Chapter 2
Instruction Set

Dr. Md Abu Sayeed
EET 340

Operands

Low level language (Assembly Language): An operand is a value (an argument) on which the instruction operates. The operand may be a processor register, a memory address, or a literal constant.

High level language (C/C++/Java): Operands are the constants or variables which the operators operate upon.

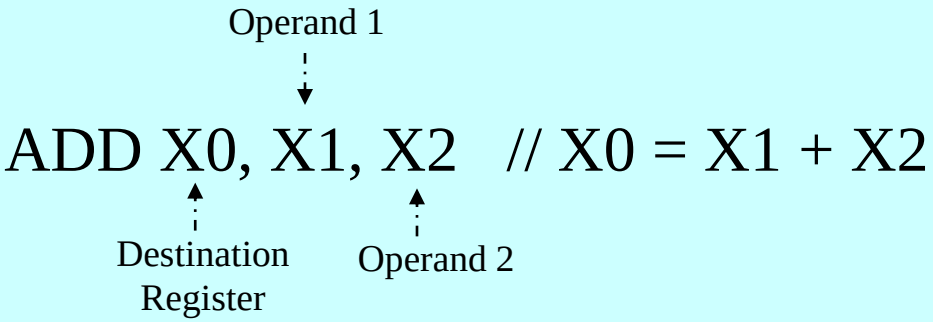
LEGv8 Registers

Name	Register number	Usage
X0-X7	0–7	Arguments/Results
X8	8	Indirect result location register
X9-X15	9–15	Temporaries
X16 (IP0)	16	May be used by linker as a scratch register; other times used as temporary register
X17 (IP1)	17	May be used by linker as a scratch register; other times used as temporary register
X18	18	Platform register for platform independent code; otherwise a temporary register
X19-X27	19–27	Saved
X28 (SP)	28	Stack Pointer
X29 (FP)	29	Frame Pointer
X30 (LR)	30	Link Register (return address)
XZR	31	The constant value 0

Arithmetic Instructions

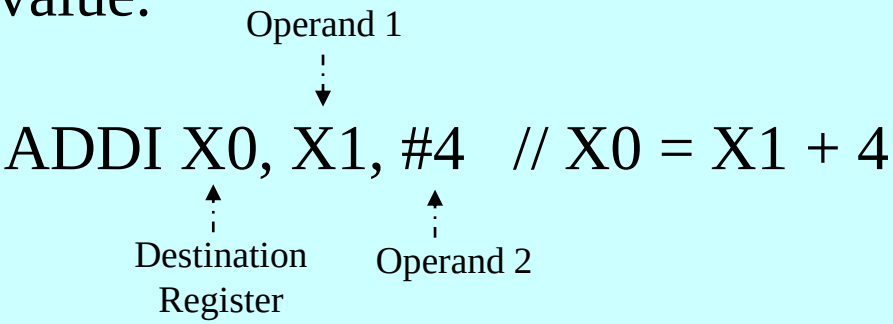
ADD -

- The ADD instruction adds two operands and place the result on destination register. Both operands are register.



ADDI -

- The ADD instruction adds two operands and place the result on destination register. Operand 1 is a register, operand 2 is an immediate value.



SUB -

- The SUB instruction subtracts operand 2 from operand 1 and place the result on destination register. Both operands are register.

```
SUB X0, X1, X2   // X0 = X1 - X2
```

SUBI -

- The SUBI instruction subtracts operand 2 from operand 1 and place the result on destination register. Operand 1 is a register, operand 2 is an immediate value.

```
SUBI X0, X1, #4   // X0 = X1 - 4
```

MOV -

- This instruction loads a 64-bit value into the destination register from another register.

```
MOV  X1, X2   // X1 = X2
```

MOVI -

- This instruction loads a 64-bit value into the destination register from an immediate value.

```
MOVI  X1, #8   // X1 = 8
```

LSL – Logical shift left

- LSL instruction effectively multiply the contents of a register by 2^i. For example, the below LSL shifts by 2, which multiply the contents of X2 by 2^2 or 4 and place the result on destination register X0.
- Each bit of the register is shifted left, the MSB is removed and empty bits are filled with zeros.

```
LSL  X0, X2, #2   // X0 = X2*4
```

Problem: Assume that X2 = 2 (0X0000000000000002 in hexadecimal). What will be the value of X0 after running the following instruction: LSL X0, X2, #2

X2 → 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000010 2

1st shift→ 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000100 4

2nd shift→ 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001000 8

The content of X2 will be 8 (0X0000000000000008 in hexadecimal)

LSR – Logical shift Right

- LSR instruction effectively divide the contents of a register by 2^i. For example, the below LSR shifts by 3, which divide the contents of X2 by 2^3 or 8 and place the result on destination register X0.
- Each bit of the register is shifted right, the LSB is removed and empty bits are filled with zeros.

```
LSR  X0, X2, #3        // X0 = X2/8
```

