MA4832

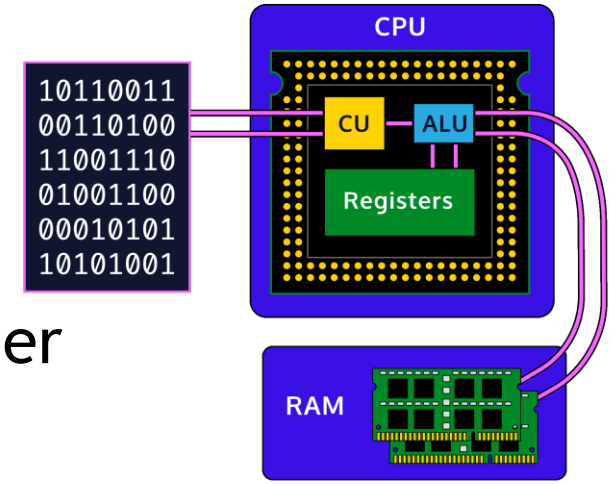
Microprocessor Systems

Xie Ming, PhD (France)

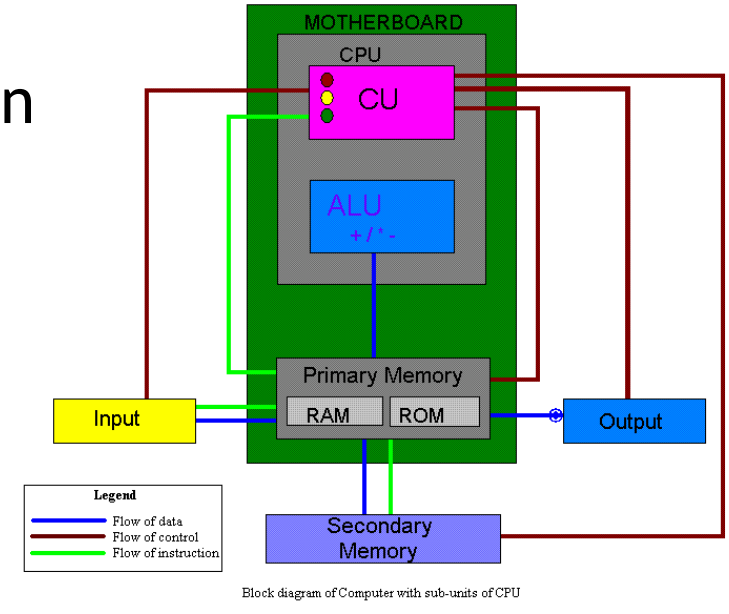
mmxie@ntu.edu.sg

http://personal.ntu.edu.sg/mmxie

Outline



- Lecture 1: Basics of ARM Microcontroller
- Lecture 2: ARM's Memories
- Lecture 3: ARM's Data Representation
- Lecture 4: ARM's Programming
- Lecture 5: ARM's Data Input/Output
- Lecture 6: ARM's Data Processing



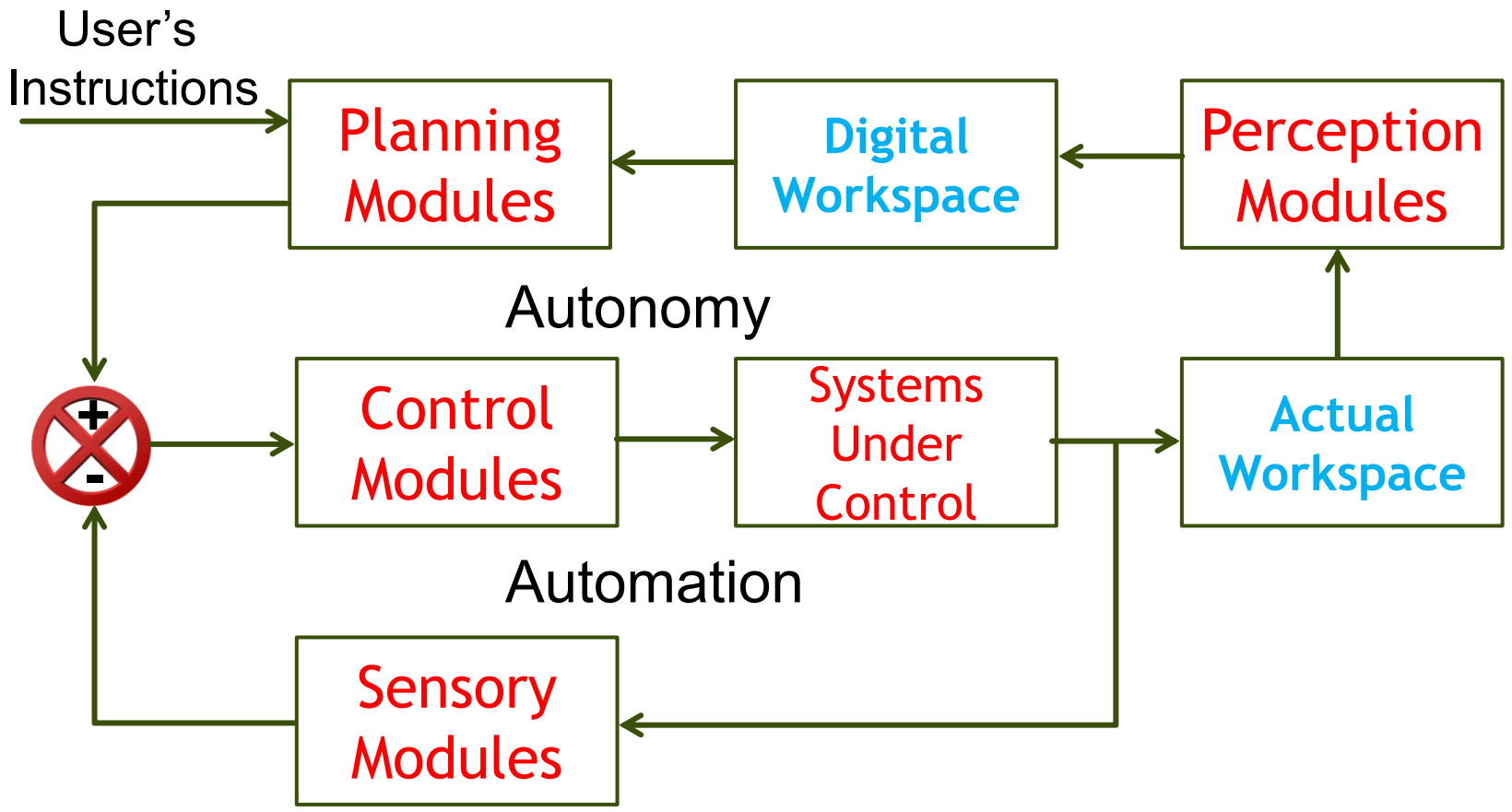
Block diagram of Computer with sub-units of CPU

Remember your mission as MAE undergraduates …

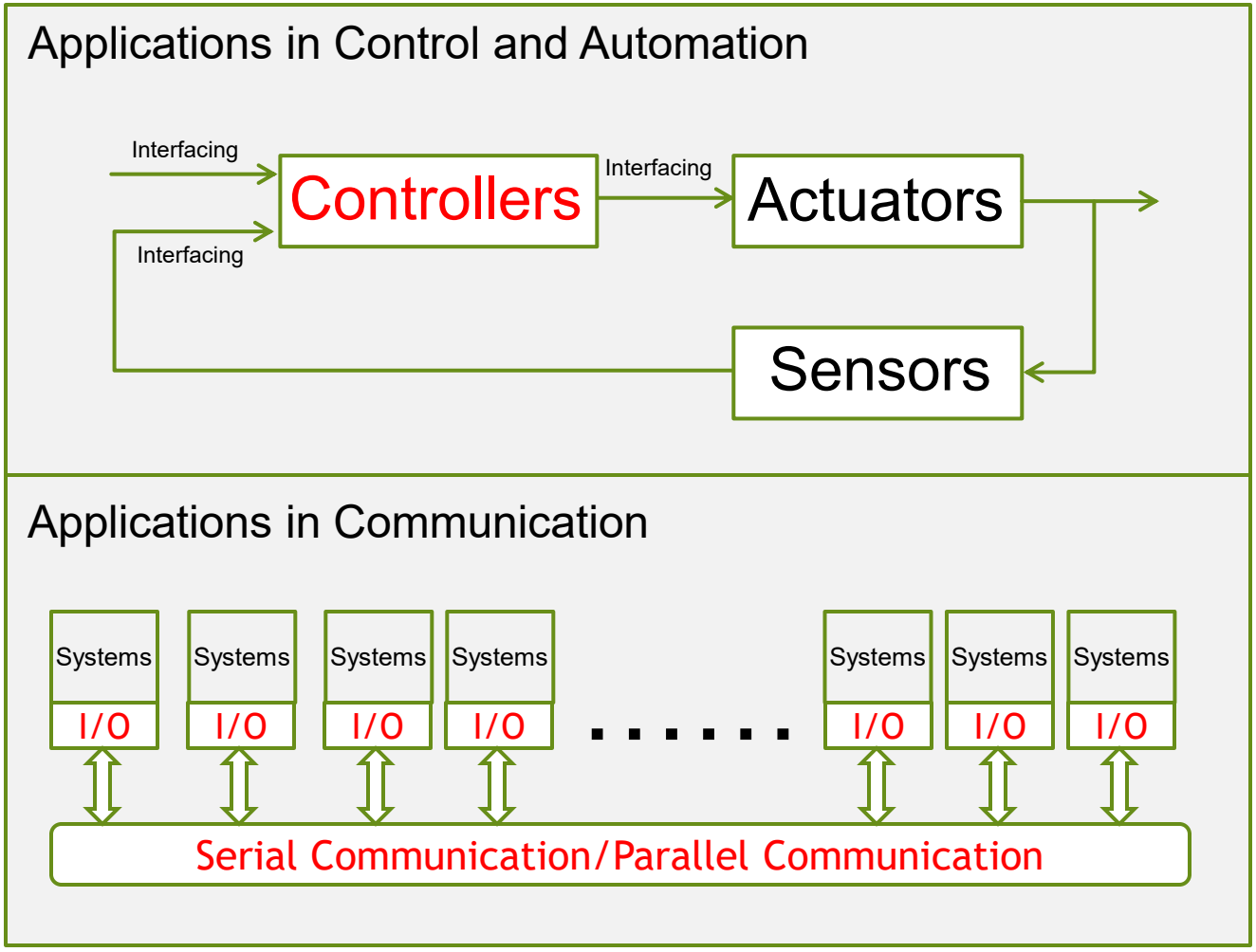
- You are here to grow your knowledge and skills so as to be able to design machines with controllable behaviors and hopefully in some intelligent ways.

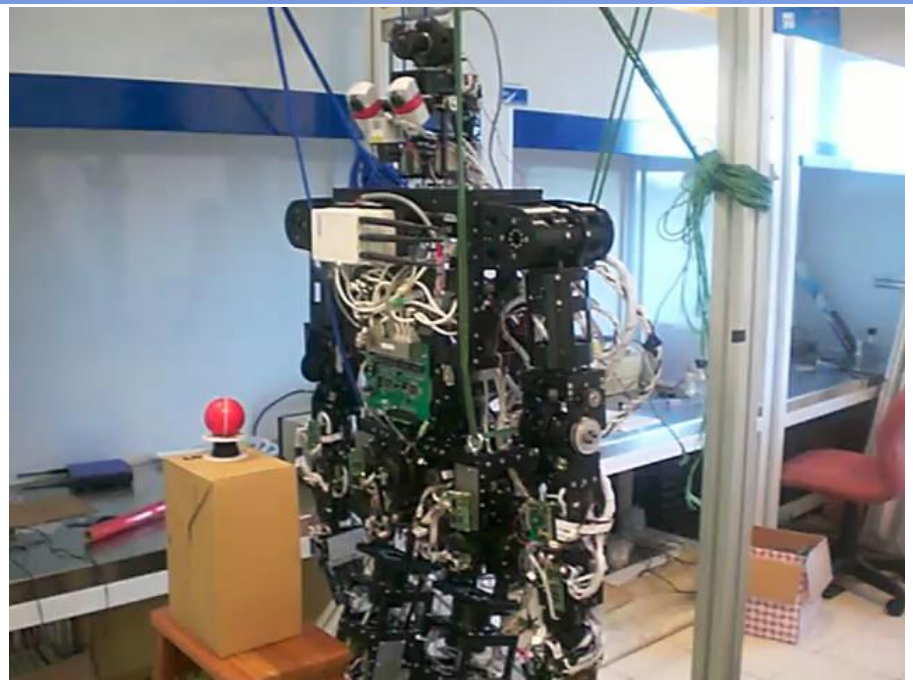
How to fulfill your mission?

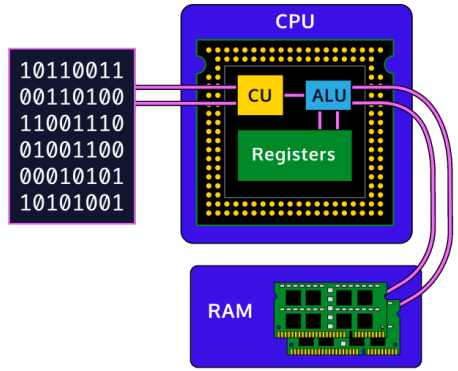
- To apply learnt knowledge and skills into the implementation of the following universal blueprint underlying all the intelligent machines or systems.

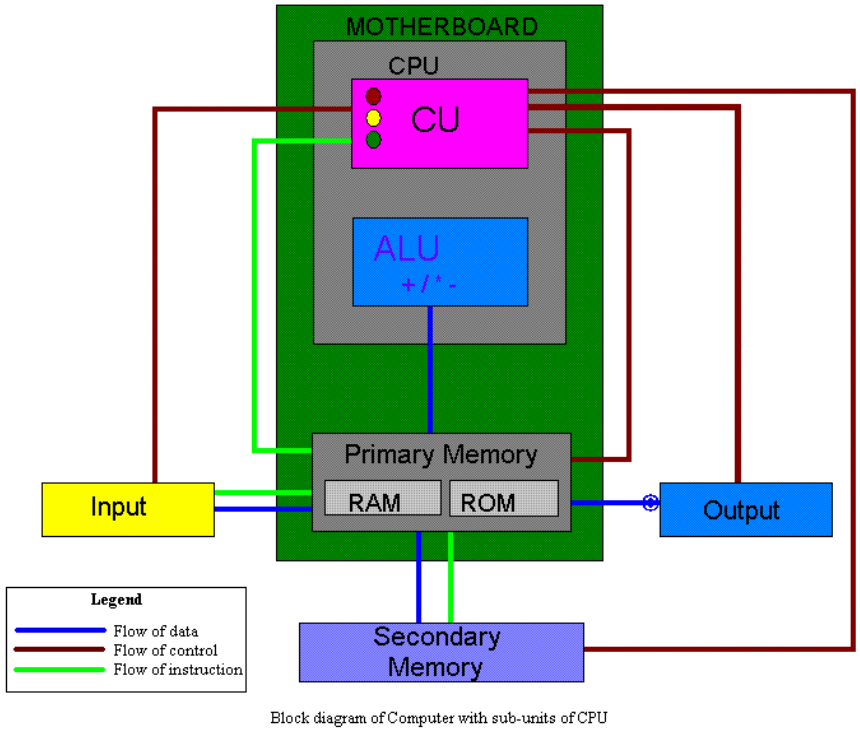


Why to study?


