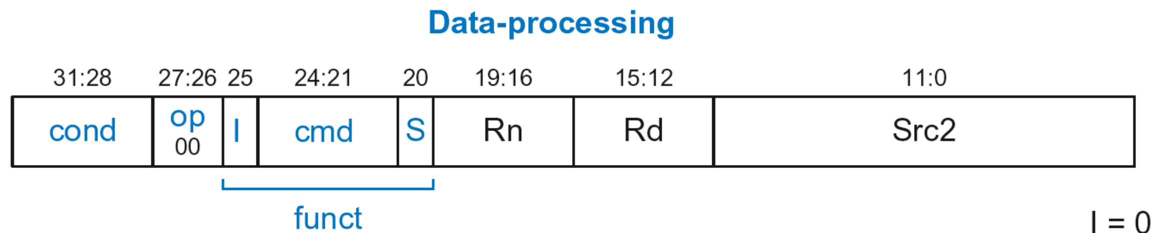
0xE1A01BE2

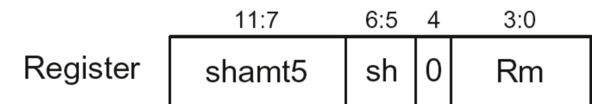
Shift Instructions: Immediate shamt

```
ROR R1, R2, #23
```

- **Operation:** R1 = R2 ROR 23
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 1101_2 (13) for all shifts (LSL, LSR, ASR, and ROR)
- ***Src2*** is an immediate-shifted register so ***I***=0
- ***Rd*** = 1, ***Rn*** = 0, ***Rm*** = 2
- ***shamt5*** = 23, ***sh*** = 11_2 (ROR)

Uses (immediate-shifted) register *Src2* encoding





1110 00 0 1101 0 0000 0001 10111 11 0 0010

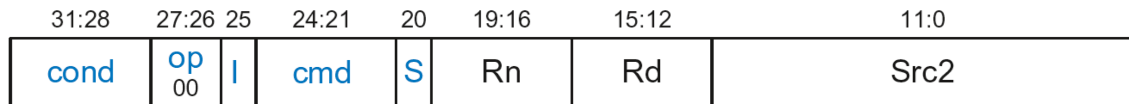
0xE1A01BE2

Shift Instructions: Register shamt

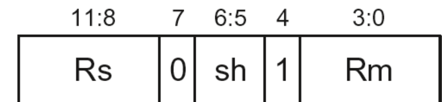
```
ASR R5, R6, R10
```

- **Operation:** R5 = R6 >>> $R10_{7:0}$
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 1101_2 (13) for all shifts (LSL, LSR, ASR, and ROR)
- ***Src2*** is a register so ***I***=0
- ***Rd*** = 5, ***Rn*** = 0, ***Rm*** = 6, ***Rs*** = 10
- ***sh*** = 10_2 (ASR)

Data-processing



I = 0



Register-shifted Register

1110 00 0 1101 0 0000 0101 1010 0 10 1 0110

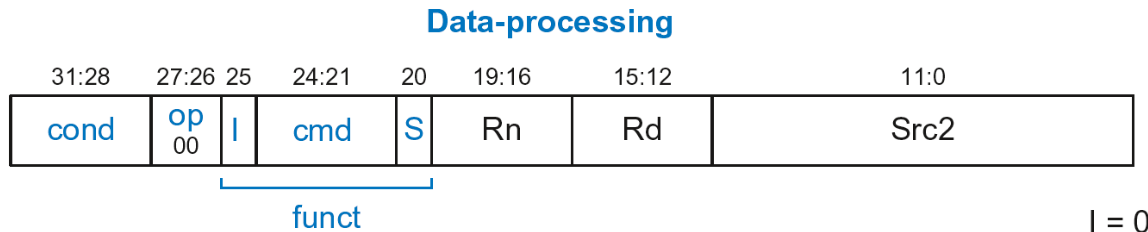
0xE1A05A56

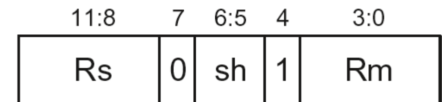
Shift Instructions: Register shamt

```
ASR R5, R6, R10
```

- **Operation:** R5 = R6 >>> $R10_{7:0}$
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 00_2 (0) for data-processing instructions
- ***cmd*** = 1101_2 (13) for all shifts (LSL, LSR, ASR, and ROR)
- ***Src2*** is a register so ***I***=0
- ***Rd*** = 5, ***Rn*** = 0, ***Rm*** = 6, ***Rs*** = 10
- ***sh*** = 10_2 (ASR)

