Computer Organization and Architecture

COSC 2425

Lecture – 7

Sept 12th, 2022

Acknowledgement: Slides from Edgar Gabriel & Kevin Long

Chapter 2

Instructions: Language of the Computer

Review

Number System

1. Decimal System
    1. Integers
    2. Fractions
    3. Positional number system
2. Binary representation
    1. Integers
    2. Fractions
    3. Addition
3. Conversion
    1. Binary to Decimal
    2. Decimal to Binary
4. Hexadecimal
5. Signed Integers
    1. 2’s complement representation

Review

Signed integers

- We have been dealing so far with unsigned integers

Slide based on a lecture at: http://people.sju.edu/~ggrevera/arch/slides/binary-arithmetic.ppt

Review

Signed integers

- We have been dealing so far with unsigned integers

- Multiple ways for representing signed integers:
  1. Sign and magnitude
  2. 2's complement

Slide based on a lecture at: http://people.sju.edu/~ggrevera/arch/slides/binary-arithmetic.ppt

Review

Sign and Magnitude

- Lets consider three bits to represent numbers.
- Number from 0 – 7 can be represented
- Sign and magnitude: Uses one additional bit to represent positive/negative, called sign bit.
- 0 ➔ positive number
- 1 ➔ negative numbers

0	000	+0
0	001	+1
0	010	+2
0	011	+3
0	100	+4
0	101	+5
0	110	+6
0	111	+7

1	000	-0
1	001	-1
1	010	-2
1	011	-3
1	100	-4
1	101	-5
1	110	-6
1	111	-7

Review

Sign and Magnitude

- Sign and magnitude: Uses one additional bit to represent positive/negative, called sign bit.
- Shortcomings:
  - Where to put the sign bit (left/right)
  - Adders may need extra step to set the sign bit
  - Both a positive and negative zero

0	000	+0	1	000	-0
0	001	+1	1	001	-1
0	010	+2	1	010	-2
0	011	+3	1	011	-3
0	100	+4	1	100	-4
0	101	+5	1	101	-5
0	110	+6	1	110	-6
0	111	+7	1	111	-7

Review

2's Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
- Split in to two halves
  - First half ➔ Positive (same as before)
  - Second half ➔ Negative (declining order)
  - Range -8, -7 … 6, 7

Most negative number

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Review

2's Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
- Split in to two halves
  - First half ➔ Positive (same as before)
  - Second half ➔ Negative (declining order)
- Many advantages:
  - Leading 0 ➔ Positive, Leading 1 ➔ Negative
  - Test only one bit to check positive/negative
  - Made hardware implementation simple

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Review

2’s Complement

- Conversion to decimal is straight forward

1	0	1	1
-2^3	2^2	2^1	2^0

$$= 1X\left(-2^3\right) + 0X\left(2^2\right) + 1\,X\,(2^1) + 1\,X\,(2^0)$$
$$= -8 + 0 + 2 + 1 = -5$$

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Review

Shortcut to Negate

- Determine the binary value of -27 in 2's complement representation using 8 bits

+27 in binary is: 0001 1011

Bitwise complement: 1110 0100
Add 1: + 1
\-\-\-\-\-\-\-\-\-\-\-\-\-
1110 0101

2's complement for -27

Verify:

$$= 1\ X\ (-2^7) + 1X2^6 + 1\ X\ 2^5 + 0\ X2^4 + 0\ X\ 2^3 + 1\ X\ 2^2 + 0\ X\ 2^1 + 1X\ 2^0$$

$$= -128 + 64 + 32 + 0 + 0 + 4 + 0 + 1$$
$$= -27$$

Review

Sign Extension

- Example: show the representation of +4 and -4 for 4 bits and 8 bits

+4: 0100

0000 0100

-4: 1100

1111 1100

LEGv8: Signed and Unsigned Numbers

- LEGv8: 64 bit double word representation.

- Example Representation: $11_{ten} = 1101_{two}$

00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000**1101**

63 ← → 0

Most significant bit

Least significant bit

LEGv8: Unsigned Numbers

- LEGv8: 64 bit double word representation.
  - Can represent 2^{64} different patterns.
- Numbers range from $[0, 2^{64} - 1](18{,}446{,}774{,}073{,}709{,}551{,}615_{ten})$

LEGv8: Unsigned Numbers

- LEGv8: 64 bit double word representation.
  - Can represent 2^{64} different patterns.
- Numbers range from $[0, 2^{64} - 1](18{,}446{,}774{,}073{,}709{,}551{,}615_{ten})$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = 0_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$

. . .   . . .