Applications in Control and Automation
Interfacing
Interfacing
Controllers
Interfacing
Actuators
Sensors
Applications in Communication
Systems
I/O
Systems
I/O
Systems
I/O
Systems
I/O
Systems
I/O
Systems
I/O
Systems
I/O
Serial Communication/Parallel Communication



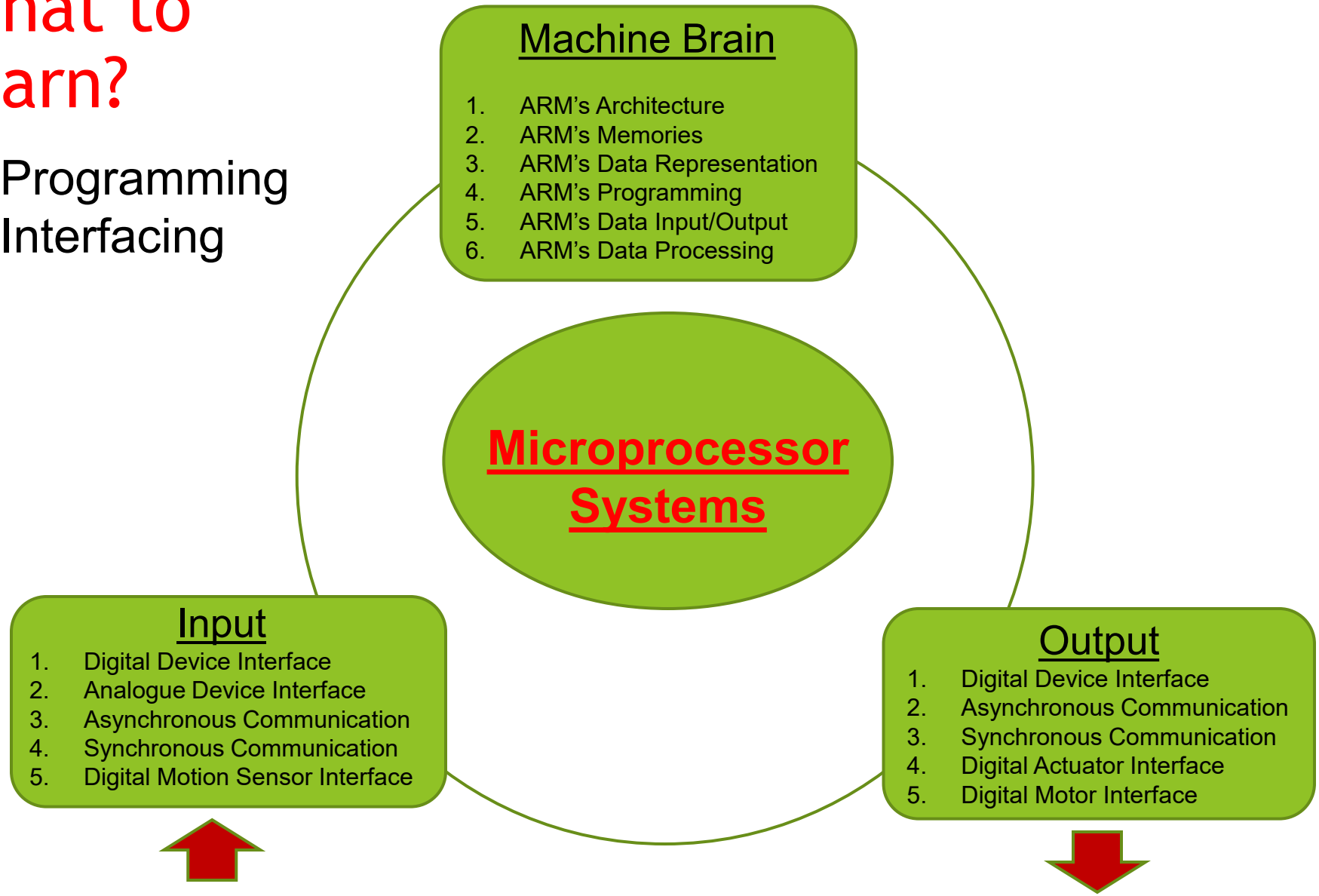




Block diagram of Computer with sub-units of CPU

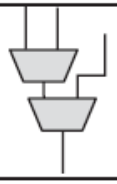
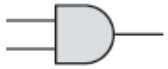
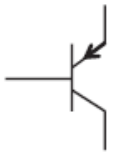
What to learn?

- Programming
- Interfacing



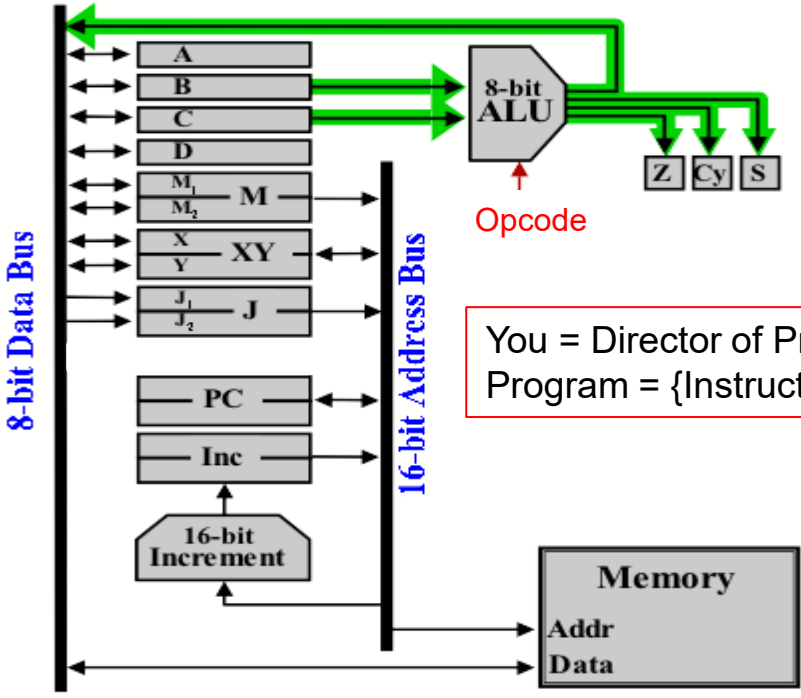
What is your role?

- Data = {Values, Symbols, Addresses, Instructions}
- Instructions = {Op Code + Addresses + Value/Symbol}

Applications/OS	
Languages	C++, Java
ISA	`EOR r3,r2,r1` `BEQ Table`
Microarchitecture	
Gates	
Transistors	

Algorithms or Solutions

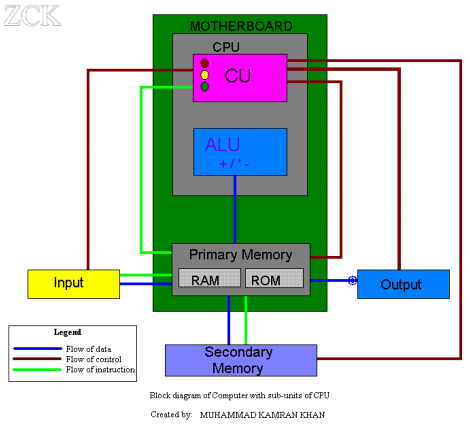
Problems to be solved



You = Director of Program
Program = {Instructions}

Memory = {Address + Data/Instruction}

How to learn?



- To understand data flows inside a microcontroller.
- To translate your solutions into data flows.
- To pay attention to <memory address> and <memory data>.
  - Memory Address: Address Label/Name and Address Value.
  - Memory Data: Data Label/Name and Data Value.

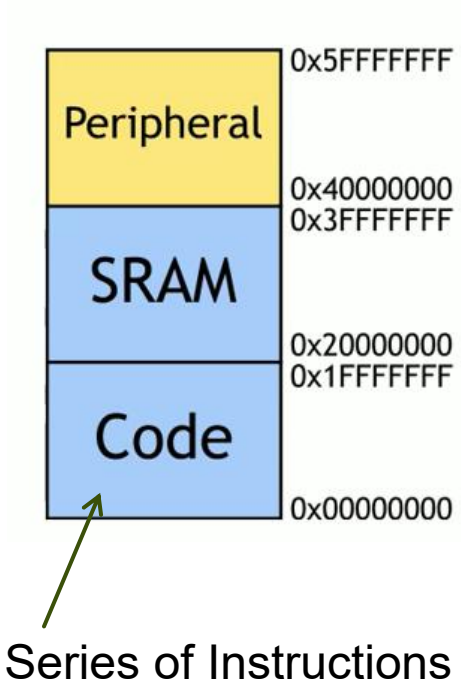
Memory Content

Data	Memory Address

- To pay attention to <memory address> and <memory code>.
  - Memory Address: Address Label/Name and Address Value.
  - Memory Code: Code Label/Name and Code Value.

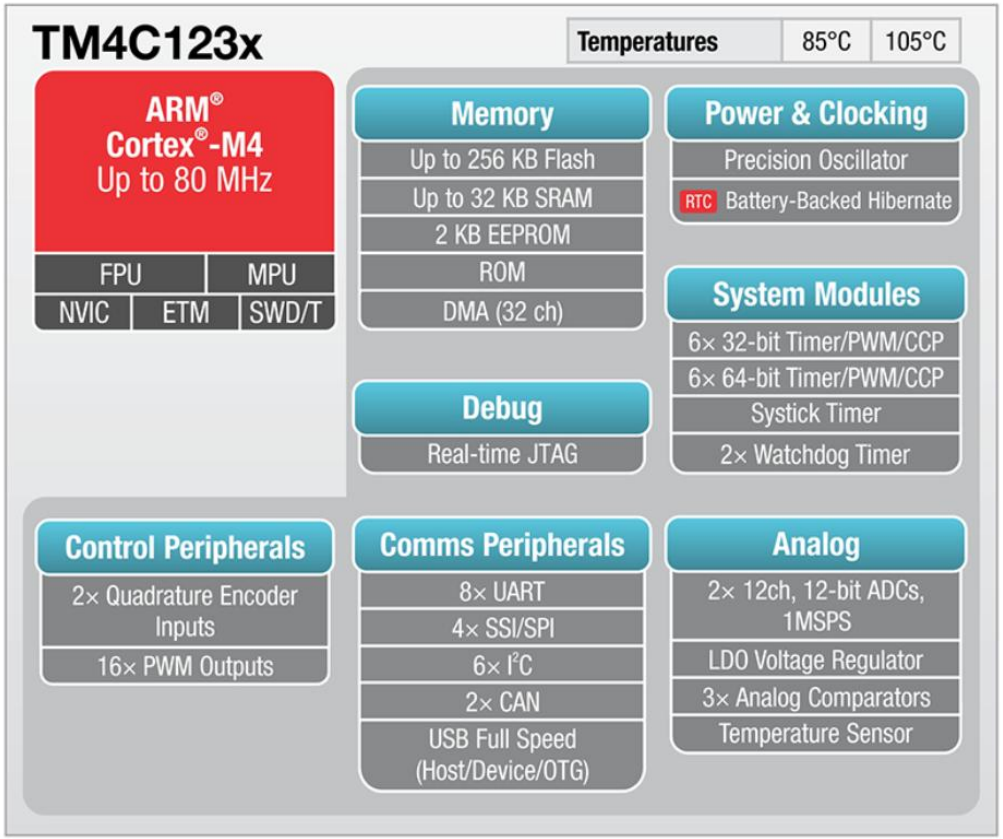
Memory Content

Instruction	Memory Address



Example of Using I/O Modules

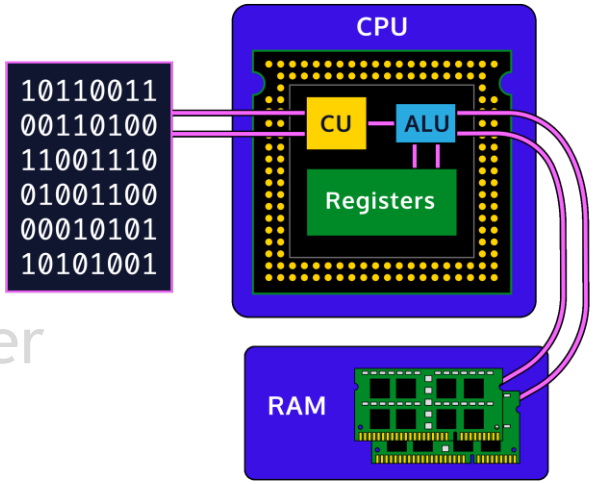
- Configure Control Registers
- Clear/Monitor Status Registers
- Read/Write Data Registers
- Instructions:
  - MOV <address of destination>, <source of value>
  - LDR <address of destination>, <source of value>
  - STR <source of value>, <address of destination>



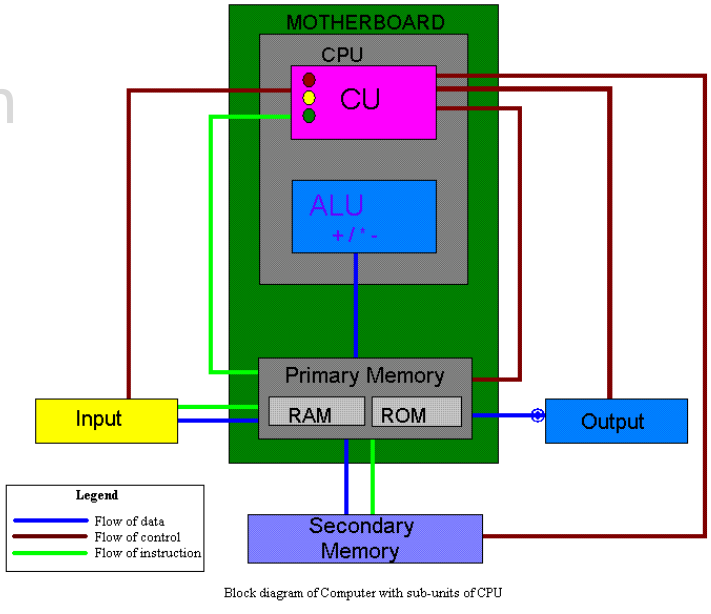

```
MOV   R0, #0x11
MOV   R1, #2560
MVN   R2, #4
MOVW  R3, #0xC0DE
MOVT  R3, #0xFEED
MOV   R4, R1
```

Word = 2 Bytes

Today's Lecture …



- Lecture 1: Basics of ARM Microcontroller
- Lecture 2: ARM's Memories
- Lecture 3: ARM's Data Representation
- Lecture 4: ARM's Programming
- Lecture 5: ARM's Data Input/Output
- Lecture 6: ARM's Data Processing



Block diagram of Computer with sub-units of CPU

NANYANG TECHNOLOGICAL UNIVERSITY

School of Mechanical & Aerospace Engineering

Design, Machine, Control and Intelligence

MA4832

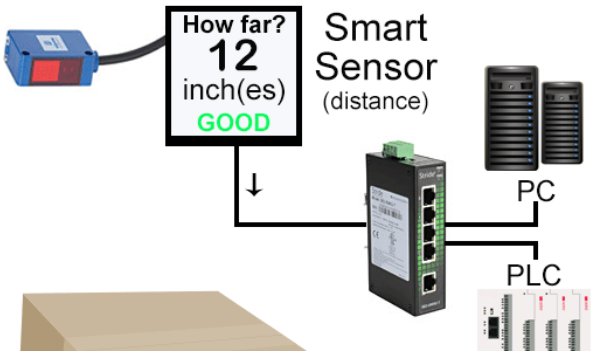
ARM's Data Processing

Xie Ming, PhD (France)

http://personal.ntu.edu.sg/mmxie

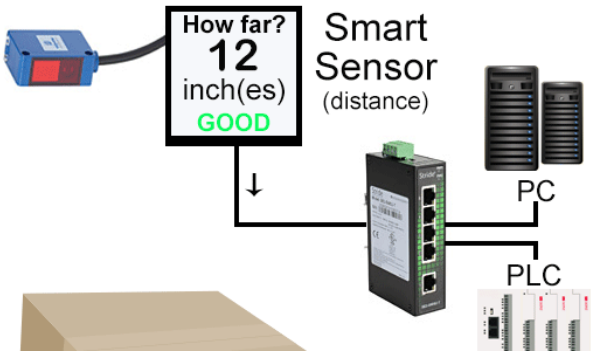
Outline

- Computations, Status and Condition Codes
- Arithmetic Operations
- Logical Operations
- Control of Computational Condition
- Control of Computational Flow