Uses register-shifted register *Src2* encoding





Register-shifted Register

1110 00 0 1101 0 0000 0101 1010 0 10 1 0110

0xE1A05A56

Review: Data-processing Format

- *Src2* can be:
  - Immediate
  - Register
  - Register-shifted register

Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2

funct (bits 25:20: I, cmd, S)

I = 1

Immediate

11:8	7:0
rot	imm8

I = 0

Register

11:7	6:5	4	3:0
shamt5	sh	0	Rm

Register-shifted Register

11:8	7	6:5	4	3:0
Rs	0	sh	1	Rm

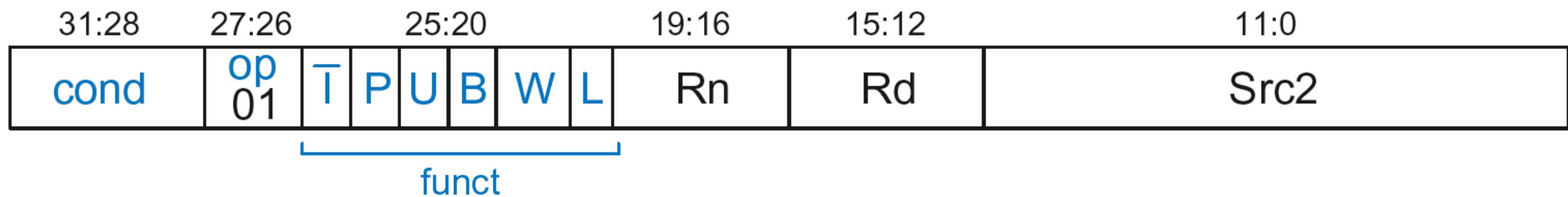
Instruction Formats

- Data-processing
- **Memory**
- Branch

Memory Instruction Format

Encodes: `LDR`, `STR`, `LDRB`, `STRB`

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



Offset Options

Recall: Address = Base Address + Offset

Example: `LDR R1, [R2, #4]`

Base Address = R2, Offset = 4

Address = (R2 + 4)

- Base address always in a register
- The offset can be:
  - an immediate
  - a register
  - or a scaled (shifted) register