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = 18{,}446{,}774{,}073{,}709{,}551{,}613_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = 18{,}446{,}744{,}073{,}709{,}551{,}614_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = 18{,}446{,}744{,}073{,}709{,}551{,}615_{ten}$

LEGv8: Unsigned Numbers

- LEGv8: 64 bit double word representation.
  - Can represent 2^{64} different patterns.
- Numbers range from $[0, 2^{64} - 1](18{,}446{,}774{,}073{,}709{,}551{,}615_{ten})$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = 0_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$

. . .   . . .

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = 18{,}446{,}774{,}073{,}709{,}551{,}613_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = 18{,}446{,}744{,}073{,}709{,}551{,}614_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = 18{,}446{,}744{,}073{,}709{,}551{,}615_{ten}$

$$(x_{63} \times 2^{63}) + (x_{62} \times 2^{62}) + (x_{61} \times 2^{61}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

LEGv8: Unsigned Numbers

- Add, subtract, multiply these binary bit patterns.

$$
\begin{array}{r}
11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} \\
+ \\
00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} \\
\hline
\end{array}
$$

An overflow occurs

- Programming languages, OS, program can decide how to handle.

Review

2's Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
- Split in to two halves
  - First half ➔ Positive (same as before)
  - Second half ➔ Negative (declining order)
  - Range -8, -7 … 6, 7

Most negative number

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

LEGv8: Signed Numbers

- LEGv8: 64 bit double word **2's complement** representation.

Positive

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = 0_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$

. . . . . .

$01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = 9{,}223{,}372{,}036{,}854{,}775{,}805_{ten}$

$01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = 9{,}223{,}372{,}036{,}854{,}775{,}806_{ten}$

$01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = 9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}$

Negative

$10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = -\ 9{,}223{,}372{,}036{,}854{,}775{,}808_{ten}$

$10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = -\ 9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}$

$10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = -\ 9{,}223{,}372{,}036{,}854{,}775{,}806_{ten}$

… . . .

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = -\ 3_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = -\ 2_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = -\ 1_{ten}$

LEGv8: Signed Numbers

- LEGv8: 64 bit double word **2's complement** representation.

Positive
Have **0** in most significant bit

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = 0_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$

. . . . . .

$01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = 9{,}223{,}372{,}036{,}854{,}775{,}805_{ten}$

$01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = 9{,}223{,}372{,}036{,}854{,}775{,}806_{ten}$

$01111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = 9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}$

Sign bit

$10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = -\ 9{,}223{,}372{,}036{,}854{,}775{,}808_{ten}$

$10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = -\ 9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}$

$10000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = -\ 9{,}223{,}372{,}036{,}854{,}775{,}806_{ten}$

… . . .

Negative
Have **1** in most significant bit

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = -\ 3_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = -\ 2_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = -\ 1_{ten}$

LEGv8: Signed Numbers

- LEGv8: 64 bit double word **2’s complement** representation.
- Positive half range:
  - $[0\ to\ 9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}]$
- Negative half
  - $[-1\ to\ -9{,}223{,}372{,}036{,}854{,}775{,}808_{ten}]$

LEGv8: Signed Numbers

- LEGv8: 64 bit double word **2's complement** representation.
- Positive half range:
  - $[0\ to\ 9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}]$
- Negative half
  - $[-1\ to\ -\mathbf{9,223,372,036,854,775,808}_{\boldsymbol{ten}}]$

Most negative number

Binary to Decimal Conversion

Position weights for conversion in 2’s complement representation.

?	2^{62}							2^{1}	2^{0}

Binary to Decimal Conversion

Position weights for conversion in 2's complement representation.

$\mathbf{-2^{63}}$	2^{62}							2^1	2^0

$$(x_{63} \times -2^{63}) + (x_{62} \times 2^{62}) + (x_{61} \times 2^{61}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Example

What is the decimal value of this 64-bit two's complement number?

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111100_{two}$

Example

What is the decimal value of this 64-bit two's complement number?

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111100_{two}$$

Substituting the number's bit values into the formula above:

$$(1 \times -2^{63}) + (1 \times 2^{62}) + (1 \times 2^{61}) + \cdots + (1 \times 2^{1}) + (0 \times 2^{1}) + (0 \times 2^{0})$$

Example

What is the decimal value of this 64-bit two's complement number?