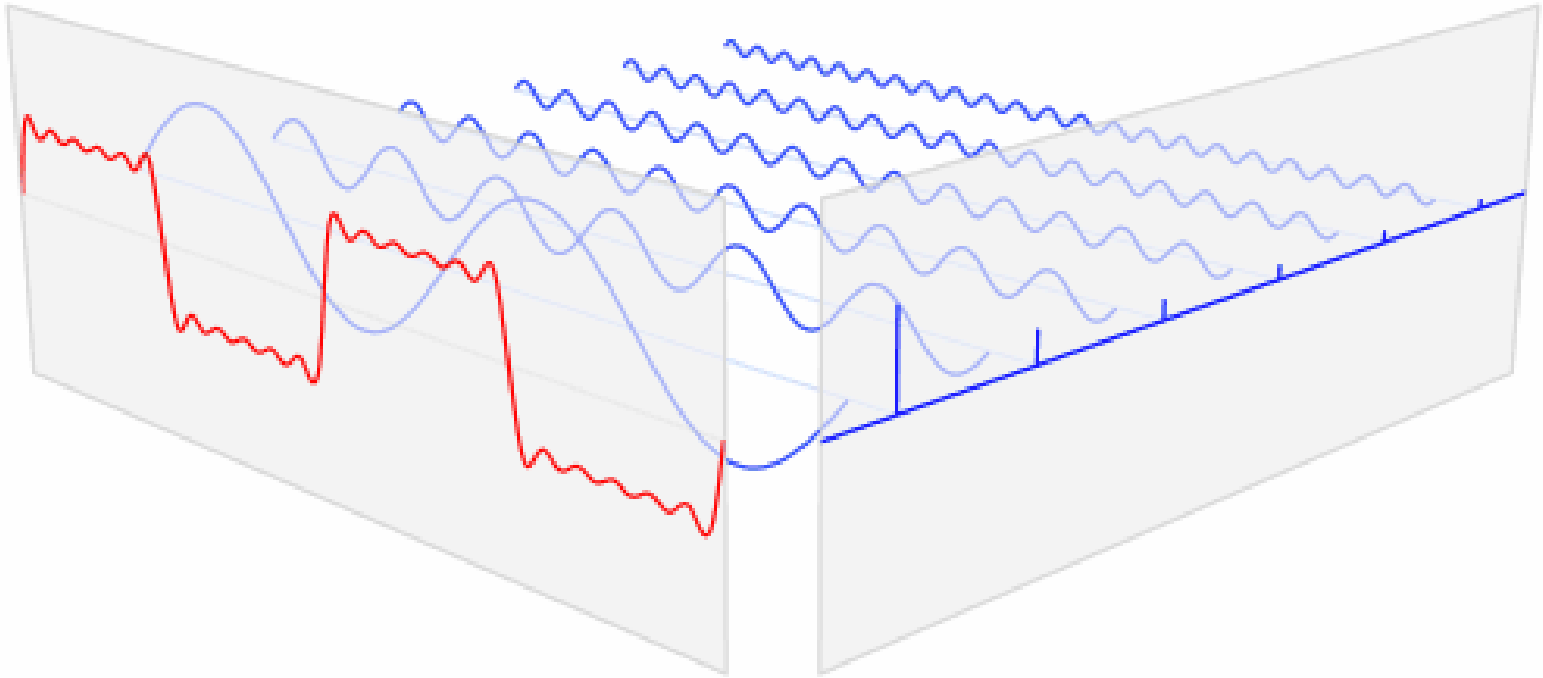
Outline

- Computations, Status and Condition Codes
- Arithmetic Operations
- Logical Operations
- Control of Computational Condition
- Control of Computational Flow



Knowledge from Fourier Transform

- Any signal or function is the sum of sine functions



Any sine function is the sum of arithmetic operations …

$$e^{jx} = \cos(x) + j \cdot \sin(x)$$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots \quad for\,|x| < 1$$

$$\tan^{-1}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots \quad for\,|x| < 1$$

Basic Types of Computations

Arithmetic

- x = y + z
- x = y - z
- x = y*y + y/z
- y = a*x + b
- etc

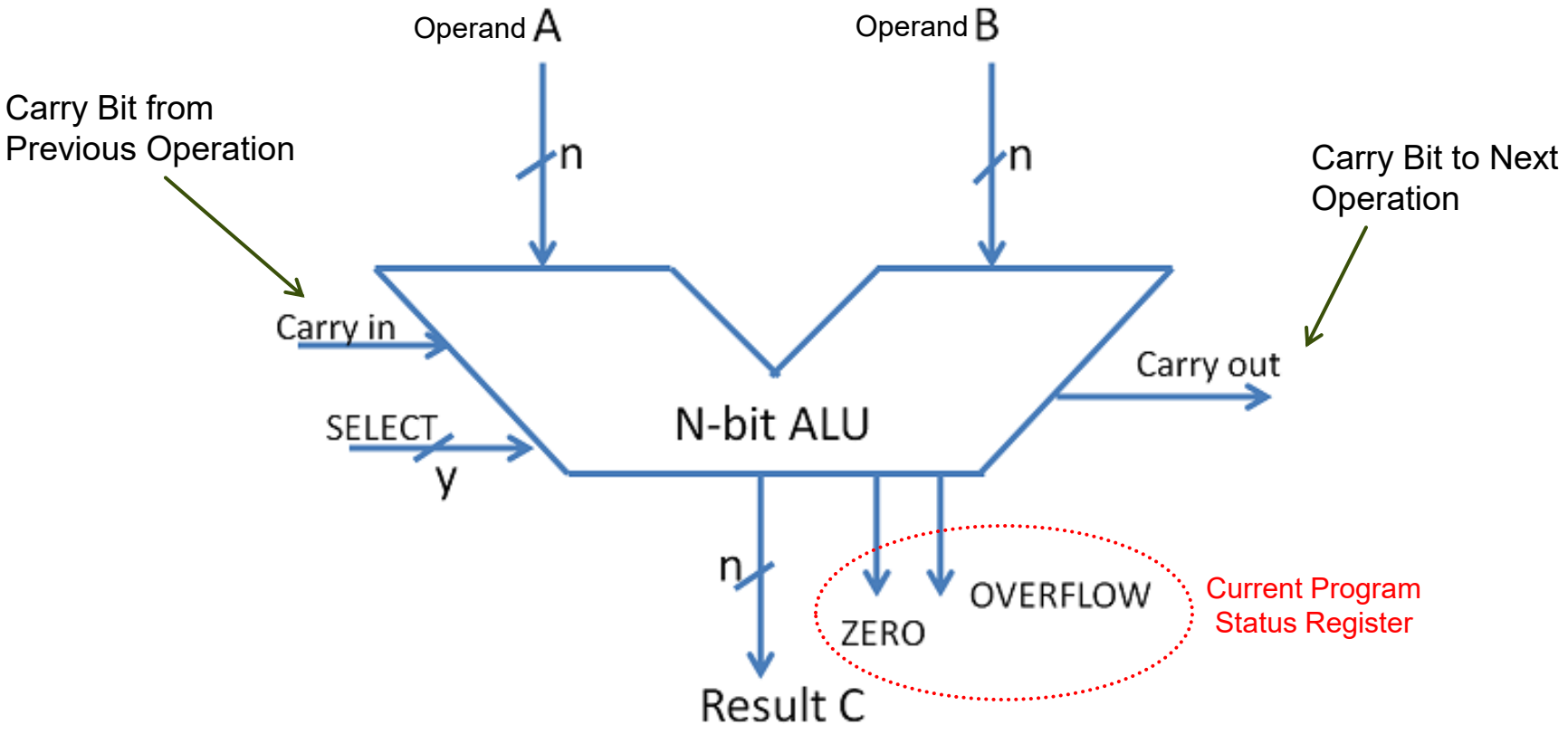
Logic

- A = B AND C
- A = B OR C
- A = B XOR C
- A = (B AND C) OR (C AND D)
- etc

The Taylor series of a real or complex-valued function $f(x)$ that is infinitely differentiable at a real or complex number a is the power series

$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots, \quad = \quad \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

All digital computers have ALU (i.e. Arithmetic and Logic Unit)



CPSR: Current Program Status Register

- ARM Cortex Microcontrollers have a 32-bit Register:

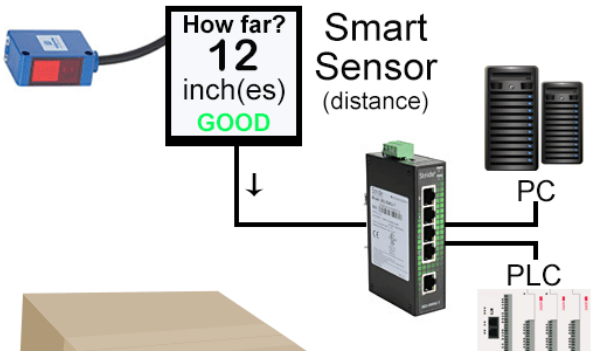
Field	Description	Architecture
N Z C V	Condition code flags	All
J	Jazelle state flag	5TEJ and above
GE[3:0]	SIMD condition flags	6
E	Endian Load/Store	6
A	Imprecise Abort Mask	6
I	IRQ Interrupt Mask	All
F	FIQ Interrupt Mask	All
T	Thumb state flag	4T and above
Mode[4:0]	Processor mode	All

Condition Codes

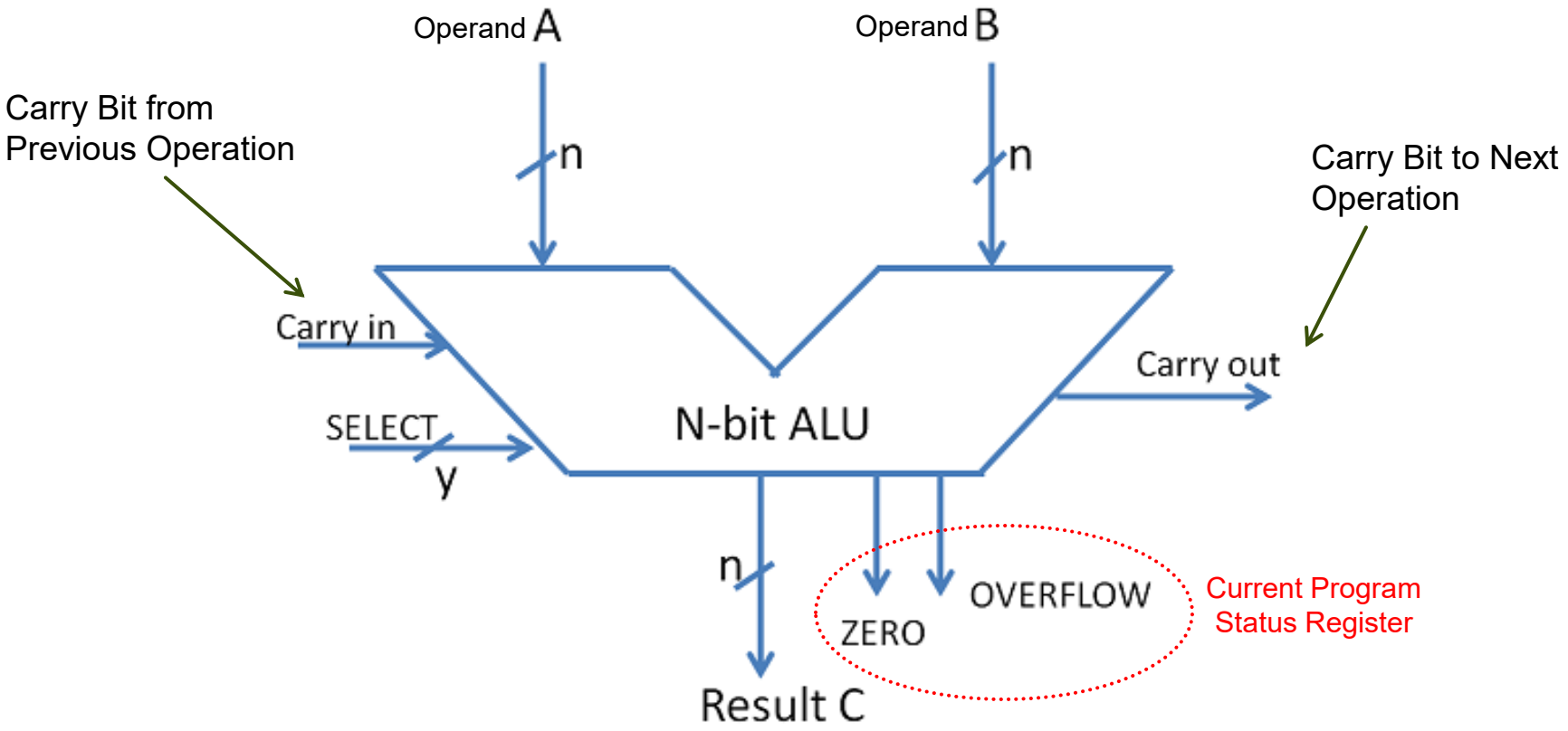
Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-

Outline

- Computations, Status and Condition Codes
- Arithmetic Operations
- Logical Operations
- Control of Computational Condition
- Control of Computational Flow

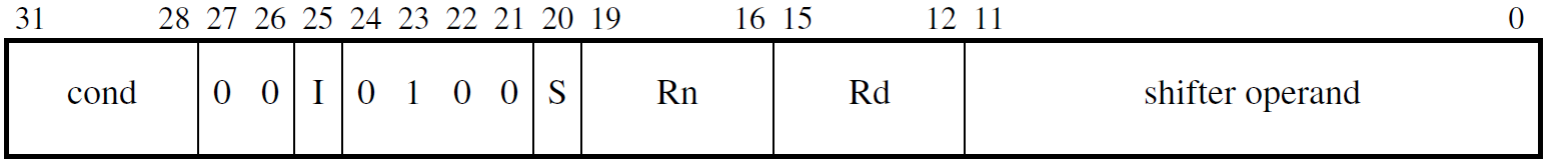


All digital computers have ALU (i.e. Arithmetic and Logic Unit)



ADD

- Encoding



ADD adds two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the addition.

ADD can optionally update the condition code flags, based on the result.

Syntax

```
ADD{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

S	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction.

Example

Use ADD to add two values together.

To increment a register value in Rx use:

```
ADD Rx, Rx, #1
```

You can perform constant multiplication of Rx by 2^n+1 into Rd with:

```
ADD Rd, Rx, Rx, LSL #n
```

shifter operand

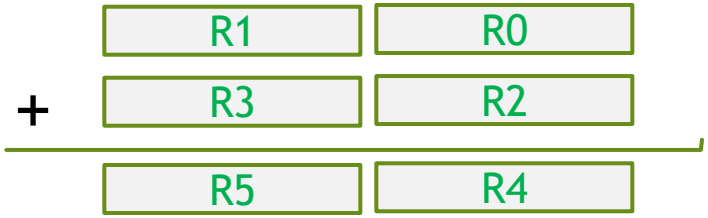
To form a PC-relative address use:

```
ADD Rd, PC, #offset
```

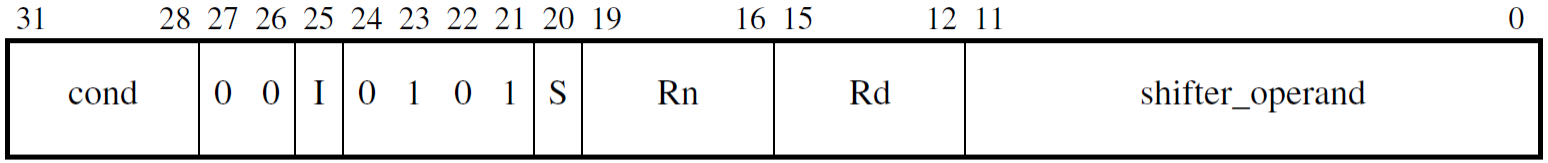
where the offset must be the difference between the required address and the address held in the PC, where the PC is the address of the ADD instruction itself plus 8 bytes.