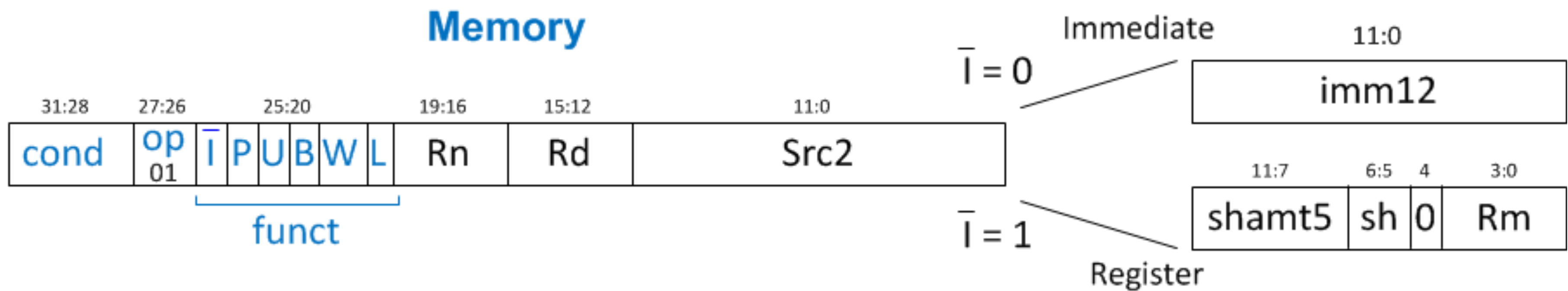
Offset Examples

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

Memory Instruction Format

Encodes: `LDR`, `STR`, `LDRB`, `STRB`

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



ELSEVIER

Indexing Modes

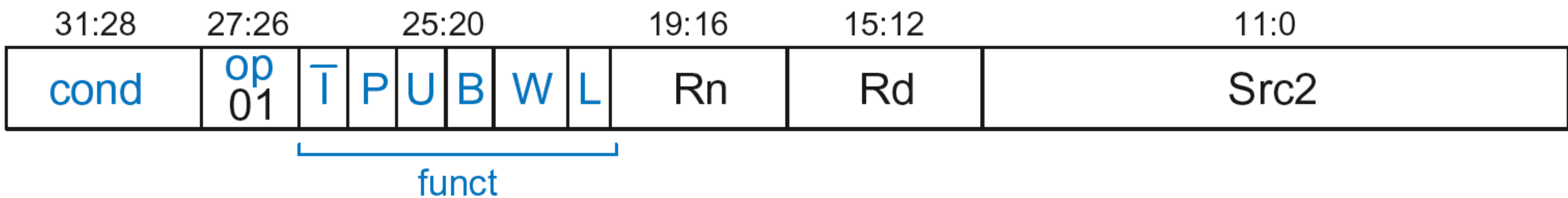
Mode	Address	Base Reg. Update
Offset	Base register ± Offset	No change
Preindex	Base register ± Offset	Base register ± Offset
Postindex	Base register	Base register ± Offset

Examples

- **Offset:**
```
LDR R1, [R2, #4]       ; R1 = mem[R2+4]
```
- **Preindex:**
```
LDR R3, [R5, #16]!     ; R3 = mem[R5+16]
                       ; R5 = R5 + 16
```
- **Postindex:**
```
LDR R8, [R1], #8       ; R8 = mem[R1]
                       ; R1 = R1 + 8
```

Memory Instruction Format

- ***funct***:
  - $\bar{I}$: Immediate bar
  - ***P***: Preindex
  - ***U***: Add
  - ***B***: Byte
  - ***W***: Writeback
  - ***L***: Load



Memory Format *funct* Encodings

Type of Operation

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

ELSEVIER

Memory Format *funct* Encodings

Type of Operation

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

Indexing Mode

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

Memory Format *funct* Encodings

Type of Operation

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

Indexing Mode

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

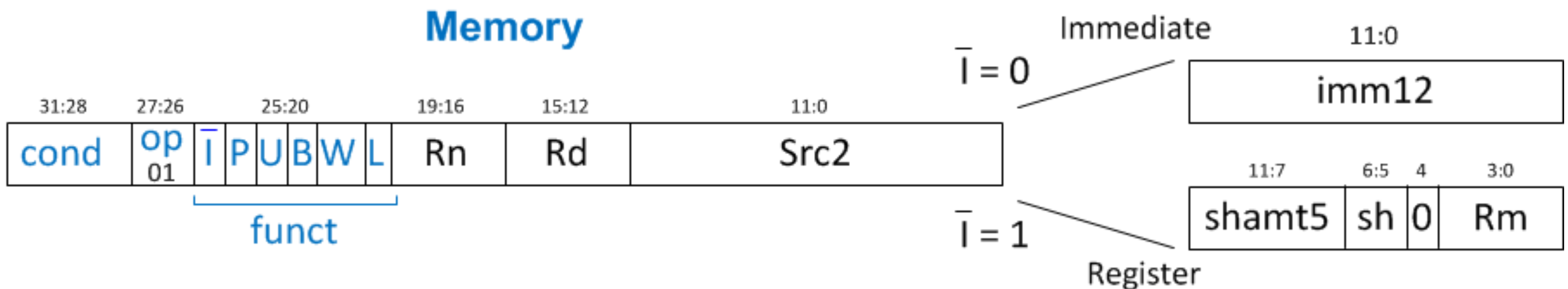
Add/Subtract Immediate/Register Offset

Value	$\bar{I}$	*U*
0	**Immediate** offset in *Src2*	**Subtract** offset from base
1	**Register** offset in *Src2*	**Add** offset to base

Memory Instruction Format

Encodes: `LDR`, `STR`, `LDRB`, `STRB`

- ***op*** = 01_2
- ***Rn*** = base register
- ***Rd*** = destination (load), source (store)
- ***Src2*** = offset: immediate or register (optionally shifted)
- ***funct*** = $\overline{I}$ (immediate bar), *P* (preindex), *U* (add), *B* (byte), *W* (writeback), *L* (load)



Memory Instr. with Immediate *Src2*

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

- **Operation:** mem[R5] <= R11; R5 = R5 - 26
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 01_2 (1) for memory instruction
- ***funct*** = 0000000_2 (0)
  $\bar{I}$ = 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

31:28	27:26	25:20	19:16	15:12	11:0
1110_2	01_2	0000000_2	5	11	26
cond	op	$\bar{I}$PUBWL	Rn	Rd	imm12
1110	01	000000	0101	1011	0000 0001 1010