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111100_{two}$$

Substituting the number's bit values into the formula above:

$$(1 \times -2^{63}) + (1 \times 2^{62}) + (1 \times 2^{61}) + \cdots + (1 \times 2^{1}) + (0 \times 2^{1}) + (0 \times 2^{0})$$
$$= -2^{63} + 2^{62} + 2^{61} + \cdots + 2^{2} + 0 + 0$$
$$= -9{,}223{,}372{,}036{,}854{,}775{,}808_{ten} + 9{,}223{,}372{,}036{,}854{,}775{,}804_{ten}$$
$$= -4_{ten}$$

Example

What is the decimal value of this 64-bit two's complement number?

$$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111100_{\text{two}}$$

Substituting the number's bit values into the formula above:

$$(1 \times -2^{63}) + (1 \times 2^{62}) + (1 \times 2^{61}) + \cdots + (1 \times 2^{1}) + (0 \times 2^{1}) + (0 \times 2^{0})$$

$$= -2^{63} + 2^{62} + 2^{61} + \cdots + 2^{2} + 0 + 0$$

$$= -9{,}223{,}372{,}036{,}854{,}775{,}808_{\text{ten}} + 9{,}223{,}372{,}036{,}854{,}775{,}804_{\text{ten}}$$

$$= -4_{\text{ten}}$$

Shortcut to Negate

Negate 2_{ten}, and then check the result by negating -2_{ten}.

$$2_{ten} = 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two}$$

Shortcut to Negate

Negate 2_{ten}, and then check the result by negating -2_{ten}.

$$2_{ten} = 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two}$$

Negating this number by inverting the bits and adding one,

$$\begin{array}{lr} & 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} \\ + & 1_{two} \\ \hline = & 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} \\ = & -2_{ten} \end{array}$$

Sign Extension

- Negate and extend to 64 bit

$0000\ 0000\ 0000\ 0010_{two}$

Sign Extension

- Negate and extend to 64 bit

$$0000\ \ 0000\ \ 0000\ \ 0010_{two}$$

becomes

$$
\begin{array}{l r}
 & 1111\ 1111\ 1111\ 1101_{two} \\
+ & 1_{two} \\
\hline
= & 1111\ 1111\ 1111\ 1110_{two}
\end{array}
$$

Sign Extension

- Negate and extend to 64 bit

$0000\ 0000\ 0000\ 0010_{two}$

becomes

$$
\begin{array}{r}
1111\ 1111\ 1111\ 1101_{two} \\
+ \qquad\qquad\qquad\qquad 1_{two} \\
\hline
\end{array}
$$