```
ADDEQ r0, r1, r2          ; r0 = r1 + r2  (if zero flag is set)
```

ADC (to achieve addition of 64-bit values)



- Encoding



ADC (Add with Carry) adds two values and the Carry flag. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the addition.

ADC can optionally update the condition code flags, based on the result.

Syntax

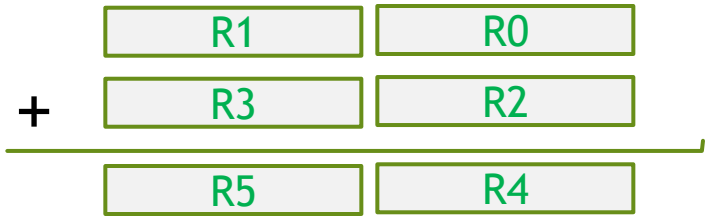
```
ADC{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

S	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction.

Example (to achieve addition of 64-bit values)

Use ADC to synthesize multi-word addition. If register pairs R0, R1 and R2, R3 hold 64-bit values (where R0 and R2 hold the least significant words) the following instructions leave the 64-bit sum in R4, R5:

```
ADDS R4,R0,R2
ADC  R5,R1,R3
```



If the second instruction is changed from:

```
ADC  R5,R1,R3
```

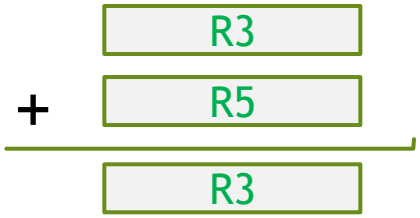
to:

```
ADCS R5,R1,R3
```

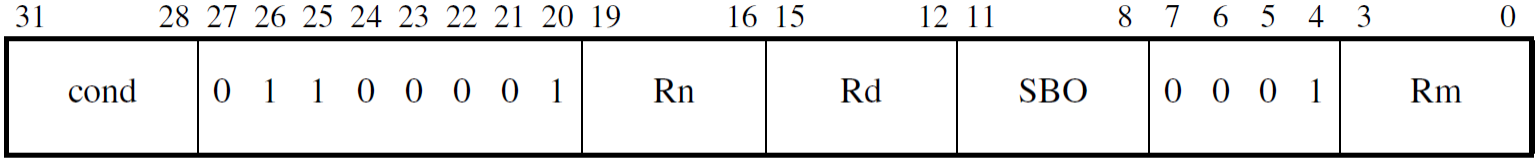
the resulting values of the flags indicate:

N	The 64-bit addition produced a negative result.
C	An unsigned overflow occurred.
V	A signed overflow occurred.
Z	The most significant 32 bits are all zero.

SADD16



- Encoding



SADD16 (Signed Add) performs two 16-bit signed integer additions. It sets the GE bits in the CPSR according to the results of the additions.

Syntax

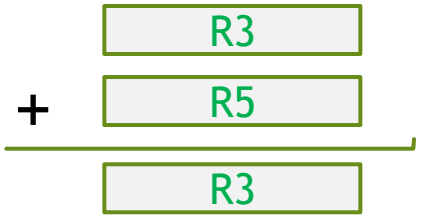
```
SADD16{<cond>}  <Rd>, <Rn>, <Rm>
```

<Rd>	Specifies the destination register.
<Rn>	Specifies the register that contains the first operand.
<Rm>	Specifies the register that contains the second operand.

Example

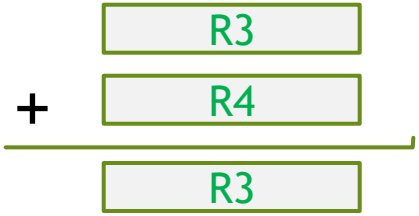
Use the SADD16 instruction to speed up operations on arrays of halfword data. For example, consider the instruction sequence:

```
LDR     R3, [R0], #4    ; Load four bytes and R0 skips four bytes
LDR     R5, [R1], #4    ; R5 = mem32[R1] then R1 = R1+4
SADD16  R3, R3, R5
STR     R3, [R2], #4    ; Store four bytes and R2 skips four bytes
```



This performs the same operations as the instruction sequence:

```
LDRH    R3, [R0], #2    ; Load two bytes and R0 skips two bytes
LDRH    R4, [R1], #2
ADD     R3, R3, R4
STRH    R3, [R2], #2    ; Store two bytes and R2 skips two bytes
```



LDRH

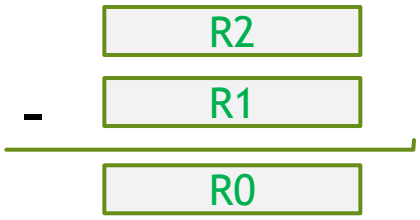
31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
cond	0	0	0	P	U	I	W	1	Rn	Rd	addr_mode	1	0	1	1	addr_mode

LDRH (Load Register Halfword) loads a halfword from memory and zero-extends it to a 32-bit word.

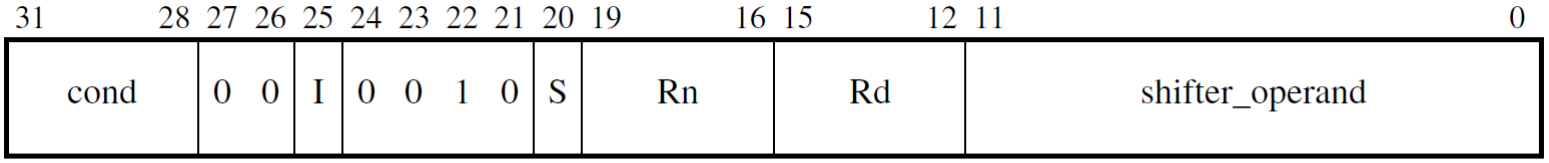
Syntax

```
LDR{<cond>}H  <Rd>, <addressing_mode>
```

SUB



- Encoding



SUB (Subtract) subtracts one value from a second value.

The second value comes from a register. The first value can be either an immediate value or a value from a register, and can be shifted before the subtraction.

SUB can optionally update the condition code flags, based on the result.

C flag If S is specified, the C flag is set to:
1 if no borrow occurs
0 if a borrow does occur.

Borrow = NOT(Carry) = C - 1

Syntax

```
SUB{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

<Rd> Specifies the destination register.

<Rn> Specifies the register that contains the first operand.

Example

C flag	If S is specified, the C flag is set to:
	1 if no borrow occurs
	0 if a borrow does occur.

Use SUB to subtract one value from another. To decrement a register value (in Ri) use:

```
SUB     Ri, Ri, #1
```

SUBS is useful as a loop counter decrement, as the loop branch can test the flags for the appropriate termination condition, without the need for a separate compare instruction:

```
SUBS    Ri, Ri, #1
```

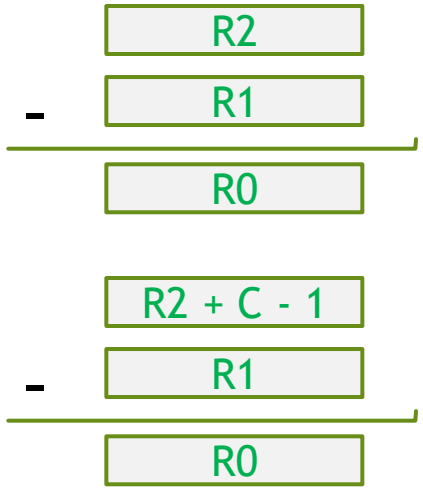
This both decrements the loop counter in Ri and checks whether it has reached zero.

RSB : reverse subtract

- RSB r0, r1, r2; r0 = r2 – r1

RSC : reverse subtract with carry

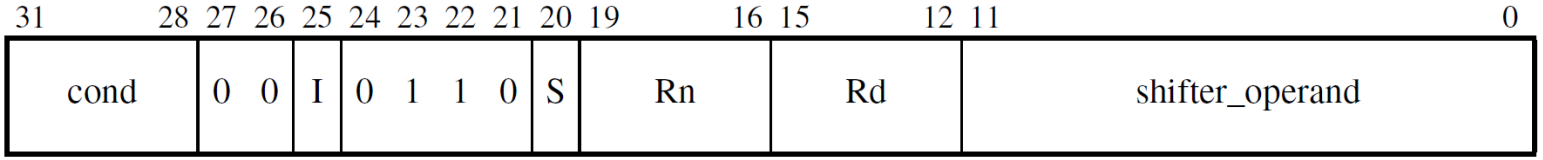
- RSC r0, r1, r2; r0 = r2 – r1 + C -1



SBC

C flag	If S is specified, the C flag is set to:
	1 if no borrow occurs
	0 if a borrow does occur.

- Encoding



SBC (Subtract with Carry) subtracts the value of its second operand and the value of NOT(Carry flag) from the value of its first operand. The first operand comes from a register. The second operand can be either an immediate value or a value from a register, and can be shifted before the subtraction.

Use SBC to synthesize multi-word subtraction.

SBC can optionally update the condition code flags, based on the result.

Syntax

```
SBC{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

Borrow = NOT(Carry) = C - 1

`<Rd>` Specifies the destination register.

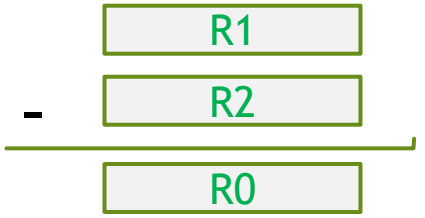
`<Rn>` Specifies the register that contains the first operand.

Example

C flag	If S is specified, the C flag is set to:
	1 if no borrow occurs
	0 if a borrow does occur.

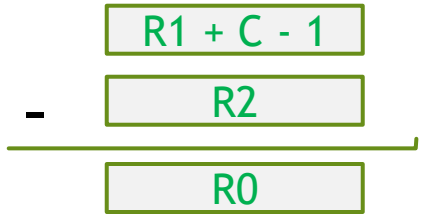
SUB : subtract

- SUB r0, r1, r2; r0 = r1 - r2



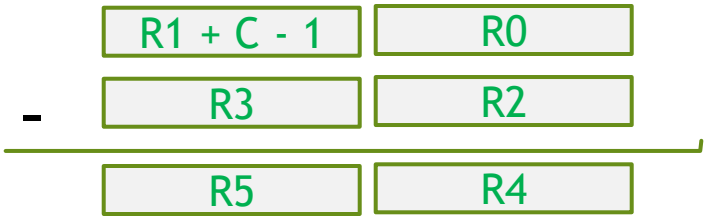
SBC : subtract with carry

- SBC r0, r1, r2; r0 = r1 - r2 + C -1



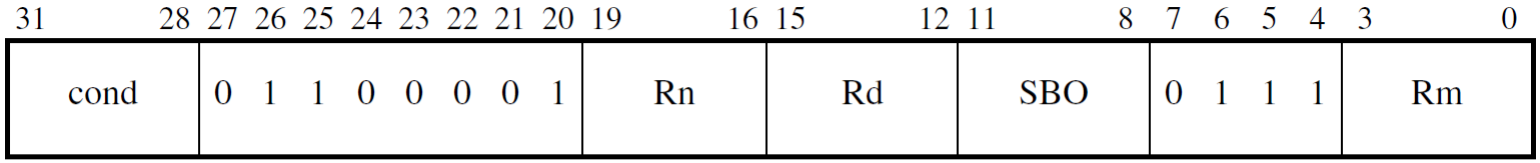
If register pairs R0,R1 and R2,R3 hold 64-bit values (R0 and R2 hold the least significant words), the following instructions leave the 64-bit difference in R4,R5:

```
SUBS    R4,R0,R2
SBC     R5,R1,R3
```



SSUB16

- Encoding



SSUB16 (Signed Subtract) performs two 16-bit signed integer subtractions. It sets the GE bits in the CPSR according to the results of the subtractions.

Syntax

```
SSUB16{<cond>}  <Rd>, <Rn>, <Rm>
```

<Rd>	Specifies the destination register.
<Rn>	Specifies the register that contains the first operand.
<Rm>	Specifies the register that contains the second operand.

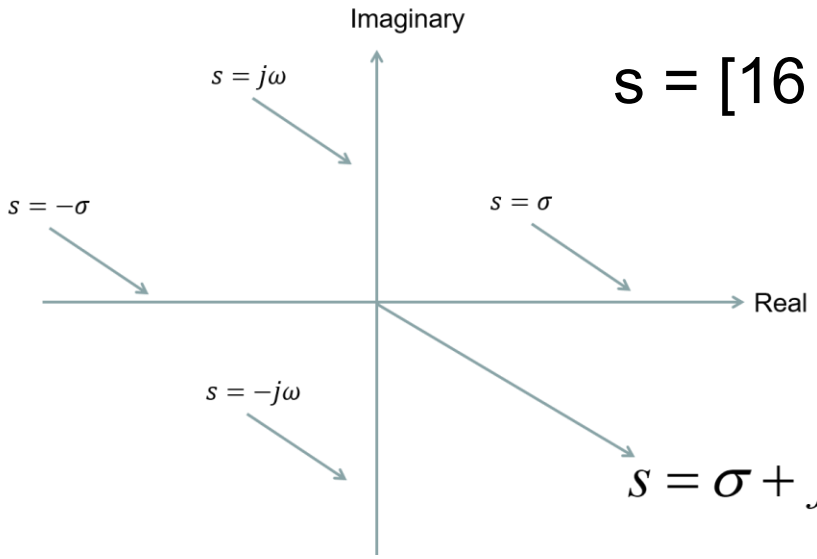
Example

Use SSUB16 to speed up operations on arrays of halfword data. This is similar to the way you can use SADD16.