E 4 0 5 B 0 1 A

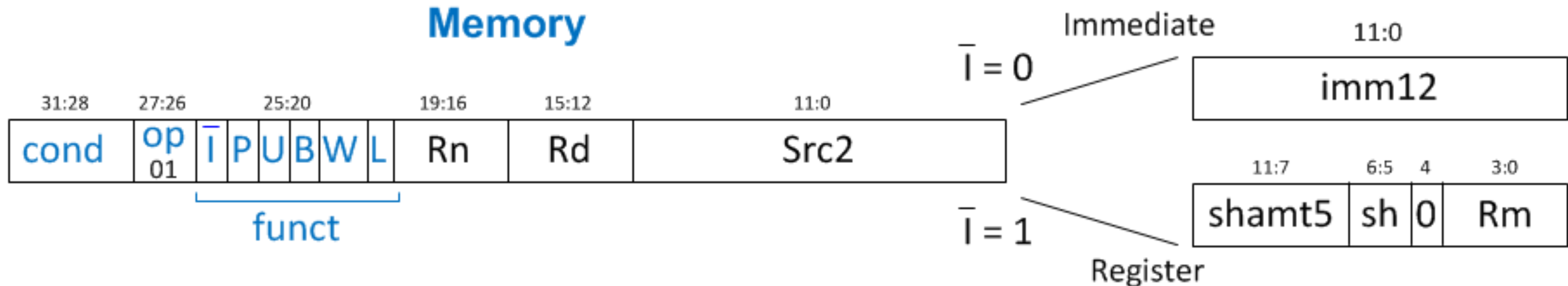
ELSEVIER

Memory Instr. with Register *Src2*

```
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)
  $\overline{I}$ = 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**

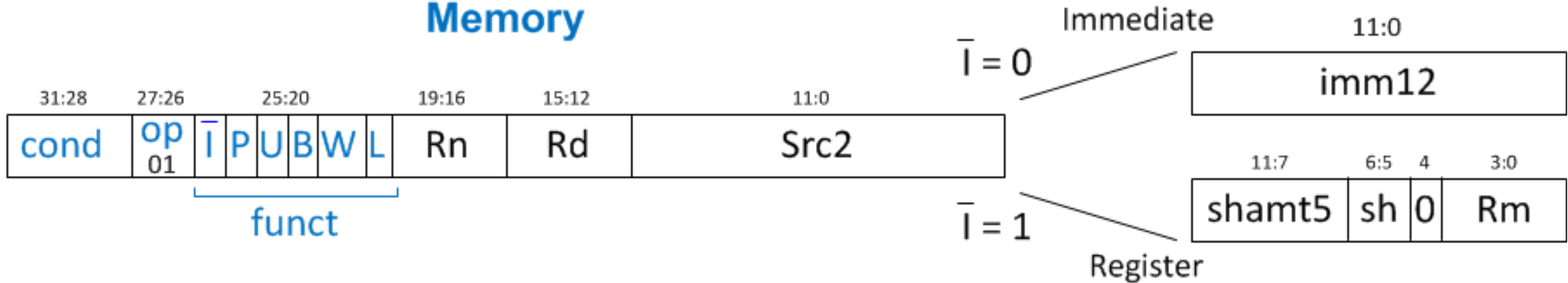


Memory Instr. with Scaled Reg. *Src2*

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

- **Operation:** mem[R1 + (R3 << 2)] <= R9
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 01_2 (1) for memory instruction
- ***funct*** = 111000_2 (0)
  $\bar{I}$ = 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**

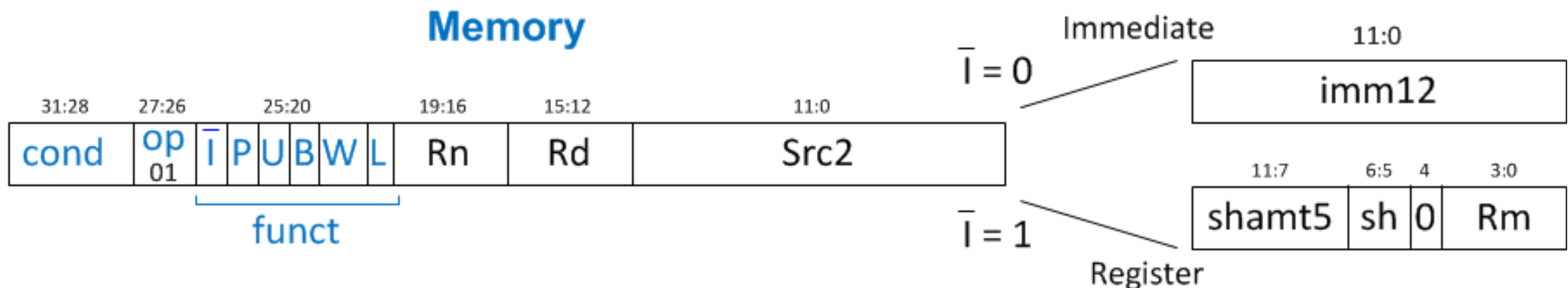


ELSEVIER

Review: Memory Instruction Format

Encodes: `LDR`, `STR`, `LDRB`, `STRB`

- ***op*** = 01_2
- ***Rn*** = base register