$= 1111\ 1111\ 1111\ 1110_{two}$

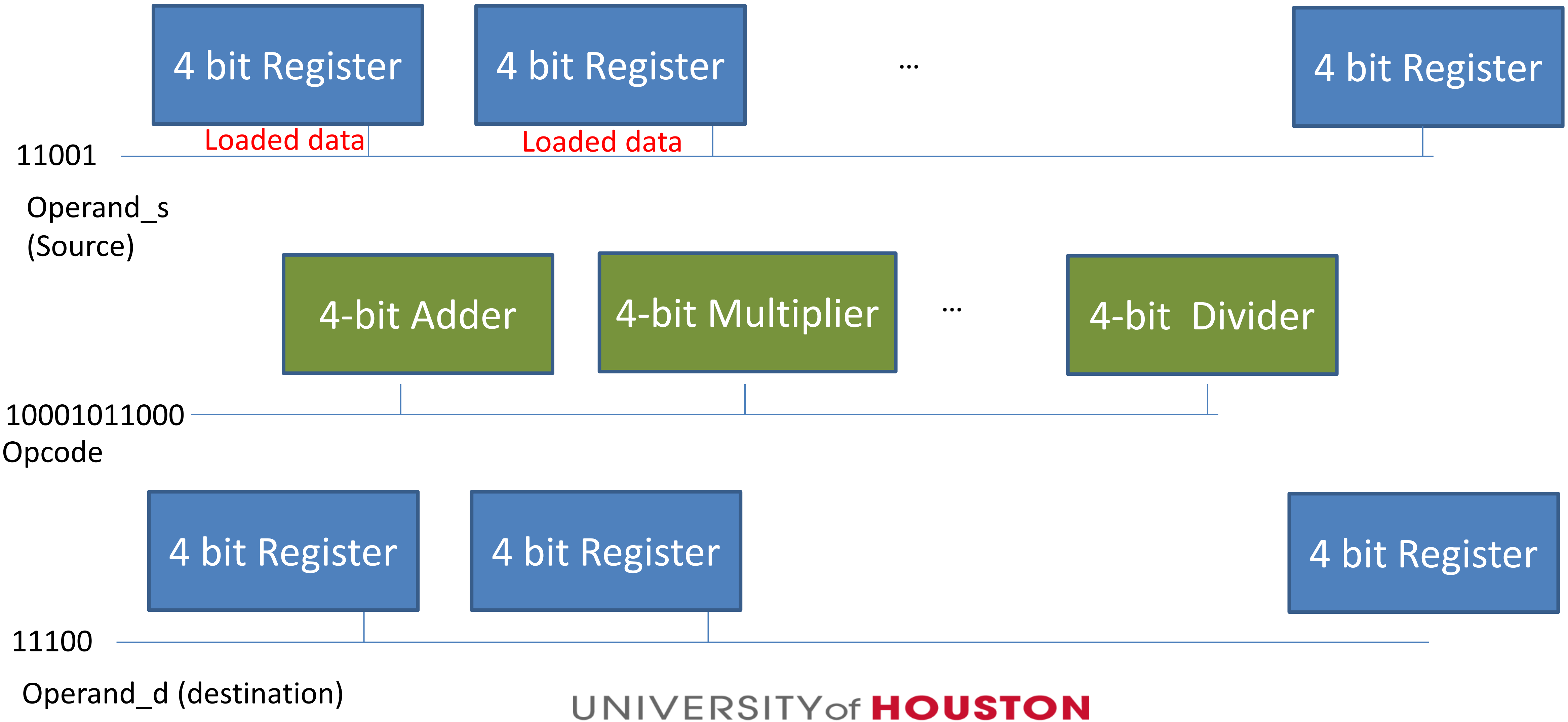
Creating a 64-bit version of the negative number means copying the sign bit 48 times and placing it on the left:

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = -2_{ten}$

Representing Instructions

Review Slide

Instruction



Review Slide

Instruction Example

Opcode	Operand_s1	Operand_s2	Operand_d
10001011000	11001	11010	11100

Instruction : 10001011000 11001 11010 11100

Instruction are represented in binary form. Stored in memory.
The only language a computer understand.
Byte code, machine code, …

What is the format for LEGv8?

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- **LEGv8** instructions
  - Encoded as **32-bit instruction words**

Instructions

Arithmetic	ADD, SUB, MUL
Data transfer	LDUR, STUR
Arithmetic Immediate	ADDI, SUBI

Representing Instructions

- Instructions are encoded in binary
  - Called machine code

- **LEGv8** instructions
  - Encoded as **32-bit instruction words**
  - **Different formats exists (but a small number)**
    - **R-Type →Arithmetic**
    - **D-Type → Data transfer**
    - **I-Type → Immediate**
    - **…**
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

R-format Example

```
ADD X9,X20,X21
```

R-format Example

`ADD X9,X20,X21`

1112_{ten}

opcode

opcode : operation code

R-format Example

`ADD X9,X20,X21`

1112_{ten}	21_{ten}
opcode	Rm

opcode : operation code
Rm: the second **register source** operand

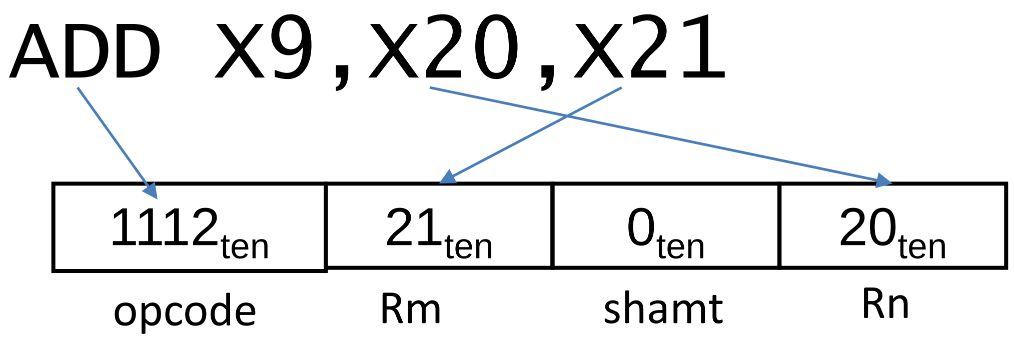
R-format Example

`ADD X9,X20,X21`

1112_{ten}	21_{ten}	0_{ten}
opcode	Rm	shamt

opcode : operation code
Rm: the second **register source** operand
shamt: shift amount (00000 for now)