You can also use SSUB16 for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
SSUB16  Rd, Ra, Rb
```

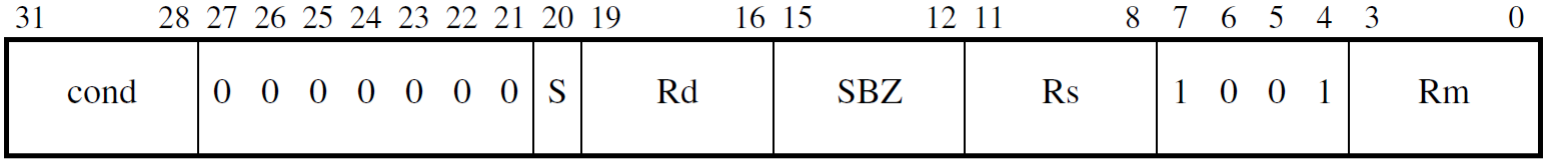
performs the complex arithmetic operation Rd = Ra - Rb.



s = [16 bits] + j [16 bits]

MUL

- Encoding



MUL (Multiply) multiplies two signed or unsigned 32-bit values. The least significant 32 bits of the result are written to the destination register.

MUL can optionally update the condition code flags, based on the result.

Syntax

```
MUL{<cond>}{S}  <Rd>, <Rm>, <Rs>
```

<Rd>	Specifies the destination register for the instruction.
<Rm>	Specifies the register that contains the first value to be multiplied.
<Rs>	Holds the value to be multiplied with the value of <Rm>.

Example

Arithmetic Operators

Multiplication

In assembly we write:

```
MUL   R2, R0, R1
MUL   R5, R4, R3
MUL   R8, R6, R7
```

0x00000002	R0
0x00000004	R1
0x00000008	R2
0xFFFFFF10	R3
0x00000077	R4
0xFFFF9070	R5
0x0000BEAD	R6
0x000157B5	R7
0x00009B51	R8
	R9
	R10
	R11
	R12
	R13
	R14
	R15

(there is an overflow)

With overflow

$$\begin{array}{r} 48813 \\ \times\ 87989 \\ \hline 39761 \end{array}$$

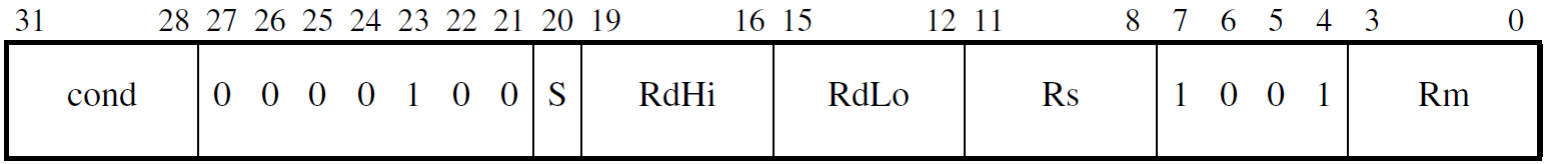
True Result

$$\begin{array}{r} 48813 \\ \times\ 87989 \\ \hline 4295007057 \end{array}$$

0xBEAD = 48813
0x157B5 = 87989
0x9B51 = 39761
0x100009B51 = 4295007057

UMULL

- Encoding



UMULL (Unsigned Multiply Long) multiplies the unsigned value of register <Rm> with the unsigned value of register <Rs> to produce a 64-bit result. The upper 32 bits of the result are stored in <RdHi>. The lower 32 bits are stored in <RdLo>. The condition code flags are optionally updated, based on the 64-bit result.

Syntax

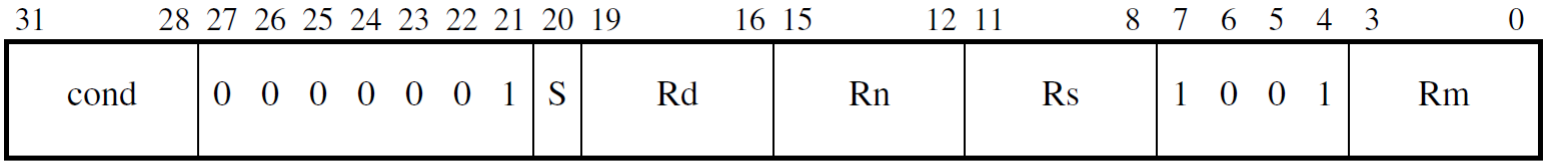
```
UMULL{<cond>}{S}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

<RdLo>	Stores the lower 32 bits of the result.
<RdHi>	Stores the upper 32 bits of the result.
<Rm>	Holds the signed value to be multiplied with the value of <Rs>.
<Rs>	Holds the signed value to be multiplied with the value of <Rm>.

MLA

$$y = a*x + b$$

- Encoding



MLA (Multiply Accumulate) multiplies two signed or unsigned 32-bit values, and adds a third 32-bit value. The least significant 32 bits of the result are written to the destination register.

MLA can optionally update the condition code flags, based on the result.

Syntax

MLA r0, r1, r2, r3; r0 = (r1 * r2) + r3

```
MLA{<cond>}{S}  <Rd>, <Rm>, <Rs>, <Rn>
```

<Rd>	Specifies the destination register.
<Rm>	Holds the value to be multiplied with the value of <Rs>.
<Rs>	Holds the value to be multiplied with the value of <Rm>.
<Rn>	Contains the value that is added to the product of <Rs> and <Rm>.

UDIV (Could Be Done with Shift)

Syntax

`UDIV{cond} {Rd}, Rn, Rm`

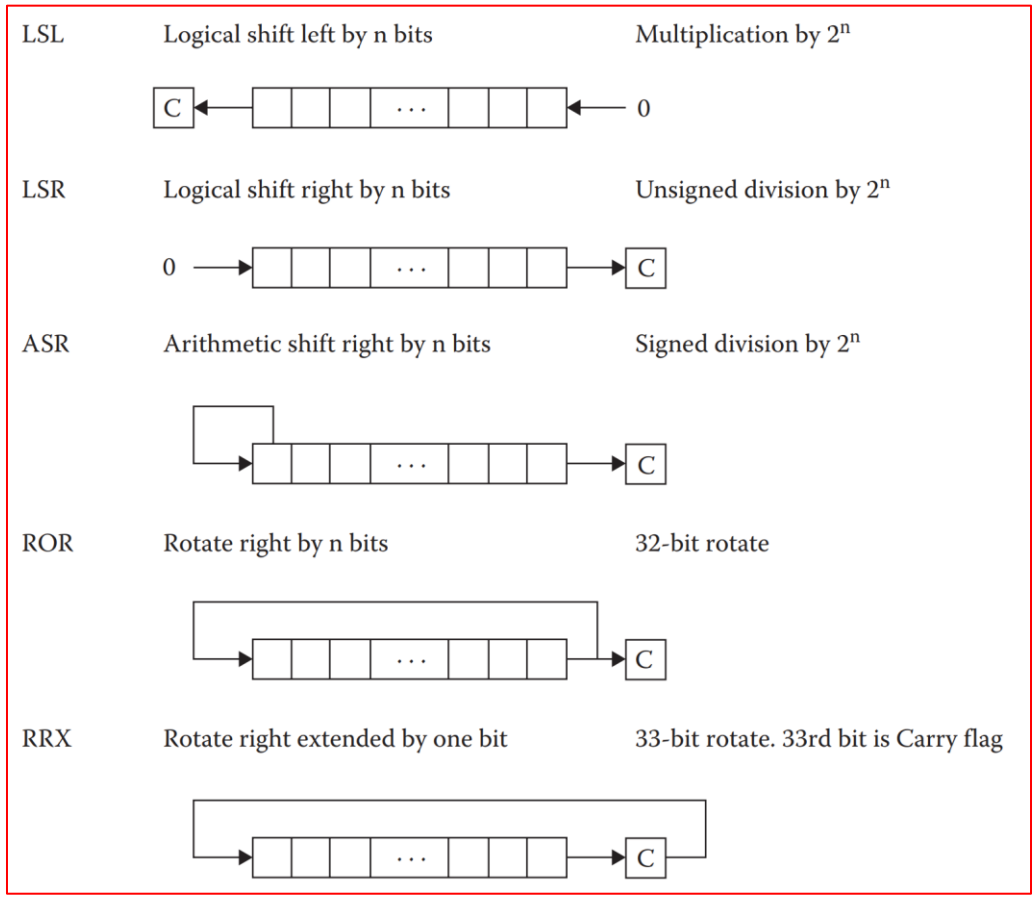
where:

cond
is an optional condition code.

Rd
is the destination register.

Rn
is the register holding the value to be divided.

Rm
is a register holding the divisor.



Register restrictions

PC or SP cannot be used for `Rd`, `Rn`, or `Rm`.

Architectures

This 32-bit Thumb instruction is available in ARMv7-R and ARMv7-M.
This ARM instruction is optional in ARMv7-R.
There is no 16-bit Thumb `UDIV` instruction.

Shift Operators

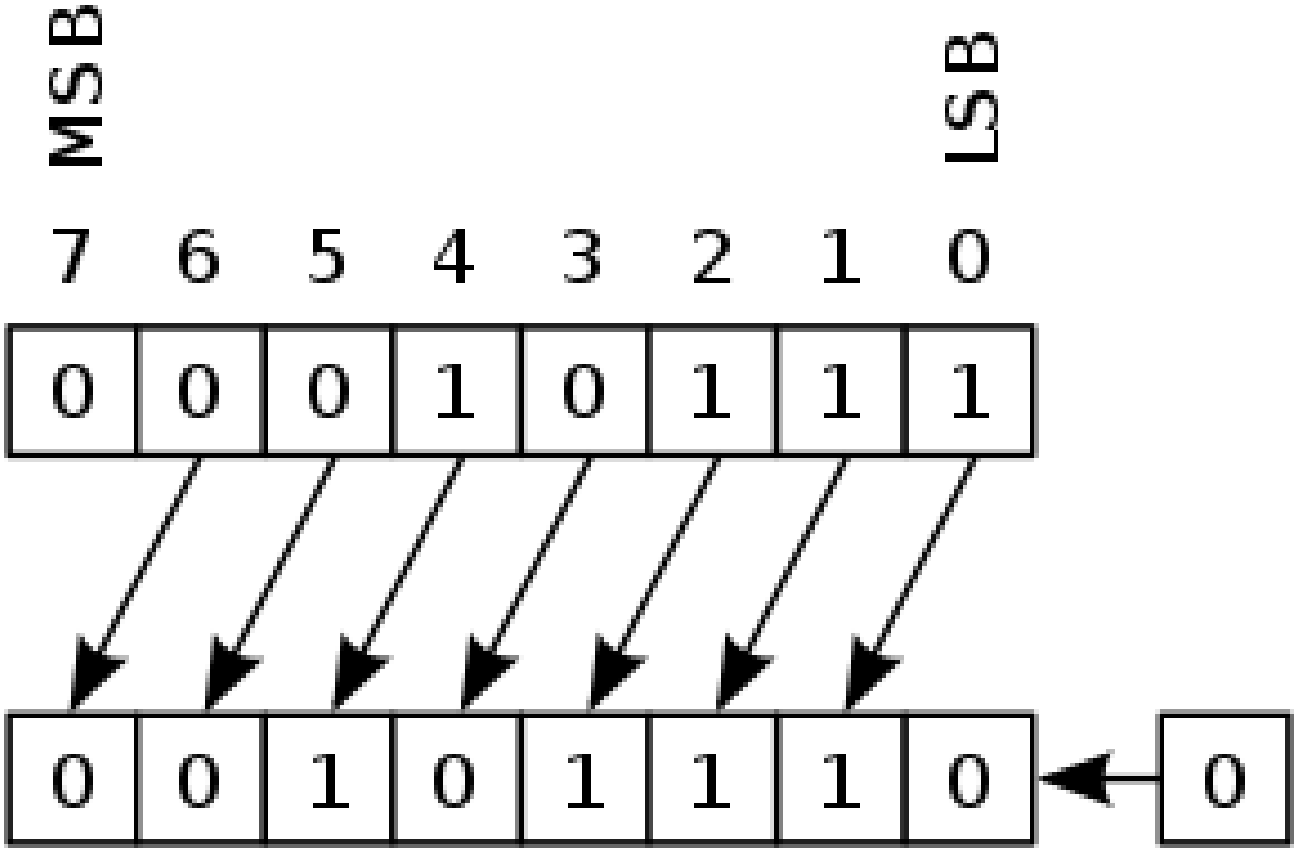
- LSL: logical shift left
  - x << y, the least significant bits are filled with zeroes
- LSR: logical shift right:
  - (unsigned) x >> y, the most significant bits are filled with zeroes
- ASR: arithmetic shift right
  - (signed) x >> y, copy the sign bit to the most significant bit
- ROR: rotate right
  - ((unsigned) x >> y) | (x << (32-y))
- *RRX: rotate right extended*
  - c flag << 31 | ((unsigned) x >> 1)
  - Performs 33-bit rotate, with the CPSR's C bit being inserted above sign bit of the word

Bit 31 does not shift

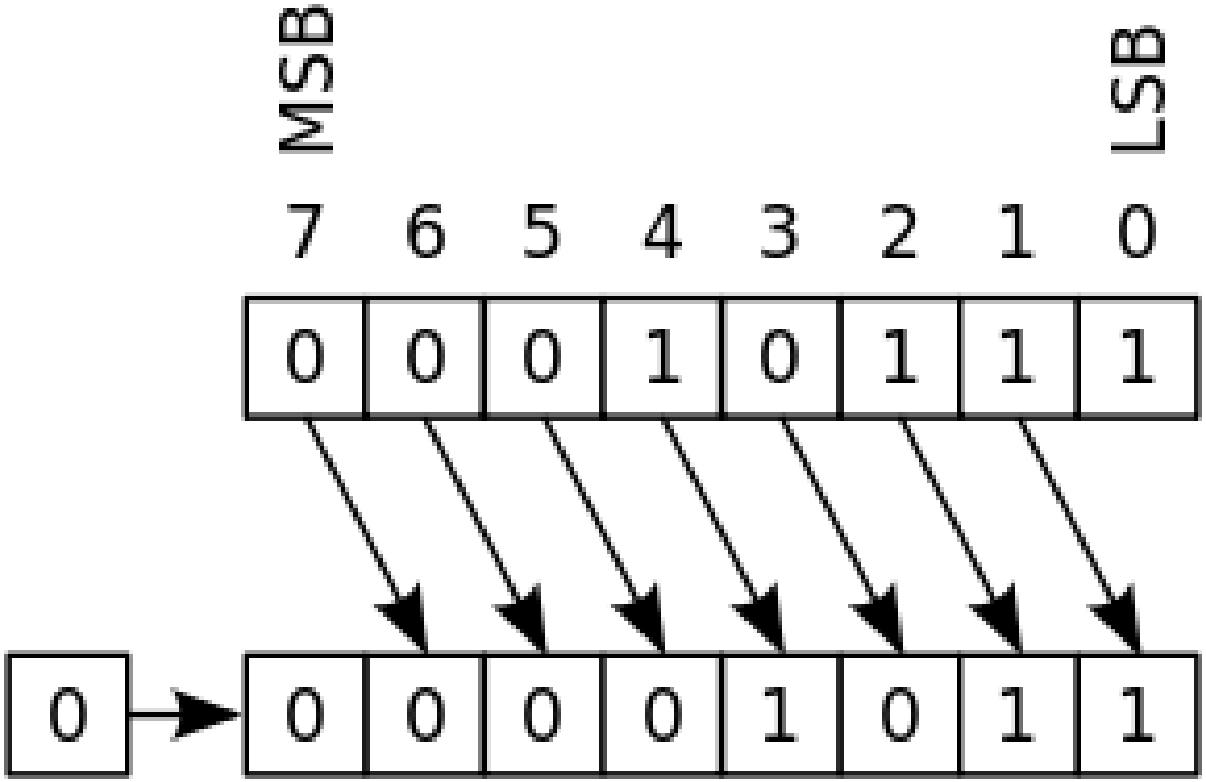
y = 7; 32 – y = 25; 25 + 6 = 31

(0000000b31…b7) | (b6…b0 0…0)