- ***Rd*** = destination (load), source (store)
- ***Src2*** = offset: register (optionally shifted) or immediate
- ***funct*** = $\bar{I}$ (immediate bar), *P* (preindex), *U* (add), *B* (byte), *W* (writeback), *L* (load)



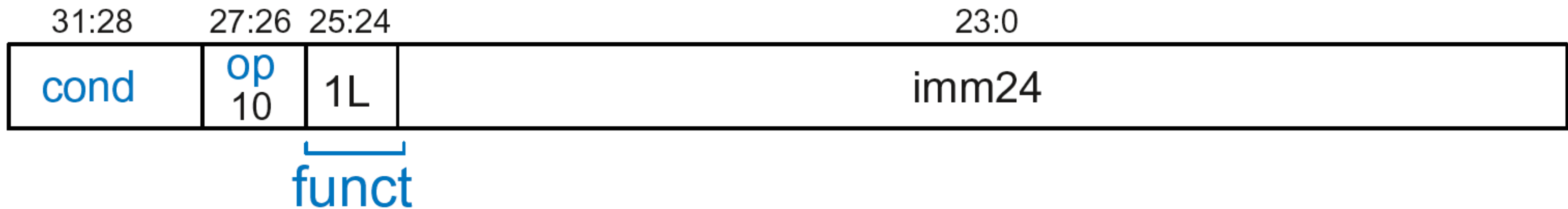
Instruction Formats

- Data-processing
- Memory
- **Branch**

Branch Instruction Format

Encodes `B` and `BL`

- ***op*** = 10_2
- ***imm24***: 24-bit immediate
- ***funct*** = $1L_2$: ***L*** = 1 for `BL`, ***L*** = 0 for `B`



Encoding Branch Target Address

- ***Branch Target Address* (BTA)*:*** Next PC when branch taken
- BTA is relative to current PC + 8
- *imm24* encodes BTA
- ***imm24*** = # of words BTA is away from PC+8

ELSEVIER

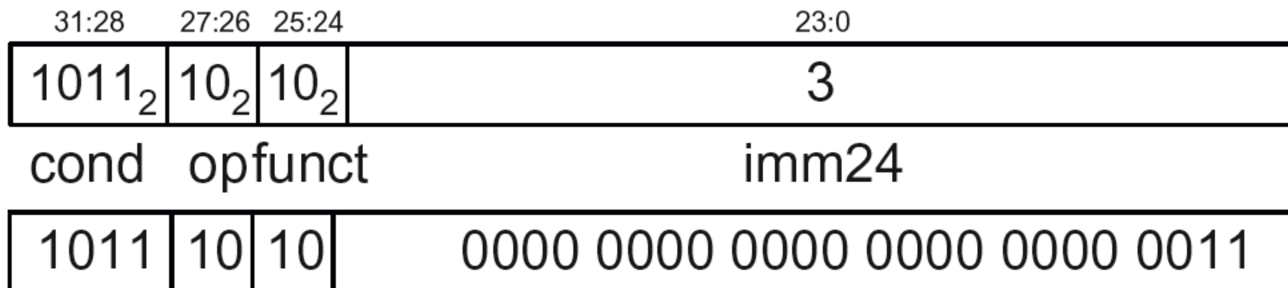
Branch Instruction: Example 1

ARM assembly code