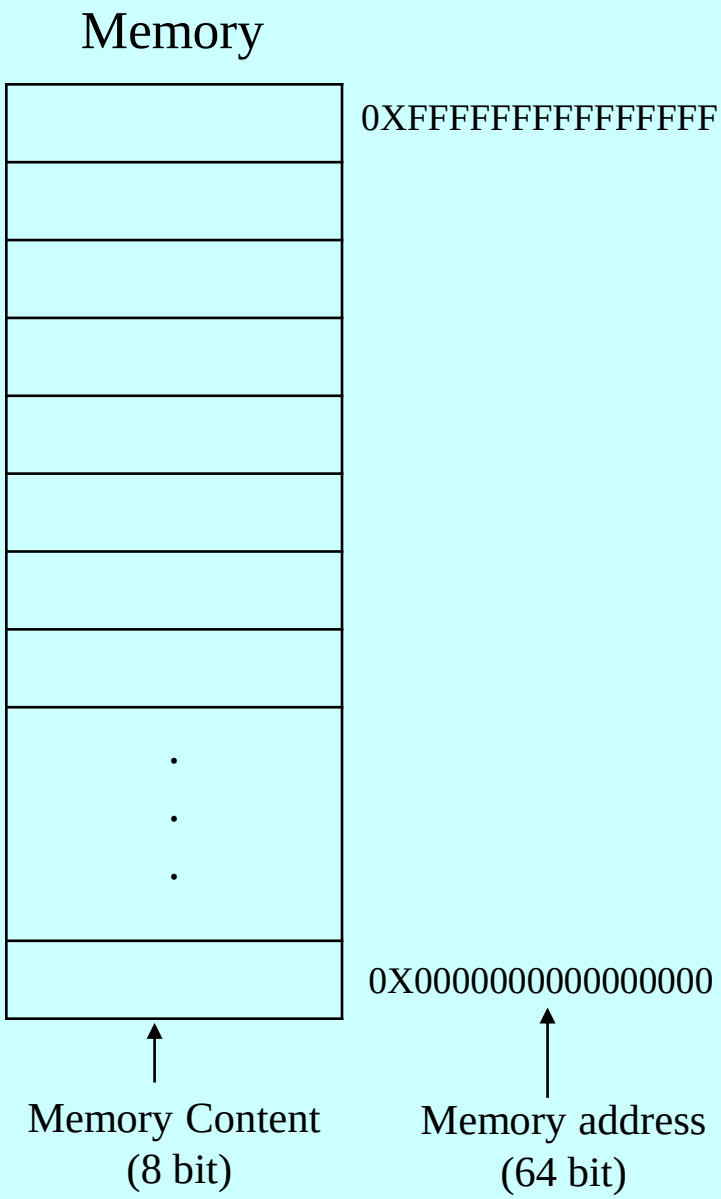
Problem: Assume that X2 = 24 (0X0000000000000018 in hexadecimal). What will be the value of X0 after running the following instruction: LSL X0, X2, #2

```
X2 →        00000000 00000000 00000000 00000000 00000000 00000000 00000000 00011000   24

1st shift→  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001100   12
2nd shift→  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000110   6
3rd shift→  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000011   3
```

The content of X2 will be 3 (0X0000000000000003 in hexadecimal)