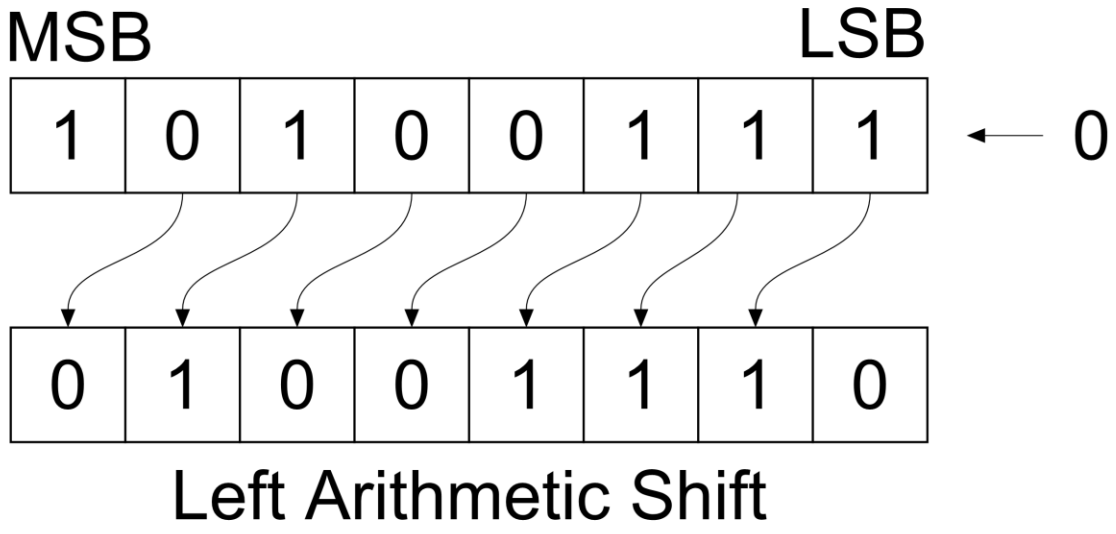
Logic Shift Left

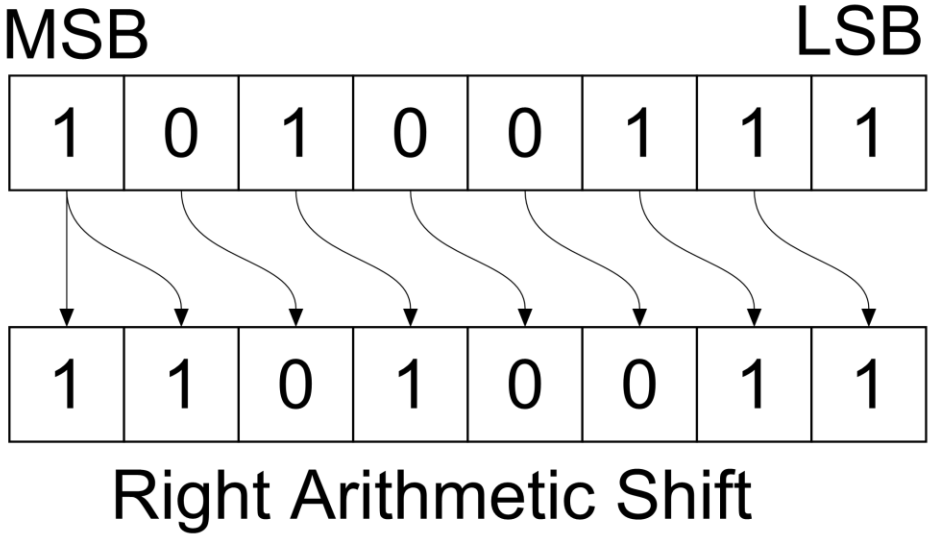


Logic Shift Right

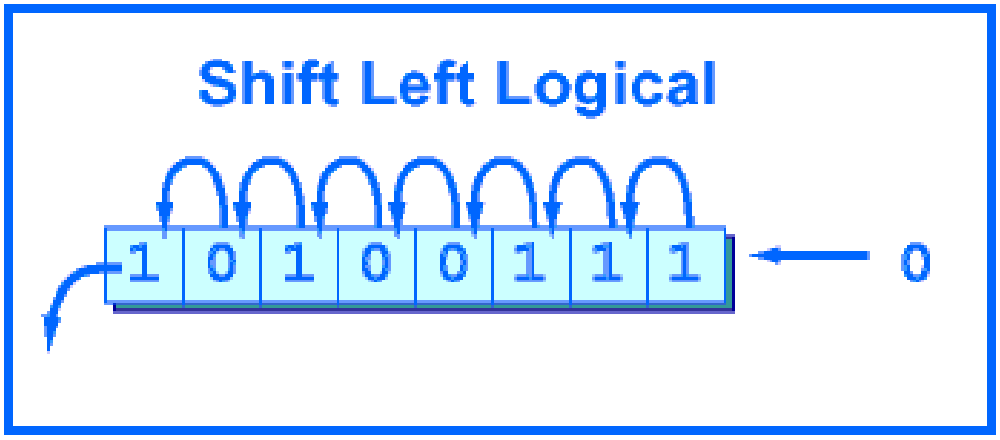


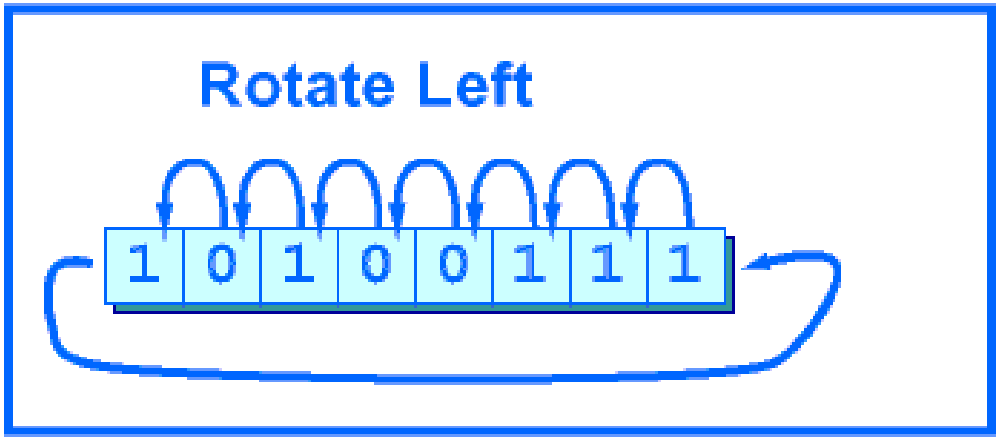
Arithmetic Shift



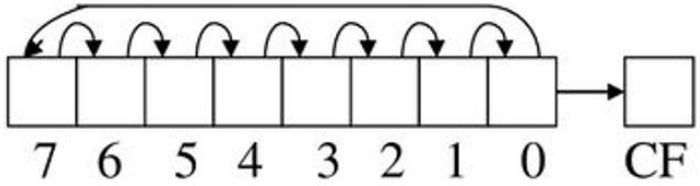


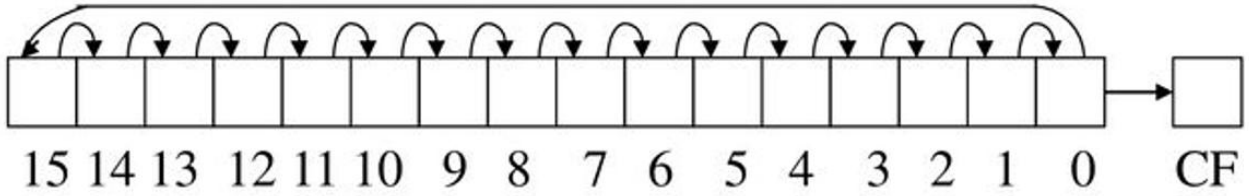
Shift Left versus Rotate Left





Rotate Right





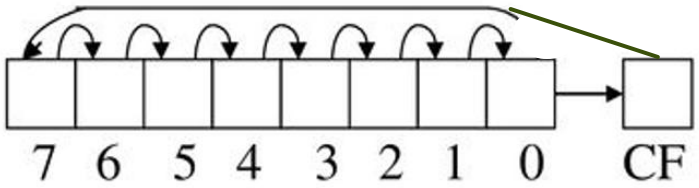
x >> y; y=7

y = 7; 32 – y = 25; 25 + 6 = 31

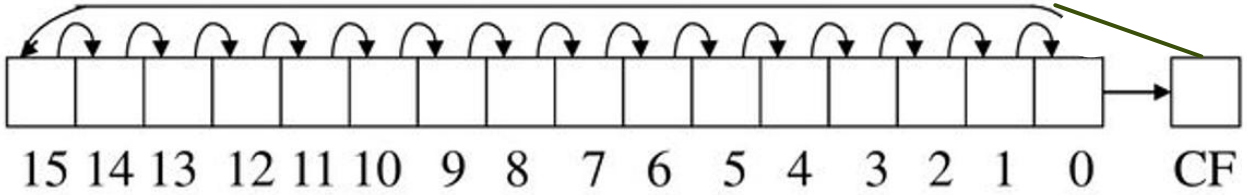
(0000000b31…b7) | (b6…b0 0…0)

Rotate Right Extended

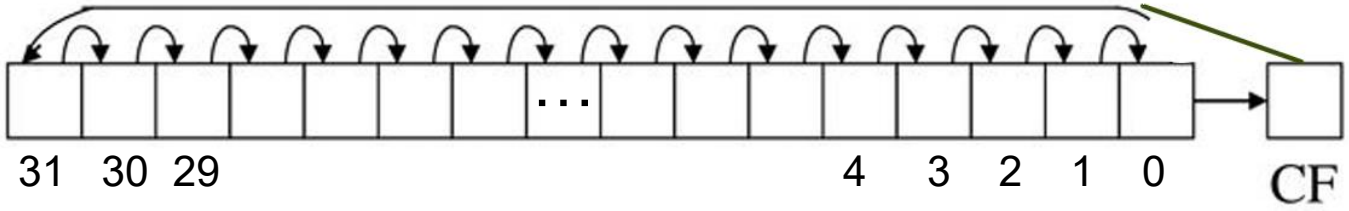
Perform 9-bit rotate-right



Perform 17-bit rotate-right



Perform 33-bit rotate-right

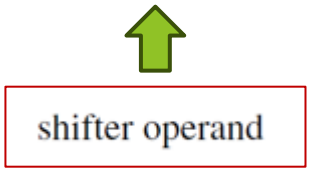


Example

`LSL Rd, Rm, #0 to 31`	Logical Shift Left	Rd := Rm LSL 5-bit immediate
`LSL Rd, Rs`	Logical Shift Left	Rd := Rd LSL Rs
`LSR Rd, Rm, #1 to 32`	Logical Shift Right	Rd := Rm LSR 5-bit immediate
`LSR Rd, Rs`	Logical Shift Right	Rd := Rd LSR Rs

MOV r0, r1, LSL #1 ; r0 = r1 *2

MOV r7, r5, LSL #2; r7 = r5 << 2 = r5*4



Example of Shifting Data

```
        AREA Prog1, CODE, READONLY
        ENTRY

        MOV     r0, #0x11       ; load initial value
        LSL     r1, r0, #1      ; shift 1 bit left
        LSL     r2, r1, #1      ; shift 1 bit left

stop    B       stop            ; stop program
        END
```

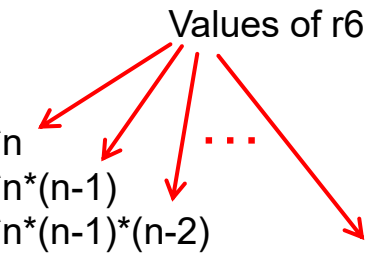
Example of Doing n! (factorial of n)

```
        AREA Prog2, CODE, READONLY
        ENTRY
        MOV     r6,#10          ; load n into r6
        MOV     r7,#1           ; if n=0, at least n!=1
loop    CMP     r6, #0
        MULGT   r7, r6, r7      ; 1*n*(n-1)*(n-2)*…*2*1
        SUBGT   r6, r6, #1      ; decrement n
        BGT     loop            ; do another mul if counter!= 0
stop    B       stop            ; stop program
        END
```

What is the value at r7?

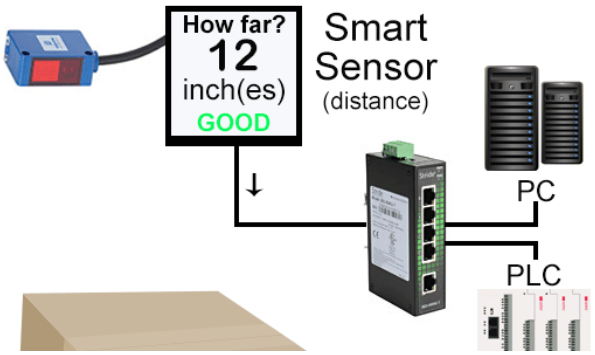
Answer:

1. Initial value: 1
2. After Loop 1: 1*n
3. After Loop 2: 1*n*(n-1)
4. After Loop 3: 1*n*(n-1)*(n-2)
5. After Final Loop: 1*n*(n-1)*(n-2)*…*2*1

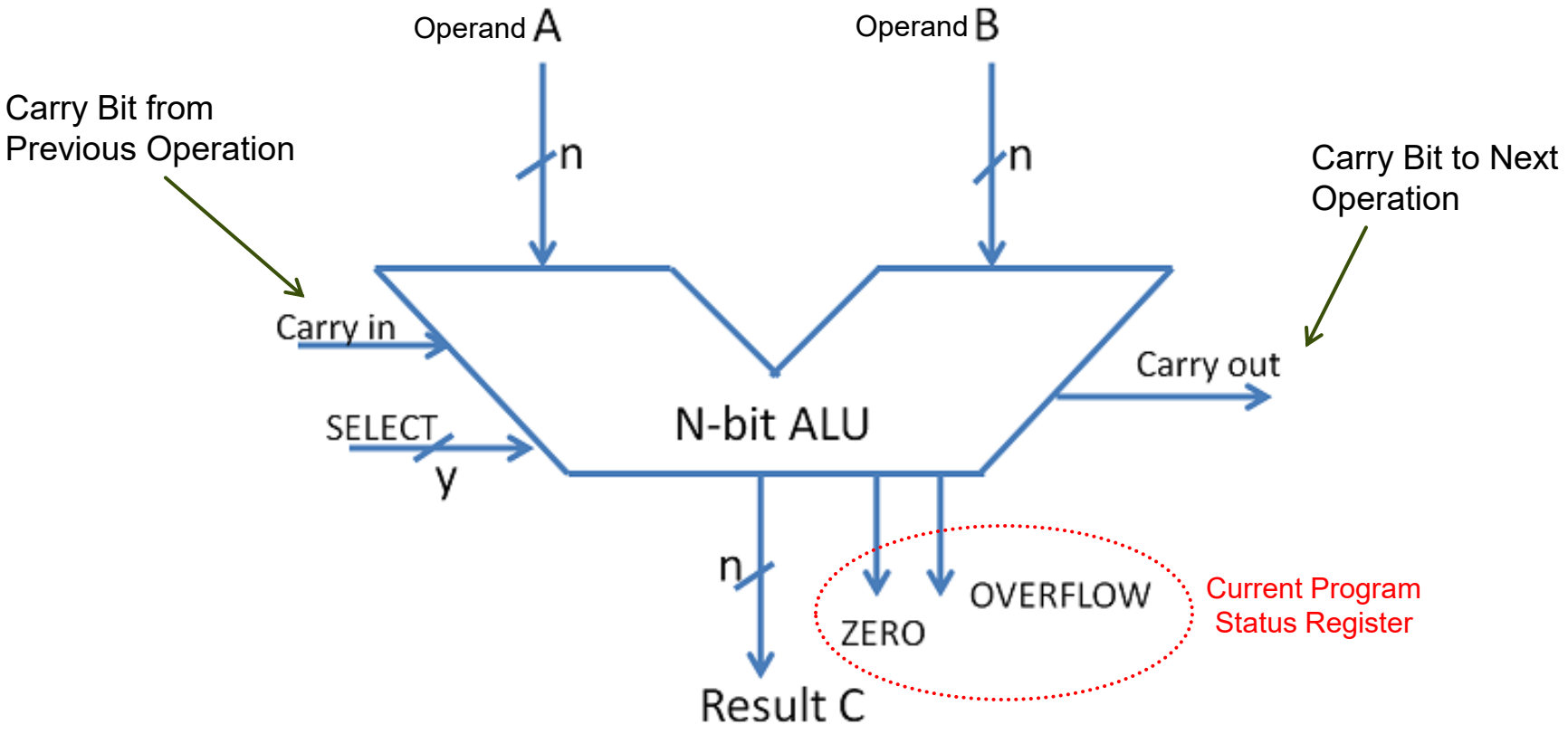


Outline

- Computations, Status and Condition Codes
- Arithmetic Operations
- Logical Operations
- Control of Computational Condition
- Control of Computational Flow

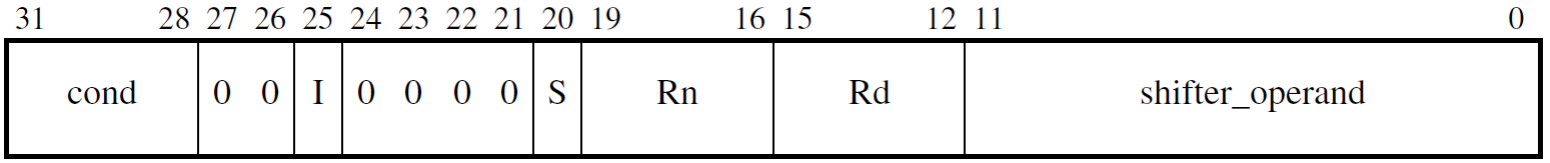


All digital computers have ALU (i.e. Arithmetic and Logic Unit)



AND

- Encoding



AND performs a bitwise AND of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the AND operation.