```
0xA0          BLT  THERE          ← PC
0xA4          ADD  R0, R1, R2
0xA8          SUB  R0, R0, R9     ← PC+8
0xAC          ADD  SP, SP, #8
0xB0          MOV  PC, LR
0xB4   THERE  SUB  R0, R0, #1     ← BTA
0xB8          BL   TEST
```

- PC = 0xA0
- PC + 8 = 0xA8
- THERE label is 3 instructions past PC+8
- So, *imm24* = 3

Field Values



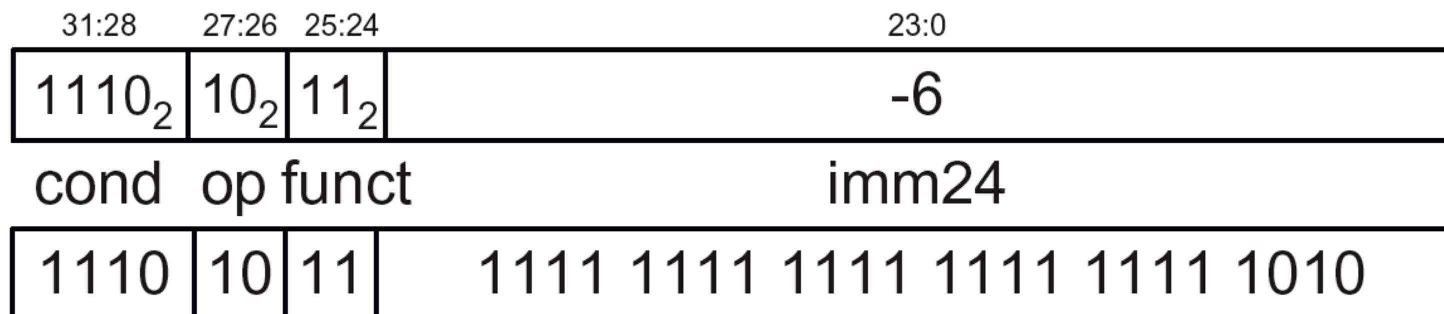
0xBA000003

Branch Instruction: Example 2

ARM assembly code

```
0x8040 TEST   LDRB R5, [R0, R3]   ← BTA
0x8044        STRB R5, [R1, R3]
0x8048        ADD  R3, R3, #1
0x8044        MOV  PC, LR
0x8050        BL   TEST           ← PC
0x8054        LDR  R3, [R1], #4
0x8058        SUB  R4, R3, #9     ← PC+8
```

- PC = 0x8050
- PC + 8 = 0x8058
- `TEST` label is 6 instructions before PC+8
- So, *imm24* = -6



0xEBFFFFFA

ELSEVIER

Review: Instruction Formats

Data-processing

31:28	27:26	25	24:21	20	19:16	15:12	11:0
cond	op 00	I	cmd	S	Rn	Rd	Src2

funct = bits 25:20 (I, cmd, S)

Src2:

I = 1 — Immediate

11:8	7:0
rot	imm8

I = 0 — Register

11:7	6:5	4	3:0
shamt5	sh	0	Rm

I = 0 — Register-shifted Register

11:8	7	6:5	4	3:0
Rs	0	sh	1	Rm

Memory

31:28	27:26	25:20						19:16	15:12	11:0
cond	op 01	$\bar{I}$	P	U	B	W	L	Rn	Rd	Src2

funct = bits 25:20 ($\bar{I}$, P, U, B, W, L)

Src2:

$\bar{I} = 0$ — Immediate

11:0
imm12

$\bar{I} = 1$ — Register

11:7	6:5	4	3:0
shamt5	sh	0	Rm

Branch

31:28	27:26	25:24	23:0
cond	op 10	1L	imm24

funct = bits 25:24 (1L)

ELSEVIER

Conditional Execution

Encode in *cond* bits of machine instruction

For example,

```
ANDEQ R1, R2, R3       (cond = 0000)
ORRMI R4, R5, #0xF     (cond = 0100)
SUBLT R9, R3, R8       (cond = 1011)
```

Review: Condition Mnemonics

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\overline{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\overline{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\overline{N}$
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	$\overline{V}$
1000	HI	Unsigned higher	$\overline{Z}C$
1001	LS	Unsigned lower or same	Z OR $\overline{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	Z OR $(N \oplus V)$
1110	AL (or none)	Always / unconditional	Ignored

Conditional Execution: Machine Code

Assembly Code