Memory

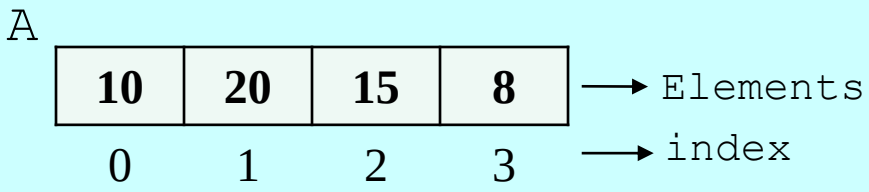


* 1 byte = 8 bit

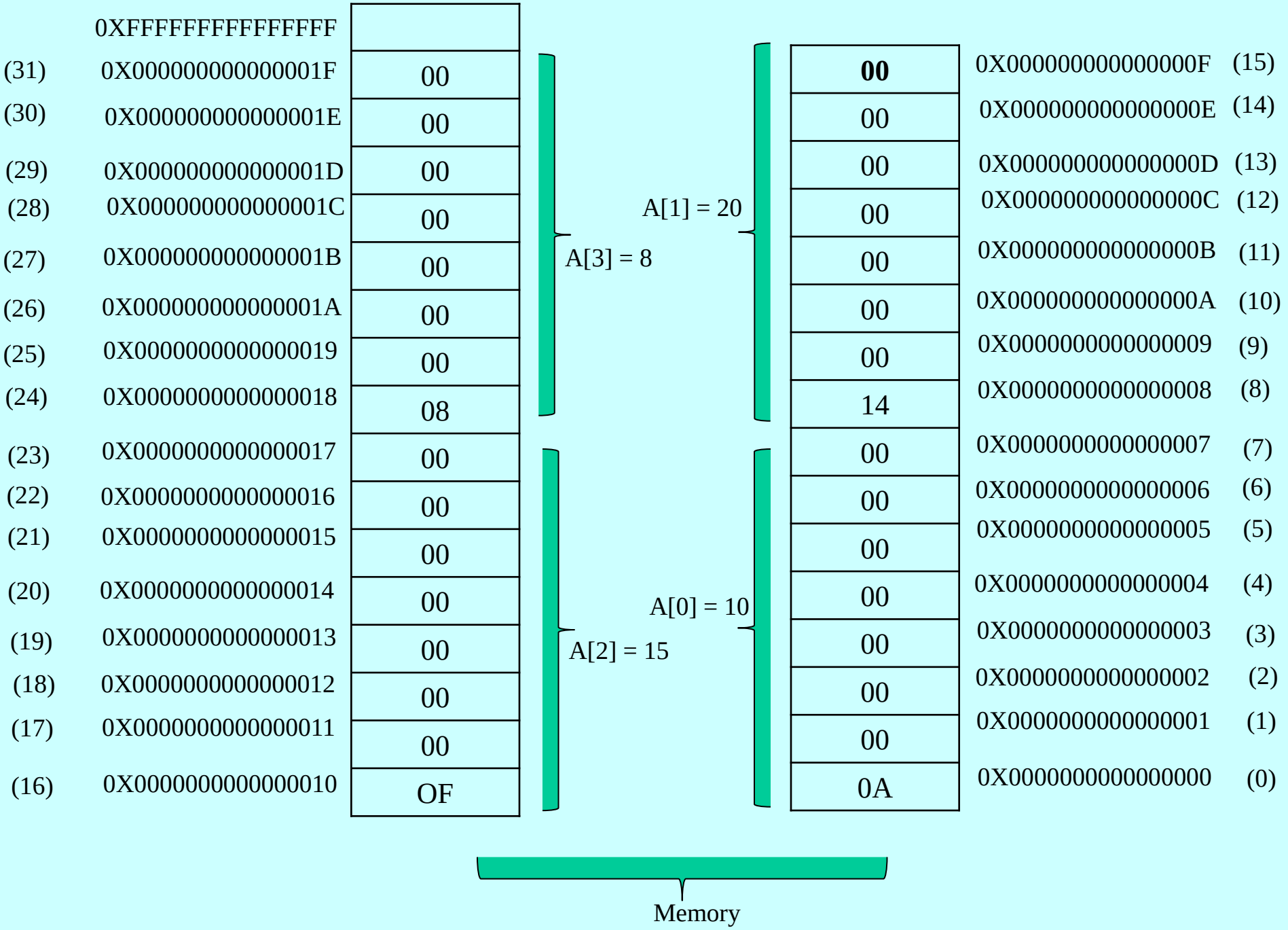
Problem: Assume that A is an array of 4 integer type elements (10, 20, 15, 8). Each element is 64 bit (doubleword). How does the array elements contained on the memory ? The base address of the array A is 0X0000000000000000.

SOLUTION:

int A = {10, 20, 15, 8};



A [0] = 10 = 0X000000000000000A
A [1] = 20 = 0X0000000000000014
A [2] = 15 = 0X000000000000000F
A [3] = 8 = 0X0000000000000008


0XFFFFFFFFFFFFFFFF
(31) 0X000000000000001F 00
(30) 0X000000000000001E 00
(29) 0X000000000000001D 00
(28) 0X000000000000001C 00
(27) 0X000000000000001B 00
(26) 0X000000000000001A 00
(25) 0X0000000000000019 00
(24) 0X0000000000000018 08
(23) 0X0000000000000017 00
(22) 0X0000000000000016 00
(21) 0X0000000000000015 00
(20) 0X0000000000000014 00
(19) 0X0000000000000013 00
(18) 0X0000000000000012 00
(17) 0X0000000000000011 00
(16) 0X0000000000000010 OF
A[3] = 8
A[2] = 15
A[1] = 20
A[0] = 10
00 0X000000000000000F (15)
00 0X000000000000000E (14)
00 0X000000000000000D (13)
00 0X000000000000000C (12)
00 0X000000000000000B (11)
00 0X000000000000000A (10)
00 0X0000000000000009 (9)
14 0X0000000000000008 (8)
00 0X0000000000000007 (7)
00 0X0000000000000006 (6)
00 0X0000000000000005 (5)
00 0X0000000000000004 (4)
00 0X0000000000000003 (3)
00 0X0000000000000002 (2)
00 0X0000000000000001 (1)
0A 0X0000000000000000 (0)
Memory

Address of A[0] = Base address + 0*8 = 0X0000000000000000

Address of A[1] = Base address + 1*8 = 0X0000000000000008

Address of A[2] = Base address + 2*8 = 0X0000000000000010

Address of A[3] = Base address + 3*8 = 0X0000000000000018

.