R-format Example



opcode : operation code
Rm: the second **register source** operand
shamt: shift amount (00000 for now)
Rn: the first **register source** operand

R-format Example

`ADD X9,X20,X21`

1112_{ten}	21_{ten}	0_{ten}	20_{ten}	9_{ten}
opcode	Rm	shamt	Rn	Rd

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**

R-format Example

`ADD X9,X20,X21`

1112_{ten}	21_{ten}	0_{ten}	20_{ten}	9_{ten}	
opcode	Rm	shamt	Rn	Rd	
10001011000_{two}	10101_{two}	000000_{two}	10100_{two}	01001_{two}	In Binary

R-format Example

`ADD X9,X20,X21`

1112_{ten}	21_{ten}	0_{ten}	20_{ten}	9_{ten}

10001011000_{two}	10101_{two}	000000_{two}	10100_{two}	01001_{two}

$1000\ 1011\ 0001\ 0101\ 0000\ 0010\ 1000\ 1001_{two}$

LEGv8 R-format Instructions

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

Why 5 bits?

- 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

LEGv8 R-format Instructions

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

$2^5=32$ i.e 5 bits required to distinguish 32 Registers

- 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

R-format Example

ADD `X9,X20,X21`

1112_{ten}	21_{ten}	0_{ten}	20_{ten}	9_{ten}
10001011000_{two}	10101_{two}	000000_{two}	10100_{two}	01001_{two}

$$\underbrace{1000\ 1011\ 0001\ 0101\ 0000\ 0010\ 1000\ 1001_{two}}$$

Tedious use higher base. Hexadecimal representation

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

R-format Example

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

ADD X9,X20,X21

1112_{ten}	21_{ten}	0_{ten}	20_{ten}	9_{ten}

10001011000_{two}	10101_{two}	000000_{two}	10100_{two}	01001_{two}

$1000\ 1011\ 0001\ 0101\ 0000\ 0010\ 1000\ 1001_{two}$ =

$8B150289_{16}$

Can we use R-Type for LDUR instruction

$LDUR\ X19, [X22, \#\boldsymbol{const}]$

Could use to specify
source & destination

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

Can we use R-Type for LDUR instruction?

$LDUR\ X19, [X22, \#\boldsymbol{const}]$

Could use to specify source & destination

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

If we user Rm (5 bits),
#Const value cannot greater that 31
Arrays and data structures, usually need much larger values.

Different format for Data Transfer (D-Type)

- ***Design Principle 3:*** Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Instruction Set Design Principals

Design Principle 1: Simplicity favors regularity

Design Principle 2: Smaller is faster

Design Principle 3: Good design demands good compromises

LEGv8 D-format Instructions

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

- Load/store instructions
  - Rn: base register
  - address: constant offset from contents of base register (+/- 32 doublewords)
  - Rt: destination (load) or source (store) register number

Example: D-Type

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

$$LDUR\ X9, [X22, \#\mathbf{64}]$$

1986		0		
11 bits	**9 bits**	2 bits	5 bits	5 bits

Example: D-Type

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

$$LDUR\ X9, [X22, \mathbf{\#64}]$$

1986		0		9
11 bits	**9 bits**	2 bits	5 bits	5 bits

Example: D-Type

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

$$LDUR\ X9, [X22, \mathbf{\#64}]$$

1986		0	22	9
11 bits	**9 bits**	2 bits	5 bits	5 bits

Example: D-Type

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

$$LDUR\ X9, [X22, \#\mathbf{64}]$$

1986	**64**	0	22	9
11 bits	**9 bits**	2 bits	5 bits	5 bits

Can we use D-Type to represent ADDI (Immediate)?