```
SUBS  R1, R2, R3
ADDEQ R4, R5, R6
ANDHS R7, R5, R6
ORRMI R8, R5, R6
EORLT R9, R5, R6
```

Field Values

Assembly	cond (31:28)	op (27:26)	I (25)	cmd (24:21)	S (20)	rn (19:16)	rd (15:12)	shamt5 (11:7)	sh (6:5)	4	rm (3:0)
SUBS R1, R2, R3	14	0	0	2	1	2	1	0	0	0	3
ADDEQ R4, R5, R6	0	0	0	4	0	5	4	0	0	0	6
ANDHS R7, R5, R6	2	0	0	0	0	5	7	0	0	0	6
ORRMI R8, R5, R6	4	0	0	12	0	5	8	0	0	0	6
EORLT R9, R5, R6	11	0	0	1	0	5	9	0	0	0	6

Machine Code

cond (31:28)	op (27:26)	I (25)	cmd (24:21)	S (20)	rn (19:16)	rd (15:12)	shamt5 (11:7)	sh (6:5)	4	rm (3:0)	
1110	00	0	0010	0	0010	0001	00000	00	0	0011	(0xE0421003)
0000	00	0	0100	0	0101	0100	00000	00	0	0110	(0x00854006)
0010	00	0	0000	0	0101	0111	00000	00	0	0110	(0x20057006)
0100	00	0	1100	0	0101	1000	00000	00	0	0110	(0x41858006)
1011	00	0	0001	0	0101	1001	00000	00	0	0110	(0xB0259006)

ELSEVIER

Interpreting Machine Code

- **Start with *op*:** tells how to parse rest

  op = 00 (Data-processing)

  op = 01 (Memory)

  op = 10 (Branch)

- ***I*-bit:** tells how to parse ***Src2***
- **Data-processing instructions:**

  If ***I***-bit is 0, bit 4 determines if ***Src2*** is a register (bit 4 = 0) or a register-shifted register (bit 4 = 1)

- **Memory instructions:**

  Examine ***funct*** bits for indexing mode, instruction, and add or subtract offset

ELSEVIER

Interpreting Machine Code: Example 1

0xE0475001

ELSEVIER

Interpreting Machine Code: Example 1

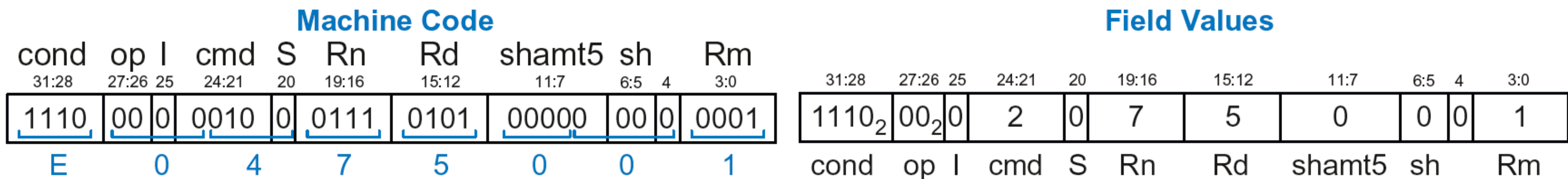
0xE0475001

- **Start with *op*:** 00_2, so data-processing instruction

Interpreting Machine Code: Example 1

0xE0475001

- **Start with *op*:** 00_2, so data-processing instruction
- ***I*-bit:** 0, so *Src2* is a register
- **bit 4:** 0, so *Src2* is a register (optionally shifted by *shamt5*)

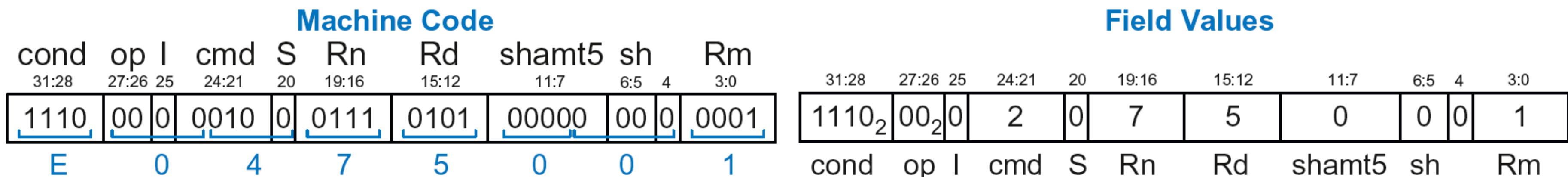


ELSEVIER

Interpreting Machine Code: Example 1

0xE0475001

- **Start with *op*:** 00_2, so data-processing instruction
- ***I*-bit:** 0, so *Src2* is a register
- **bit 4:** 0, so *Src2* is a register (optionally shifted by *shamt5*)
- ***cmd*:** 0010_2 (2), so SUB
- **Rn**=7, **Rd**=5, **Rm**=1, ***shamt5*** = 0, ***sh*** = 0
- So, instruction is: **`SUB R5,R7,R1`**



ELSEVIER

Interpreting Machine Code: Example 2

0xE5949010

Interpreting Machine Code: Example 2

0xE5949010

- **Start with *op***: 01_2, so memory instruction
- ***funct***: *B*=0, *L*=1, so `LDR`; *P*=1, *W*=0, so offset indexing; *I*=0, so immediate offset, *U*=1, so add offset
- **Rn**=4, **Rd**=9, ***imm12*** = 16
- So, instruction is: **`LDR R9,[R4,#16]`**

Machine Code

cond (31:28)	op (27:26)	$\bar{I}$PUBWL (25:20)	Rn (19:16)	Rd (15:12)	imm12 (11:0)
1110	01	011001	0100	1001	0000 0001 0000
E	5	9	4	9	0 1 0

Field Values

cond (31:28)	op (27:26)	$\bar{I}$PUBWL (25:20)	Rn (19:16)	Rd (15:12)	imm12 (11:0)
1110_2	01_2	25	4	9	16

ELSEVIER

Addressing Modes

How do we address operands?