AND can optionally update the condition code flags, based on the result.

Syntax

```
AND{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

<Rd>	Specifies the destination register.
<Rn>	Specifies the register that contains the first operand.

0 0 1 1 1 1 0 0
1 0 1 0 1 0 1 0
AND
0 0 1 0 1 0 0 0

Example

Bit Banging

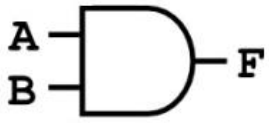
Forcing bits to 0

R0 = 0000 0001 0001 0000

~R0 = 1111 1110 1110 1111

R1 = 0000 0000 0001 0001

R3 = 1010 1011 1100 1101



A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

In C we would write:

```
# define MASK 0x0110
   DestA = SrcA & ~MASK;
   DestB = SrcB & ~MASK;
```

In assembly we write:

```
MOVW R0, #0x0110
MVN  R0, R0
AND  R2, R1, R0
AND  R4, R3, R0
```

0xFFFFFEEF	R0
0x00000011	R1
0x00000001	R2
0x0000ABCD	R3
0x0000AACD	R4
	R5
	R6
	R7
	R8
	R9
	R10
	R11
	R12
	R13
	R14
	R15

~R0 = 1111 1110 1110 1111

R1 = 0000 0000 0001 0001

R2 = 0000 0000 0000 0001

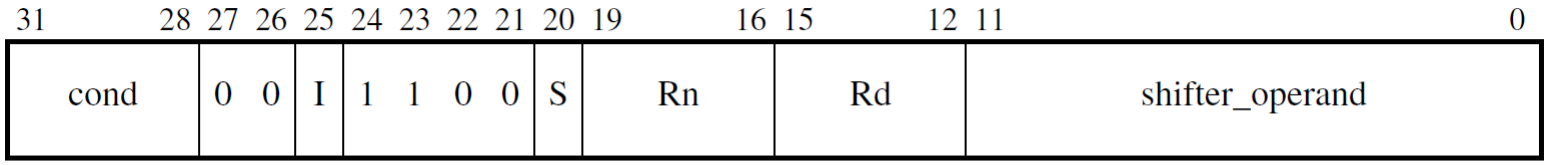
~R0 = 1111 1110 1110 1111

R3 = 1010 1011 1100 1101

R4 = 1010 1010 1100 1101

ORR

- Encoding



ORR (Logical OR) performs a bitwise (inclusive) OR of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the OR operation.

ORR can optionally update the condition code flags, based on the result.

Syntax

```
ORR{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

<Rd> Specifies the destination register.

<Rn> Specifies the register that contains the first operand.

0 0 1 1 1 1 0 0
1 0 1 0 1 0 1 0
ORR
1 0 1 1 1 1 1 0

Example

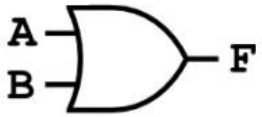
Bit Banging

Forcing bits to 1

R0 = 0000 0001 0001 0000

R1 = 0000 0000 0001 0001

R3 = 1010 1011 1100 1101



A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

In C we would write:

```
# define MASK 0x0110
   DestA = SrcA | MASK;
   DestB = SrcB | MASK;
```

In assembly we write:

```
MOVW R0, #0x0110
ORR  R2, R1, R0
ORR  R4, R3, R0
```

0x00000110	R0
0x00000011	R1
0x00000111	R2
0x0000ABCD	R3
0x0000ABDD	R4
	R5
	R6
	R7
	R8
	R9
	R10
	R11
	R12
	R13
	R14
	R15

R0 = 0000 0001 0001 0000

R1 = 0000 0000 0001 0001

R2 = 0000 0001 0001 0001

R0 = 0000 0001 0001 0000

R3 = 1010 1011 1100 1101

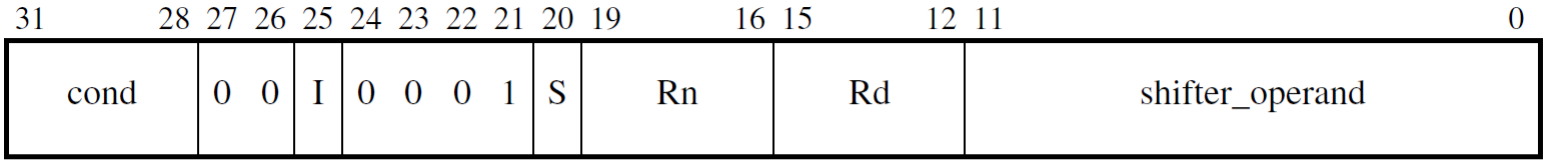
R4 = 1010 1011 1101 1101

EOR

Use EOR to invert selected bits in a register. For each bit, EOR with 1 inverts that bit, and EOR with 0 leaves it unchanged.

How to implement bitwise Negation?

- Encoding



EOR (Exclusive OR) performs a bitwise Exclusive-OR of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the exclusive OR operation.

EOR can optionally update the condition code flags, based on the result.

Syntax

```
EOR{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
```

<Rd>	Specifies the destination register.
<Rn>	Specifies the register that contains the first operand.

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

Example

Bit Banging

Flipping bits

R0 = 0000 0001 0001 0000

R1 = 0000 0000 0001 0001

R3 = 1010 1011 1100 1101

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

In C we would write:

```
# define MASK 0x0110
   DestA = SrcA ^ MASK;
   DestB = SrcB ^ MASK;
```

In assembly we write:

```
MOVW R0, #0x0110
EOR  R2, R1, R0
EOR  R4, R3, R0
```

0x00000110	R0
0x00000011	R1
0x00000101	R2
0x0000ABCD	R3
0x0000AADD	R4
	R5
	R6
	R7
	R8
	R9
	R10
	R11
	R12
	R13
	R14
	R15

R0 = 0000 0001 0001 0000

R1 = 0000 0000 0001 0001

R2 = 0000 0001 0000 0001

R0 = 0000 0001 0001 0000

R3 = 1010 1011 1100 1101

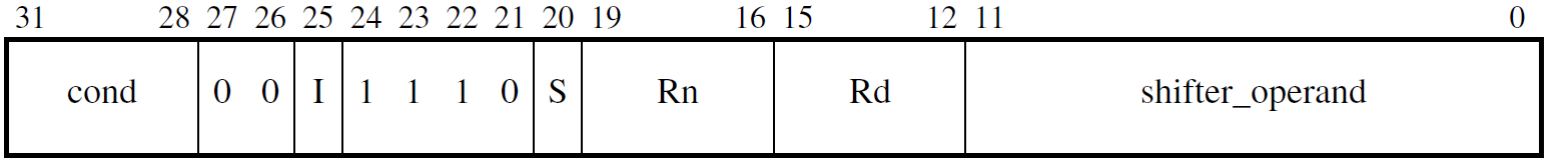
R4 = 1010 1010 1101 1101

BIC

(What does it mean by saying "write 1 to clear a bit"?)

The second operand contains the flags indicating the bits in the first operand to be cleared

- Encoding



BIC (Bit Clear) performs a bitwise AND of one value with the complement of a second value. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the BIC operation.

BIC can optionally update the condition code flags, based on the result.

Syntax

`BIC{<cond>}{S} <Rd>, <Rn>, <shifter_operand>`

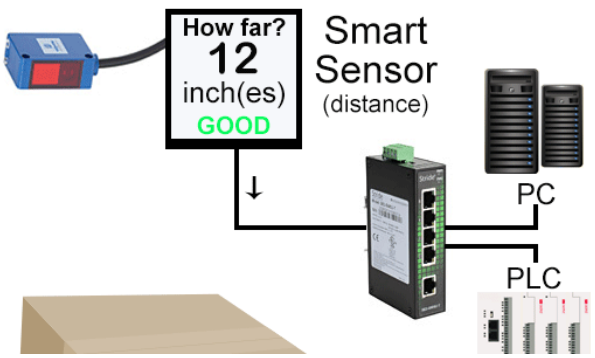
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.

```
  0 0 1 1 1 1 0 0
  1 0 1 0 1 0 1 0
BIC
  0 0 0 1 0 1 0 0
```

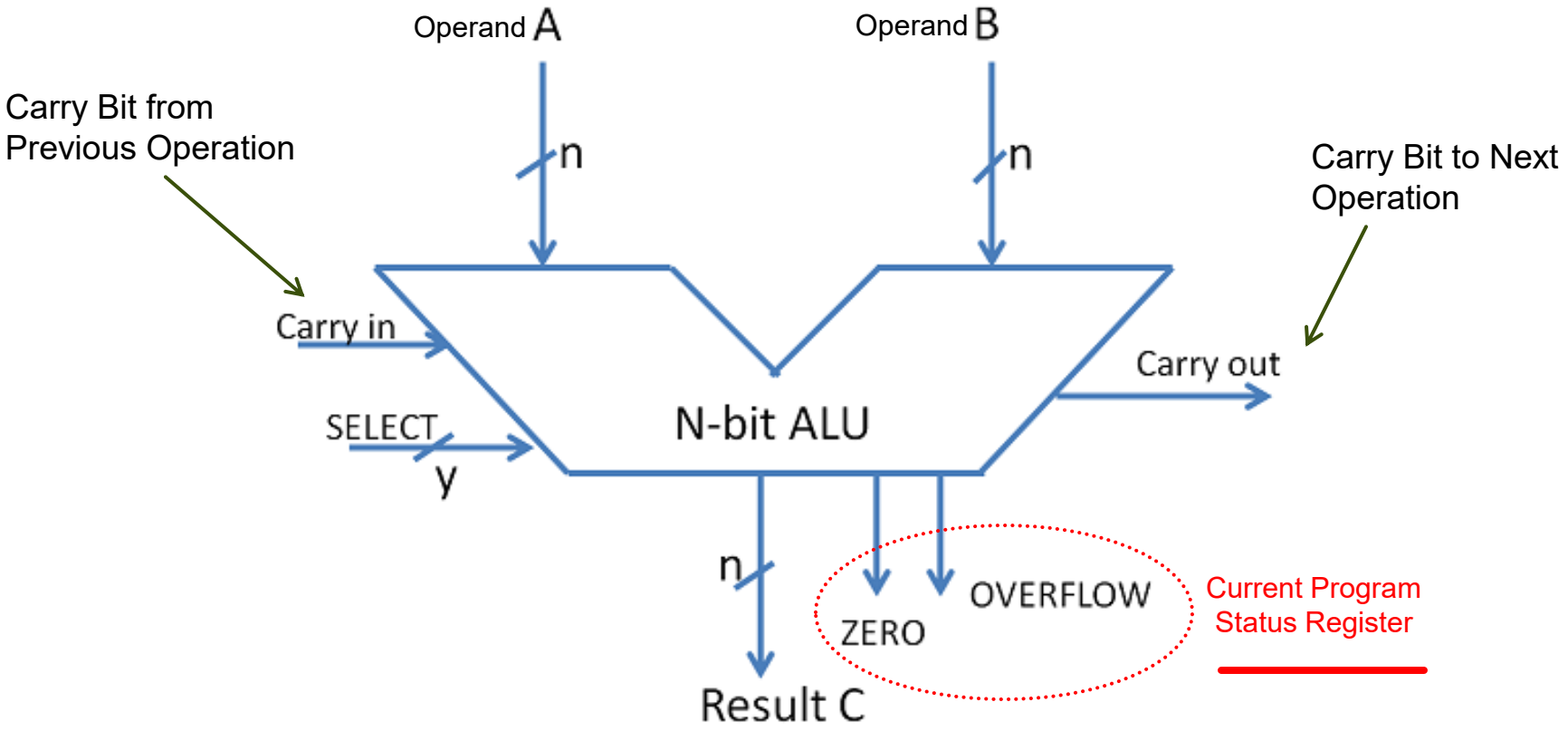
BIC r0, r1, r2 ; r0 = r1 & Not (r2)

Outline

- Computations, Status and Condition Codes
- Arithmetic Operations
- Logical Operations
- Control of Computational Condition
- Control of Computational Flow



All digital computers have ALU (i.e. Arithmetic and Logic Unit)



ARM Cortex Offers Four Instructions …

CMP : compare

- CMP r0, r1; compute (r0 - r1) and set NZCV

CMN : negated compare

- CMN r0, r1; compute (r0 + r1) and set NZCV

TST : bit-wise AND test

- TST r0, r1; compute (r0 AND r1) and set NZCV

TEQ : bit-wise exclusive-or test

- TEQ r0, r1; compute (r0 EOR r1) and set NZC

AND Gate

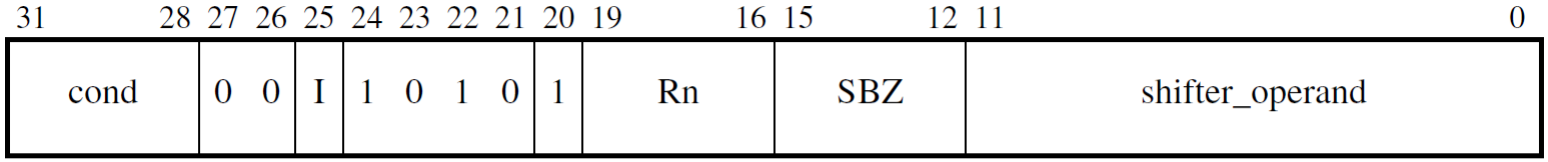
INPUT		OUTPUT
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

Exclusive OR Gate

INPUT		OUTPUT
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

CMP

- Encoding



CMP (Compare) compares two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the comparison.

CMP updates the condition flags, based on the result of subtracting the second value from the first.

Syntax

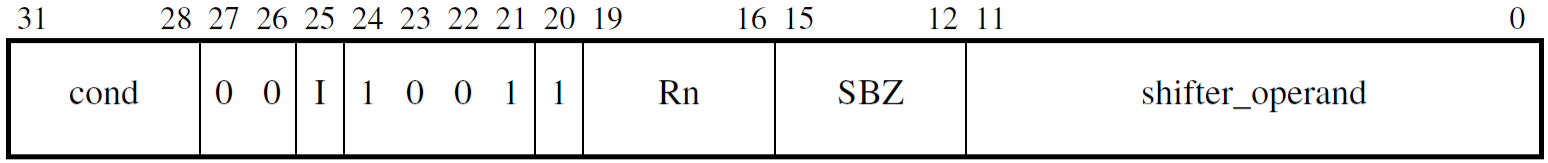
```
CMP{<cond>}  <Rn>, <shifter_operand>
```

```
If condition x < y
do
{
        codes;
}
else
{
        codes;
}
```

TEQ

Use TEQ to test if two values are equal, without affecting the V flag (as CMP does). The C flag is also unaffected in many cases. TEQ is also useful for testing whether two values have the same sign. After the comparison, the N flag is the logical Exclusive OR of the sign bits of the two operands.

- Encoding



TEQ (Test Equivalence) compares a register value with another arithmetic value. The condition flags are updated, based on the result of logically exclusive-ORing the two values, so that subsequent instructions can be conditionally executed.