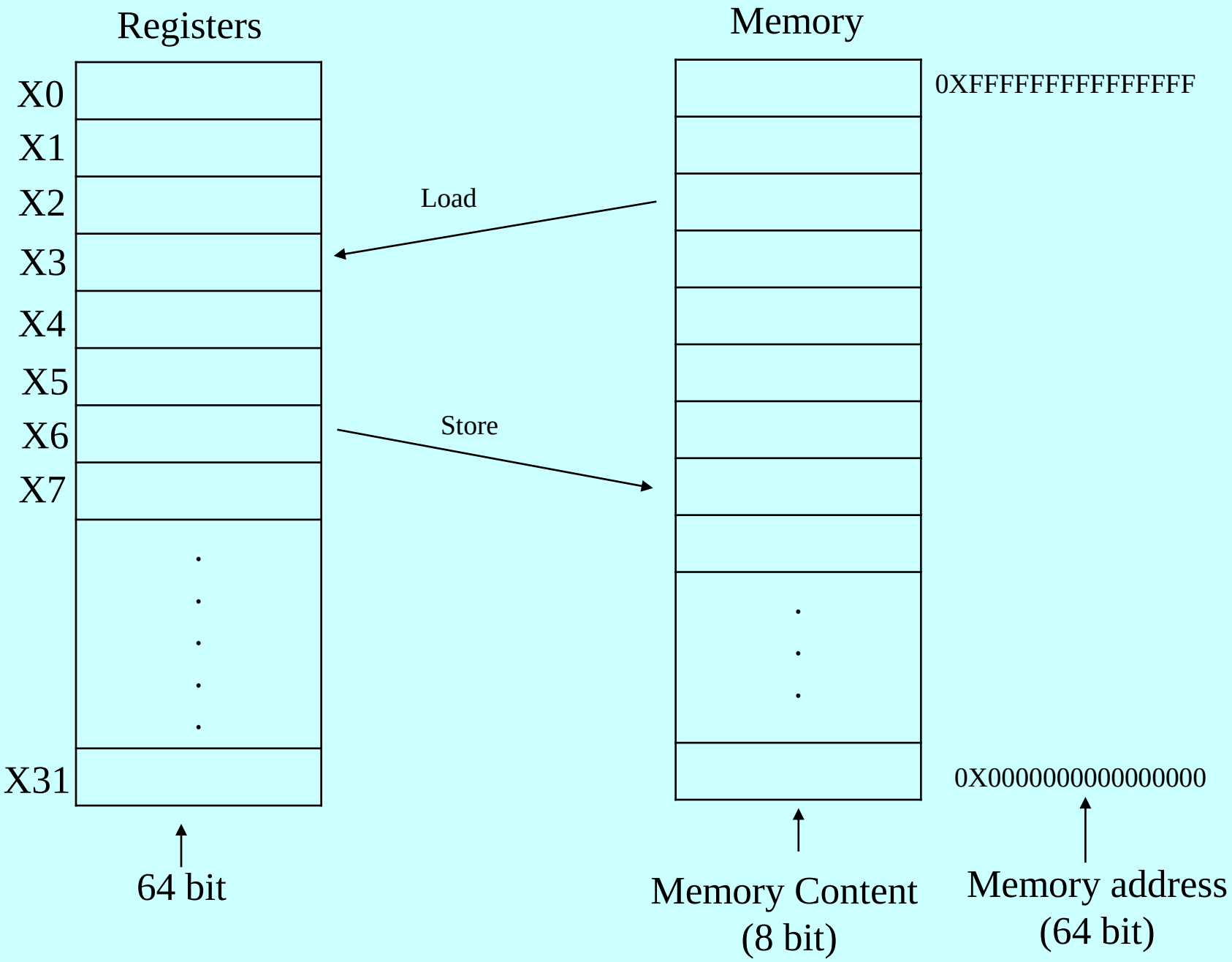
.

.

Address of A[i] = Base address + i*8

Load/Store Instruction

- The ARM CPU allows direct access to all locations in the memory, but they are done with specific instructions. Since these instructions either load the register with data from memory or store the data in the register to the memory, the are called load/store instructions.
- LDUR instruction tells the CPU to load one doubleword (64-bit or 8 bytes) of data from a memory location. Since each memory location can hold only one byte (ARM is byte addressable CPU), and the CPU registers are 64-bit width, the LDUR will bring in 8 bytes of data from 8 consecutive memory locations.


Registers
Memory
X0
X1
X2
X3
X4
X5
X6
X7
X31
Load
Store
0XFFFFFFFFFFFFFFFF
0X0000000000000000
64 bit
Memory Content
(8 bit)
Memory address
(64 bit)

Load Instruction

LDUR X2, [X5, #0]