- Register
- Immediate
- Base
- PC-Relative

ELSEVIER

Addressing Modes

How do we address operands?

- **Register Only**
- Immediate
- Base
- PC-Relative

ELSEVIER

Register Addressing

- Source and destination operands found in registers
- Used by data-processing instructions
- **Three submodes:**
  - Register-only
  - Immediate-shifted register
  - Register-shifted register

ELSEVIER

Register Addressing Examples

- **Register-only**

  Example: `ADD R0, R2, R7`

- **Immediate-shifted register**

  Example: `ORR R5, R1, R3, LSL #1`

- **Register-shifted register**

  Example: `SUB R12, R9, R0, ASR R1`

ELSEVIER

Addressing Modes

How do we address operands?

- Register Only
- **Immediate**
- Base
- PC-Relative

ELSEVIER

Immediate Addressing

- Operands found in registers **and** immediates

    Example: `ADD R9, R1, #14`

- Uses data-processing format with *I*=1
  - Immediate is encoded as
    - 8-bit immediate (*imm8*)
    - 4-bit rotation (*rot*)
  - 32-bit immediate = *imm8* ROR (*rot* x 2)

ELSEVIER

Addressing Modes

How do we address operands?

- Register Only
- Immediate
- **Base**
- PC-Relative

Base Addressing

- Address of operand is:

  base register + offset

- Offset can be a:
  - 12-bit Immediate
  - Register
  - Immediate-shifted Register

Base Addressing Examples

- **Immediate offset**

  Example: `LDR R0, [R8, #-11]`

  (R0 = mem[R8 - 11])

- **Register offset**

  Example: `LDR R1, [R7, R9]`

  (R1 = mem[R7 + R9])

- **Immediate-shifted register offset**

  Example: `STR R5, [R3, R2, LSL #4]`

  (R5 = mem[R3 + (R2 << 4)])

ELSEVIER

Addressing Modes

How do we address operands?

- Register Only
- Immediate
- Base
- **PC-Relative**

PC-Relative Addressing

- Used for branches
- Branch instruction format:
  - Operands are PC and a signed 24-bit immediate (*imm24*)
  - Changes the PC
  - New PC is relative to the old PC
  - *imm24* indicates the number of words away from PC+8
- PC = (PC+8) + (SignExtended(*imm24*) x 4)

Power of the Stored Program

- **32-bit instructions & data** stored in memory
- **Sequence of instructions:** only difference between two applications
- **To run a new program:**
  - No rewiring required
  - Simply store new program in memory
- **Program Execution:**
  - Processor *fetches* (reads) instructions from memory in sequence
  - Processor performs the specified operation

ELSEVIER

The Stored Program

Assembly Code	Machine Code
`MOV R1, #100`	`0xE3A01064`
`MOV R2, #69`	`0xE3A02045`
`ADD R3, R1, R2`	`0xE2813002`
`STR R3, [R1]`	`0xE5913000`

Stored Program

Address	Instructions
⋮	⋮
0000000C	E 5 9 1 3 0 0 0
00000008	E 2 8 1 3 0 0 2
00000004	E 3 A 0 2 0 4 5
00000000	E 3 A 0 1 0 6 4 ← PC

Main Memory

Program Counter (PC): keeps track of current instruction

Up Next

How to implement the ARM Instruction Set Architecture in Hardware

Microarchitecture