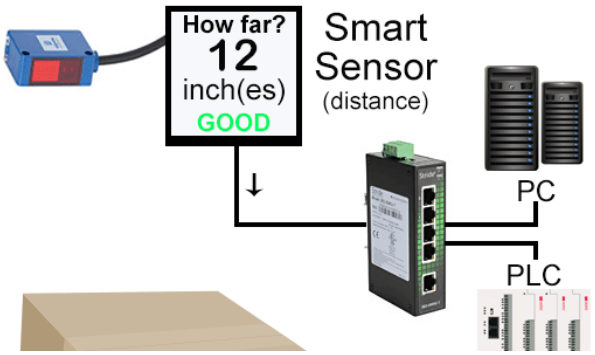
Syntax

```
TEQ{<cond>}  <Rn>, <shifter_operand>
```

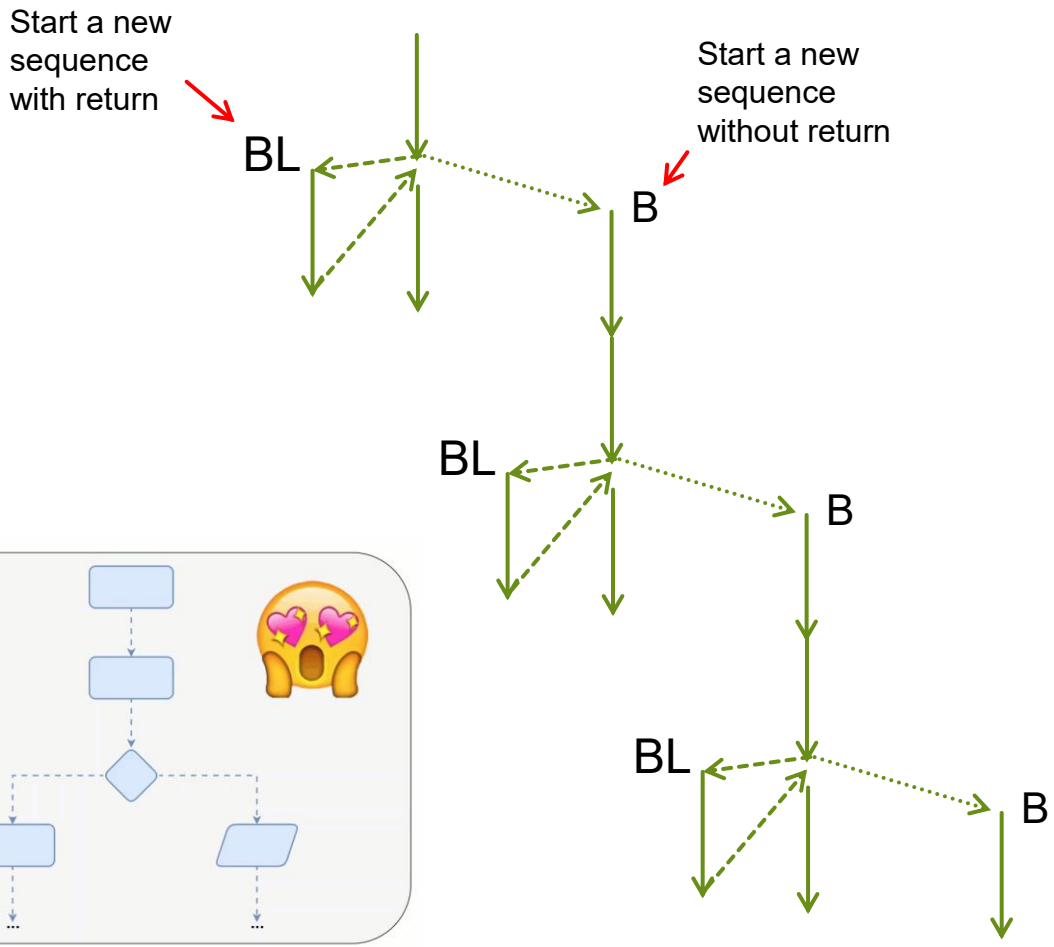
```
If condition x < y
do
{
        codes;
}
else
{
        codes;
}
```

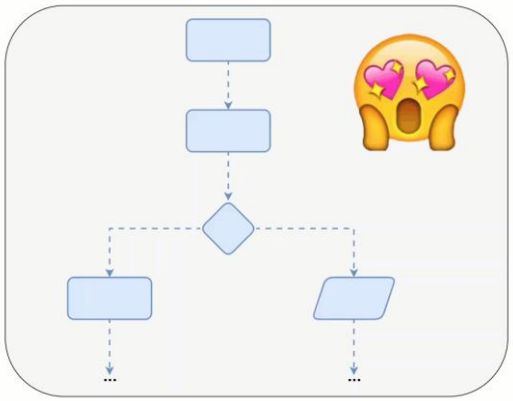
Outline

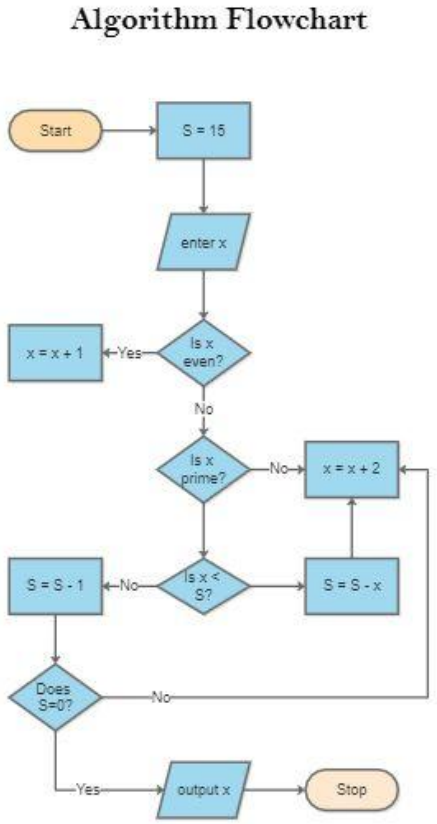
- Computations, Status and Condition Codes
- Arithmetic Operations
- Logical Operations
- Control of Computational Condition
- Control of Computational Flow



A computational flow may consist of multiple sequences of instructions …

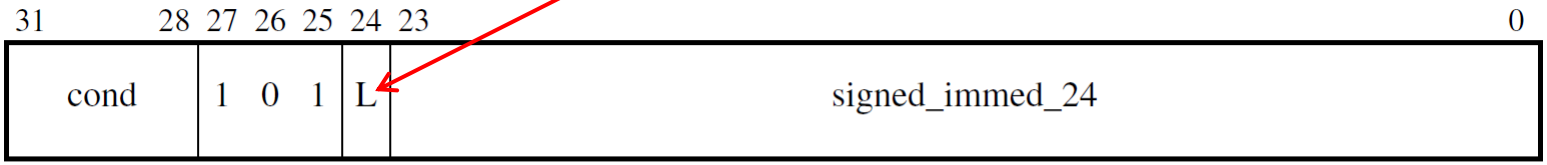






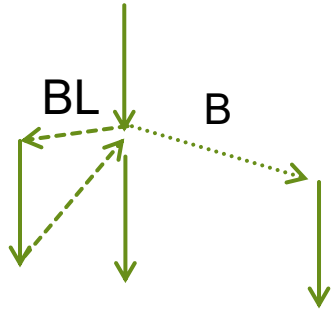
B (without return) or BL (with return)

- Encoding



B (Branch) and BL (Branch and Link) cause a branch to a target address, and provide both conditional and unconditional changes to program flow.

BL also stores a return address in the link register, R14 (also known as LR).



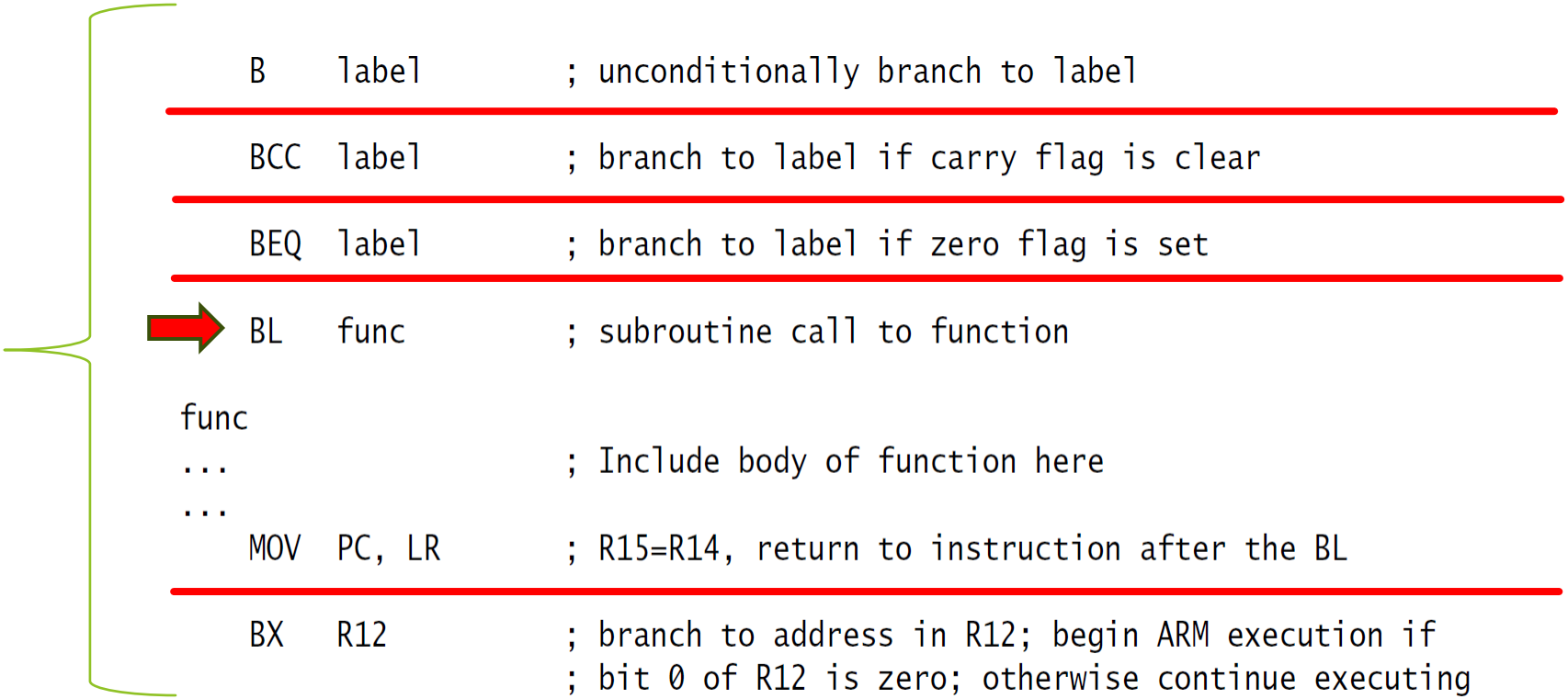
Syntax

`B{L}{<cond>}  <target_address>`

where:

L	Causes the L bit (bit 24) in the instruction to be set to 1. The resulting instruction stores a return address in the link register (R14). If L is omitted, the L bit is 0 and the instruction simply branches without storing a return address.

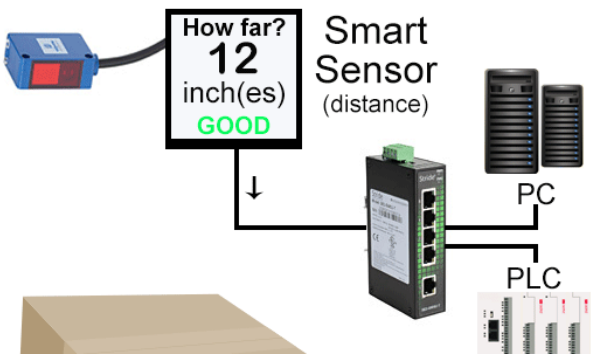
Examples



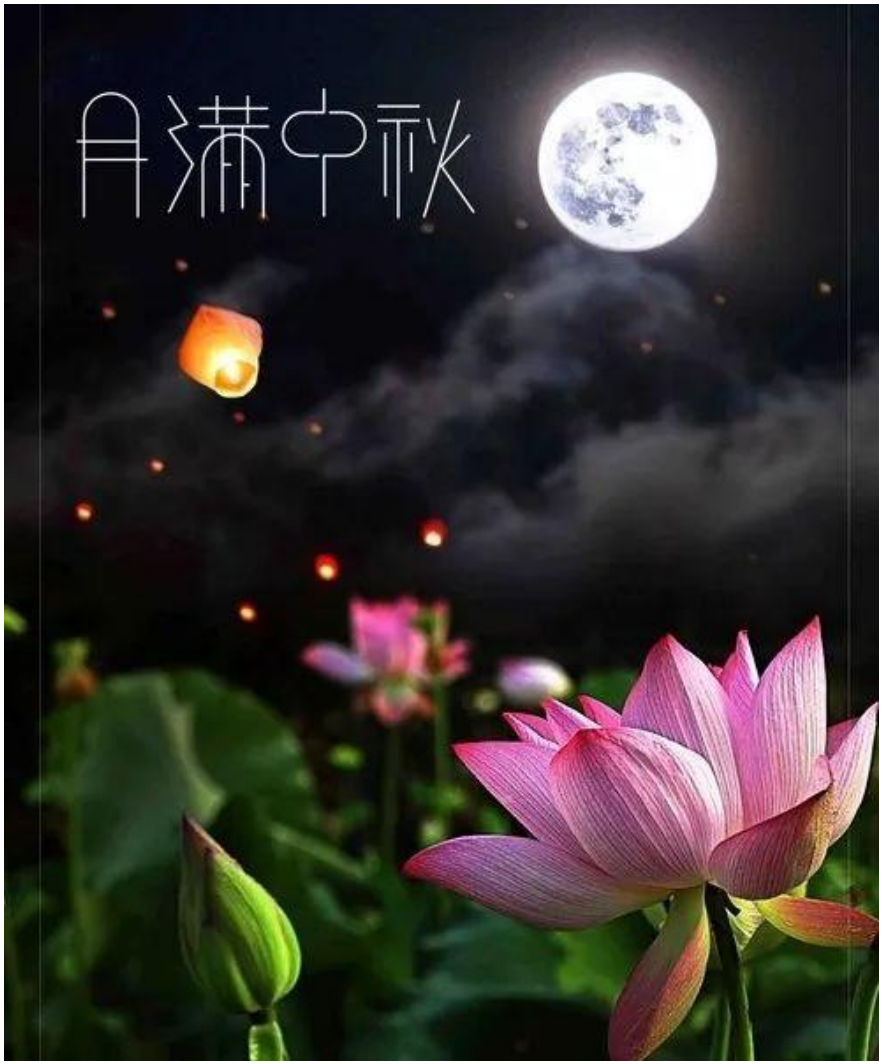
```
    B    label        ; unconditionally branch to label

    BCC  label        ; branch to label if carry flag is clear

    BEQ  label        ; branch to label if zero flag is set

    BL   func         ; subroutine call to function

func
...                   ; Include body of function here
...
    MOV  PC, LR       ; R15=R14, return to instruction after the BL

    BX   R12          ; branch to address in R12; begin ARM execution if
                      ; bit 0 of R12 is zero; otherwise continue executing
                      ; Thumb code
```

Summary

- Computations, Status and Condition Codes
- Arithmetic Operations
- Logical Operations
- Control of Computational Condition
- Control of Computational Flow



Happy Mid-Autumn Festival (2026.9.25)



"Ask not what your country can do for you – ask what you can do for your country," - John F. Kennedy

"Do not think that you are needy – think that you are needed in the world", - Manis Friedman

"Study will make you knowledgeable, resourceful, and hence more needed", - Xie Ming

Thank You for Listening!