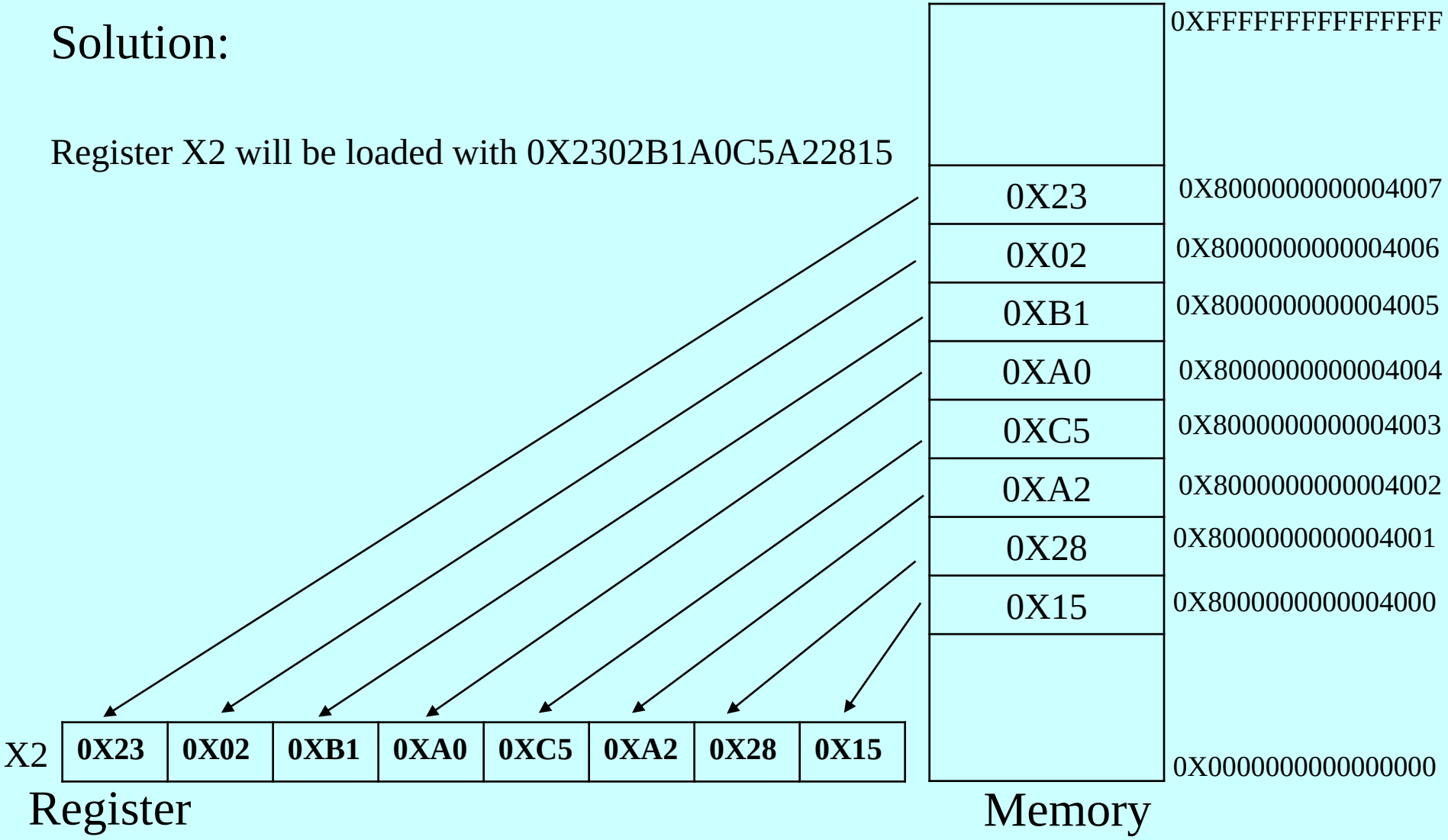
→ Load X2 with the content of memory pointed by X5+0

- LDUR X2, [X5, #0] instruction copies the content of the memory locations pointed by X5 + 0 into register X2. Since the X2 register is a 64-bit wide, it expects a 64-bit operand in the range of 0X0000000000000000 to 0XFFFFFFFFFFFFFFFF. That means the X5 register gives the base address of the memory in which it holds the data.
- Therefore, if X5=0X8000000000004000, the CPU will load register X2 with the contents of memory locations 0X8000000000004000 to 0X8000000000004007.

Problem: Assume that X5 = 0X8000000000004000 and locations 0X8000000000004000 through 0X8000000000004007 contain 0X15, 0X28, 0XA2, 0XC5, 0XA0, 0XB1, 0X02, and 0X23, respectively. What will happen to X2 after running the following instruction:
LDUR X2, [X5, #0].

Solution:

Register X2 will be loaded with 0X2302B1A0C5A22815



Compiling an assignment when an operand is on Memory

Convert the following C++ code to LEGv8 Assembly code. Assume A is an Array of 10 doublewords, variable g and h are associated with registers X20 and X21, and base address of the array A is in X22.

```
( 1 ) g = h + A [1];
( 2 ) g = h + A [8];
```

SOLUTION:

1 * 8 ↓

```
( 1 )   LDUR X9, [X22, #8]      // X9 = A [1]
        ADD X20, X9, X21        // g = h + A [1]
```

8 * 8 ↓

```
( 2 )   LDUR X9, [X22, #64]     // X9 = A [8]
        ADD X20, X9, X21        // g = h + A [8]
```

Store Instruction

- The STUR instruction tells the CPU to store (copy) the contents of CPU register to a memory location pointed by destination register. Notice that the source register of STUR instruction is placed before the destination register. Since the CPU registers are 64 – bit (8 byte) wide, we need 8 consecutive memory locations to store the contents of the register.

- See the following instruction:

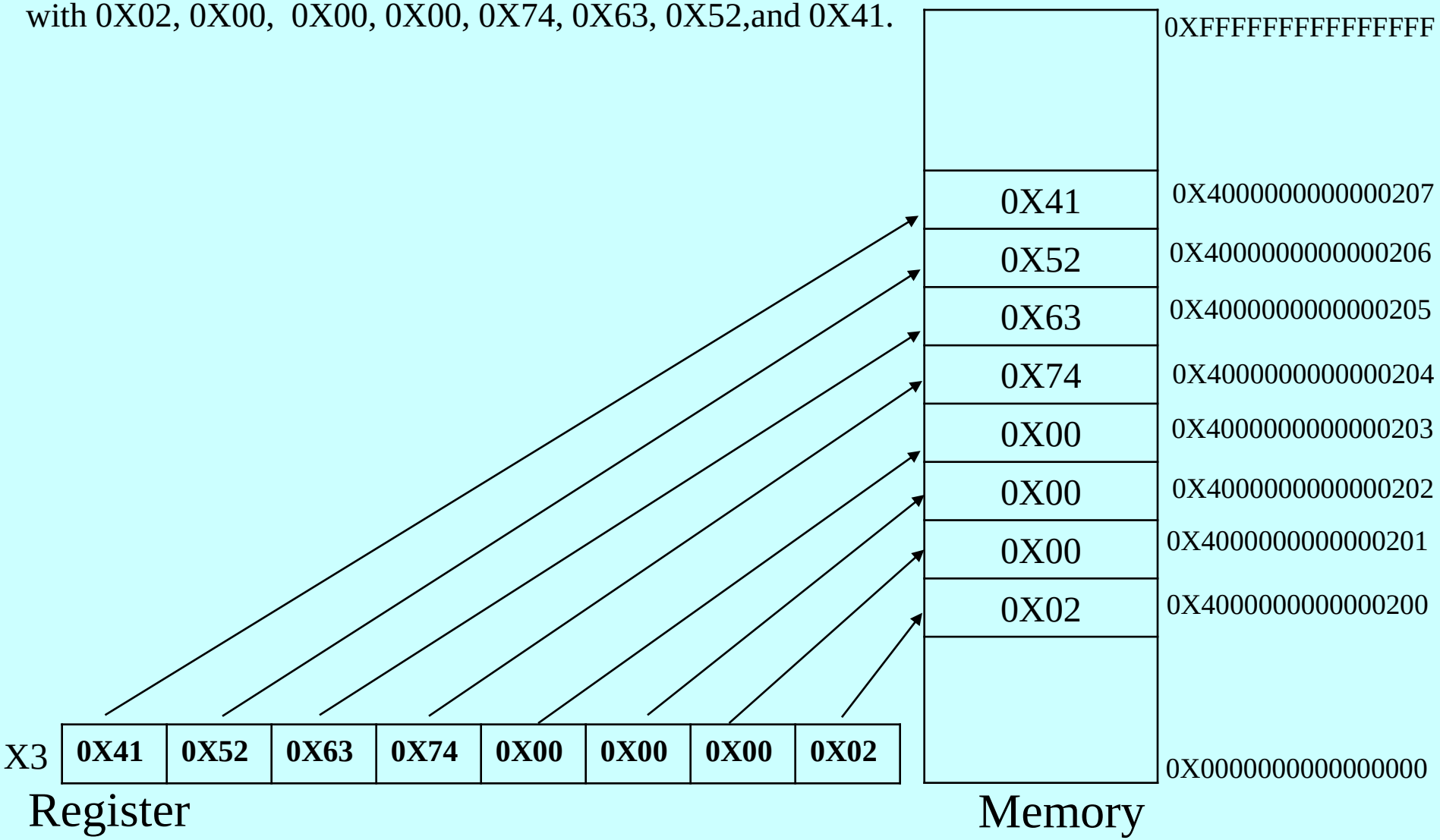
STUR X3, [X6, #0]

→ Store X3 into memory locations pointed to by X6 + 0.

- STUR X3, [X6, #0] instruction copies the content of X3 into locations pointed by X6 +0, the locations 0X4000000000000200 through 0X4000000000000207, we assumed X6 = 0X4000000000000200.

Problem: Assume that X6 = 0X4000000000000200 and X3 = 0X4152637400000002. What will happen after running the following instruction: STUR X3, [X6, #0].

Solution: Memory locations 0X4000000000000200 to 0X4000000000000207 will be loaded with 0X02, 0X00, 0X00, 0X00, 0X74, 0X63, 0X52,and 0X41.



Compiling an assignment when an operand is on Memory

Convert the following C++ code to LEGv8 Assembly code. Assume A is an Array of 10 doublewords, variable h is associated with register X21, and base address of the array A is in X22.

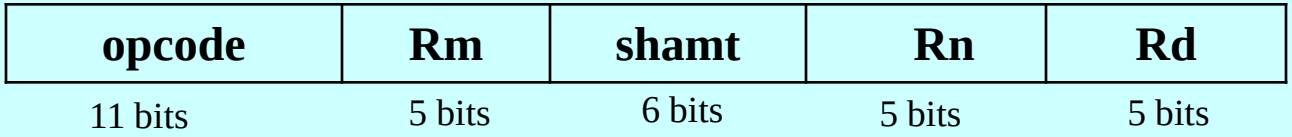
```
A [2] = h + A [1];
```

SOLUTION:

```
LDUR X9, [X22, #8]      // X9 = A [1]
ADD X9, X9, X21         // X9 = h + A [1]
STUR X9, [X22, #16]     // A [2] = h + A [1]
```

Representing Instructions in the computer (Translating a LEGV8 assembly instruction into a Machine instruction)

LEGV8 R-Format Instructions



Instruction fields

– opcode: operation code

– Rm: the second register source operand

– shamt: shift amount (00000 for now)

– Rn: the first register source operand

– Rd: the register destination

Instruction Format: The layout of an instruction is called the instruction format. LEGv8 instructions are 32 bit long. To distinguish it from assembly language, we call the numeric versions of instructions machine language and a sequence of such instructions are called machine code.

LEGv8 Instruction Encoding

Instruction	Format	opcode	Rm	shamt	address	op2	Rn	Rd
ADD (add)	R	1112_{ten}	reg	0	n.a.	n.a.	reg	reg
SUB (subtract)	R	1624_{ten}	reg	0	n.a.	n.a.	reg	reg
ADDI (add immediate)	I	580_{ten}	reg	n.a.	constant	n.a.	reg	n.a.
SUBI (sub immediate)	I	836_{ten}	reg	n.a.	constant	n.a.	reg	n.a.
LDUR (load word)	D	1986_{ten}	reg	n.a.	address	0	reg	n.a.
STUR (store word)	D	1984_{ten}	reg	n.a.	address	0	reg	n.a.