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

Can we use D-Type to represent ADDI (Immediate)

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

Can use to represent constant values.
But the developers decided to include a different format with 12 bits for immediate value, allowing the use of larger numbers. (**I-Type**)

LEGv8 I-format Instructions

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

- Immediate instructions
  - Rn: source register
  - Rd: destination register
- Immediate field is zero-extended

Example: I-Type

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

$$ADDI\ X9, X22, \#35$$

580			
10 bits	12 bits	5 bits	5 bits

Example: I-Type

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

$$ADDI\ X9, X22, \#35$$

580		22	9
10 bits	12 bits	5 bits	5 bits

Example: I-Type

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

$$ADDI\ X9, X22, \#35$$

580	35	22	9
10 bits	12 bits	5 bits	5 bits

Opcodes

- Formats distinguished using opcodes

Instruction	Format	opcode
ADD (add)	R	1112_{ten}
SUB (subtract)	R	1624_{ten}
ADDI (add immediate)	I	580_{ten}
SUBI (sub immediate)	I	836_{ten}
LDUR (load word)	D	1986_{ten}
STUR (store word)	D	1984_{ten}

Example

$$A[30] = h + A[30] + 1$$

Base address of A stored in X10, h stored in X21

LEGv8 Assembly code:

Example

$$A[30] = h + A[30] + 1$$

Base address of A stored in X10, h stored in X21

LEGv8 Assembly code:

LDUR X9, [X10, #240]
ADD X9, X21, X9
ADDI X9 ,X9, #1
STUR X9, [X10, #240]

Example

$$A[30] = h + A[30] + 1$$

Base address of A stored in X10, h stored in X21

LEGv8 Assembly code:

LDUR X9, [X10, #240]
ADD X9, X21, X9
ADDI X9 ,X9, #1
STUR X9, [X10, #240]

D-Type

Machine Language in Decimal:

opcode	Rm/address	shamt/op2	Rn	Rd/Rt
1986	240	0	10	9

Example

$$A[30] = h + A[30] + 1$$

Base address of A stored in X10, h stored in X21

LEGv8 Assembly code:

LDUR X9, [X10, #240]
ADD X9, X21, X9
ADDI X9 ,X9, #1
STUR X9, [X10, #240]

R-Type

Machine Language in Decimal:

opcode	Rm/address	shamt/op2	Rn	Rd/Rt
1986	240	0	10	9
1112	9	0	21	9

Example

$$A[30] = h + A[30] + 1$$

Base address of A stored in X10, h stored in X21

LEGv8 Assembly code:

LDUR X9, [X10, #240]
ADD X9, X21, X9
ADDI X9 ,X9, #1
STUR X9, [X10, #240]

I-Type

Machine Language in Decimal:

opcode	Rm/address	shamt/op2	Rn	Rd/Rt
1986	240	0	10	9
1112	9	0	21	9
580	1		9	9

Example

$$A[30] = h + A[30] + 1$$

Base address of A stored in X10, h stored in X21

LEGv8 Assembly code:

LDUR X9, [X10, #240]
ADD X9, X21, X9
ADDI X9 ,X9, #1
STUR X9, [X10, #240]

D-Type

Machine Language in Decimal:

opcode	Rm/address	shamt/op2	Rn	Rd/Rt
1986	240	0	10	9
1112	9	0	21	9
580	1		9	9
1984	240	0	10	9

Example

$$A[30] = h + A[30] + 1$$

opcode	Rm/address	shamt/op2	Rn	Rd/Rt
1986	240	0	10	9
1112	9	0	21	9
580	1		9	9
1984	240	0	10	9

11111000010	011110000	00	01010	01001
10001011000	01001	000000	10101	01001
1001000100	000000000001		01001	01001
11111000000	011110000	00	01010	01001

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	LEGv8
Shift left	<<	<<	`LSL`
Shift right	>>	>>	`LSR`
Bit-by-bit AND	&	&	`AND, ANDI`
Bit-by-bit OR	|	|	`OR, ORI`
Bit-by-bit NOT	~	~	`EOR, EORI`

- Operate on bits/bytes more useful than on words
  - Examine characters (8 bits) within a word
- Useful for extracting and inserting groups of bits in a word

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	LEGv8
Shift left	<<	<<	LSL
Shift right	>>	>>	LSR
Bit-by-bit AND	&	&	AND, ANDI
Bit-by-bit OR	|	|	OR, ORI
Bit-by-bit NOT	~	~	EOR, EORI

Shifts

- Operate on bits/bytes more useful than on words
  - Examine characters (8 bits) within a word
- Useful for extracting and inserting groups of bits in a word

Shift Operations

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

- What format?