Translate the following LEGV8 assembly instruction into a machine instruction:
ADD X9, X20, X21

Instruction Format

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

Decimal Representation

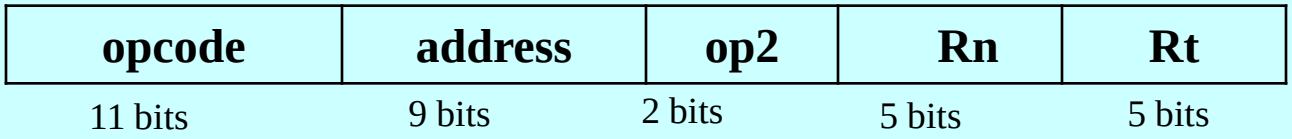
1112	21	0	20	9

Binary Representation

10001011000	10101	000000	10100	01001
11 bits	5 bits	6 bits	5 bits	5 bits

$10001011000\,10101\,000000\,10100\,01001_2 = 8B150289_{16}$

LEGv8 D-Format Instructions



Load/store instructions
– Rn: base register
– address: constant offset from contents of base register (+/- 32 double words)
– Rt: destination (load) or source (store) register number
– Example of D-type instructions are:

LDUR X9, [X10, #8]

Translate the following LEGV8 assembly instruction into a machine instruction:
LDUR X9, [X10, #8]

Instruction Format

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

Decimal Representation

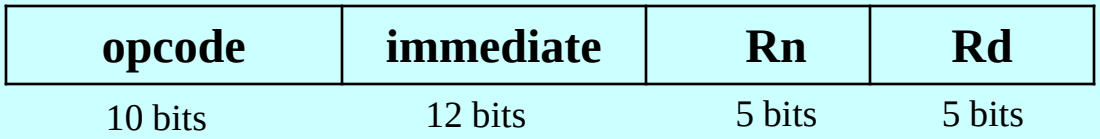
1986	8	0	10	9

Binary Representation

11111000010	000001000	00	01010	01001
11 bits	9 bits	2 bits	5 bits	5 bits

$11111000010000000100000101001001_2 = F8408149_{16}$

LEGv8 I-Format Instructions



Immediate instructions

– Rn: Source register

– Rd: destination register

– Example of immediate instructions are:

```
ADDI X9, X9, #1
SUBI  X10, X7, #1
```

Translate the following LEGV8 assembly instruction into a machine instruction:
ADDI X9, X9, #1

Instruction Format

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

Decimal Representation

580	1	9	9

Binary Representation

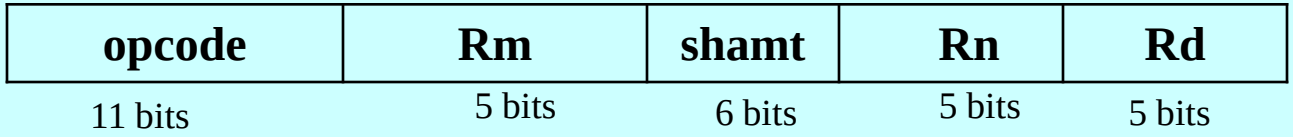
1001000100	000000000001	01001	01001
10 bits	12 bits	5 bits	5 bits

$$1001000100000000000001010010100 1_2 = 91000529_{16}$$

Logical Operations

Shift Operations

Instruction Format



Instruction fields

– opcode: operation code
– Rm: the second register source operand
– shamt: shift amount (How many position to shift)
– Rn: the first register source operand
– Rd: the register destination
– Example of shift instructions are:

LSL X11, X19, #4

Translate the following LEGV8 assembly instruction into a machine instruction:
LSL X11, X19, #4

Instruction Format

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

Decimal Representation

1691	0	4	19	11

Binary Representation

11010011011	00000	000100	10011	01011
	5 bits		5 bits	

$$11010011011000000010010011010112 = D360126B16$$

AND Operations

A logical bit-by-bit operation with two operands that calculates a 1only if there is a 1 in both operands.

For example, if X11 contains
00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

And register X10 contains
00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

AND X9, X10, X11

X10→ 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

X11→ 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

X9 → 00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

OR Operations

A logical bit-by-bit operation with two operands that calculates a 1only if there is a 1 in either operands.

For example, if X11 contains
00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

And register X10 contains
00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

OR X9, X10, X11

X10→ 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

X11→ 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

X9 → 00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

EOR Operations

A logical bit-by-bit operation with two operands that calculates the exclusive OR of the two operands. That is, it calculates a 1only if the values are different in the two operands.

For example, if X11 contains

00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

And register X10 contains

00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

EOR X9, X10, X11

X10→ 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

X11→ 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

X9 → 00000000 00000000 00000000 00000000 00000000 00000000 00110001 11000000

Branch Operations

Branch to a labeled instruction if a condition is true – Otherwise, continue sequentially

CBZ register, L1

- if (register == 0) branch to instruction labeled L1;
- Example: CBZ X9, Else

CBNZ register, L1

- if (register != 0) branch to instruction labeled L1;
- Example: CBNZ X9, Else

B L1

- branch unconditionally to instruction labeled L1;
- Example: B Exit

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
=	B.EQ	Z=1	B.EQ	Z=1
≠	B.NE	Z=0	B.NE	Z=0
<	B.LT	N!=V	B.LO	C=0
≤	B.LE	~(Z=0 & N=V)	B.LS	~(Z=0 & C=1)
>	B.GT	(Z=0 & N=V)	B.HI	(Z=0 & C=1)
≥	B.GE	N=V	B.HS	C=1

Assignment Statement

Convert the following C++ code to LEGv8 Assembly code. Assume the variables f, g, h, i, and j correspond to five registers X19, X20, X21, X22, and X23.

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

SOLUTION:

```
ADD X9, X20, X21        // Register X9 contains g+h
ADD X10, X22, X23       // Register X10 contains i+j
SUB X19, X9, X10        // f gets X9 – X10, which is (g + h) - (i + j)
```

N.B: Variables have been replaced with register names plus two temporary register X9 and X10.The words to the right of the double slashes (//) on each line above are comments for the human reader, so the computer ignores them.

If statement

Convert the following C++ code to LEGv8 Assembly code. Assume the variables a and b correspond to registers X22 and X23.

```
if (a>b)
{
    a=a+1;
}
```

SOLUTION:

```
        CMP X22, X23          // Use sub to make comparison
        B.LE Exit             // Conditional branch
        ADDI X22, X22, #1     //  a++
Exit:
```

If statement (Alternate way)

Convert the following C++ code to LEGv8 Assembly code. Assume the variables a and b correspond to registers X22 and X23.

```
if (a>b)
{
    a=a+1;
}
```

SOLUTION:

```
        CMP X22, X23          // Use sub to make comparison
        B.GT L1               // Conditional branch
        B Exit
L1:     ADDI X22, X22, #1     //  a++
Exit:
```

if-else Statement

Convert the following C++ code to LEGv8 Assembly code. Assume the variables f, g, h, i, and j correspond to five registers X19, X20, X21, X22, and X23.

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

SOLUTION:

```
        SUB X9,X22,X23          // X9 = i - j
        CBNZ X9,Else            // go to Else if X9 is not equal to zero
        ADD X19,X20,X21         // f = g + h
        B Exit                  // go to exit
Else:   SUB X19,X20,x21         // f = g - h
Exit:
```

if-else Statement (alternate way)

Convert the following C++ code to LEGv8 Assembly code. Assume the variables f, g, h, i, and j correspond to five registers X19, X20, X21, X22, and X23.

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

SOLUTION:

```
    SUB X9,X22,X23          // X9 = i - j
    CBZ X9, L1              // go to L1 if X9 is equal to zero
    SUB X19,X20,x21         // f = g - h
    B Exit                  // go to exit
L1:ADD X19,X20,X21          // f = g + h
Exit:
```

While loop

Convert the following C++ code to LEGv8 Assembly code. Assume the variables i and k correspond to registers X22 and X24.

```
while (i==k)
{
    i=i+1;
}
```

SOLUTION:

```
Loop:  SUB X11, X22,X24        //  check i == k
       CBNZ X11, Exit          //  If false, exit
       ADDI X22, X22, #1       // else i++
       B Loop                  // Loop back
 Exit:
```

Convert the following C++ code to LEGv8 Assembly code. Assume the variables i and k correspond to registers X22 and X24. The base address of the array `save` is in X25.

```
while (save[i])==k)
{
    i=i+1;
}
```

SOLUTION:

```
Loop:  LSL X10, X22, #3          // Multiply i*8
       ADD X10, X10, X25         // address of save[i]
       LDUR X9, [X10, #0] ;      // Load save[i]
       SUB X11, X9,X24           //  check save[i] == k
       CBNZ X11, Exit            //  If false, exit
       ADDI X22, X22, #1         // else i++
       B Loop                    // Loop back
 Exit:
```

for Loop

Convert the following C++ code to LEGv8 Assembly code. Assume the variable a is in X22 and base address of array b is in X23.

```
for(i=0,i<a,i++)
{
    b[i] = a + i;
}
```

SOLUTION:

```
        ADDI X9, XZR, #0        // Assuming i is in temp register X9
Loop:   CMP X9, X22             // Check for i < a
        B.GE Exit               // If false exit else continue
        ADD X10, X22, X9        // Compute a + i
        LSL X11, X9, #3         // Multiply i*8 for 64-bits
        ADD X11, X23, X11       // Compute address for b[i]
        STUR X10, [X11, #0]     // Store the a + i in b[i]
        ADDI X9, X9, #1         // i++
        B Loop                  // Loop back
Exit:
```

Write a C++ program to add two given integer numbers using user defined function.

```
#include <iostream>
using namespace std;

int add (int x, int y); // function declaration
int main() // main code
{
   int a=5, b=4, c;
   c = add(a, b); // function call
   cout<<"Result:"<<c;
   return 0;
}

int add (int x, int y) // function definition
{
   return (x + y);
}
```

Output Terminal

```
Result:9
```

Write a LEGv8 assembly code to add two given integer numbers using procedure (function). Assume that the integer numbers are stored in register X19 and X20, respectively. Store the result on register X21. Also, assume all values are 64-bits. . cin and cout function are not required to be converted into assembly. Comment your assembly code.

CONTINUE

Note: The figures, text etc included in slides are borrowed from books, other sources for academic purpose only. The author does not claim any originality.

Source:

1.Computer Organization and Design (ARM Edition) by David A. Patterson