Shift Operations

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

- Use R- format
- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - LSL Logical shift left
- Shift right logical
  - Shift right and fill with 0 bits
  - LSR Logical shift right

Example LSL

$$LSL\ X11, X19, \#4 // \ shift\ 4\ bits\ to\ left$$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001001_{two} = 9_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 1001\underbrace{0000}_{}{}_{two} = 144_{ten}$

Example LSL

$$LSL\ X11, X19, \#4 // \ shift\ 4\ bits\ to\ left$$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001001_{two} = 9_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 10010000_{two} = 144_{ten}$

$$144_{ten} = 9_{ten} * 2^4$$

Left Shift by *i* bits multiplies by 2^i

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	LEGv8
Shift left	<<	<<	LSL
Shift right	>>	>>	LSR
Bit-by-bit AND	&	&	AND, ANDI
Bit-by-bit OR	|	|	OR, ORI
Bit-by-bit NOT	~	~	EOR, EORI

- Operate on bits/bytes more useful than on words
  - Examine characters (8 bits) within a word
- Useful for extracting and inserting groups of bits in a word

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

AND X9,X10,X11

X10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
X11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
X9	00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
ORR X9,X10,X11
```

X10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
X11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
X9	00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

EOR Operations

- Exclusive OR instead of NOT
- Differencing operation
  - Set some bits to 1, leave others unchanged

```
EOR X9,X10,X12   // NOT operation
```

X10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
X12	11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
X9	11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111

Usage Example

- Set a flags to indicate certain features of an object (C/C++ code sample)

4 byte integer = 32 bits, can store 32 individual flags

```
int flags;
#define FEATURE1     0x00000001
#define FEATURE2     0x00000002
#define FEATUER3     0x00000004
#define FEATURE4     0x00000008
#define FEATURE5     0x00000010
#define FEATURE6     0x00000020
```

important: every flag must use only a single bit! e.g.
first bit
second bit
third bit, etc.

Values starting with 0x indicate hexadecimal content

Usage Example

- Clearing all flags: just set to 0, e.g.

```
flag = 0;
```

- Setting a flag is done using binary OR operation, e.g.

```
flag = flag | FEATURE1;
flag = flag | FEATURE3;
```

- Check whether a flag is set is verified using binary AND operation, e.g

```
if ( flag & FEATURE2 ) {
        //do something;
}
```

If Statement

C code:

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

If Statement

C code:

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

Conditional Branching: Jump/Branch based on condition from one location in code to another (not necessarily the next instruction)

Instructions for Making Decisions

- Define Labels for instructions.
- LEGv8 Code:

L1: *ADD X9, X21, X9*
Label

Instructions for Making Decisions

- Define Labels for instructions.
- LEGv8 Code:

  L1: *ADD X9, X21, X9*

  Label

- Labels are only for Assembly language
- Assembler changes them to address in machine code

Instructions for Making Decisions

- Define Labels for instructions.
- LEGv8 Code:

  L1: *ADD X9, X21, X9*

- Unconditional Branch: Instruct computer to branch to label
- B – branch to label

Instructions for Making Decisions

- Define Labels for instructions.
- LEGv8 Code:

  L1: *ADD X9, X21, X9*

- Unconditional Branch: Instruct computer to branch to label
- B – branch to label
- LEGv8 Code:

  B L1 // Branch to statement with label L1

Instructions for Making Decisions

- Define Labels for instructions.
- LEGv8 Code:

  L1: *ADD X9, X21, X9*

- Unconditional Branch: Instruct computer to branch to label
- Conditional Branch: Instruct computer to branch to instruction using the label if some condition is satisfied.
- CBZ – compare and branch if zero
- CBNZ – compare and branch if not zero

Instructions for Making Decisions

- Define Labels for instructions.
- LEGv8 Code:

  L1: *ADD X9, X21, X9*

- Instruct computer to branch to instruction using the label if some condition is satisfied.
- CBZ – compare and branch if zero
- CBNZ – compare and branch if not zero
- LEGv8 Code:

  $CBZ\ \ register,\ L1$ // if (register == 0) branch to instruction labeled L1;

  $CBNZ\ \ register,\ L1$ // if (register != 0) branch to instruction labeled L1;

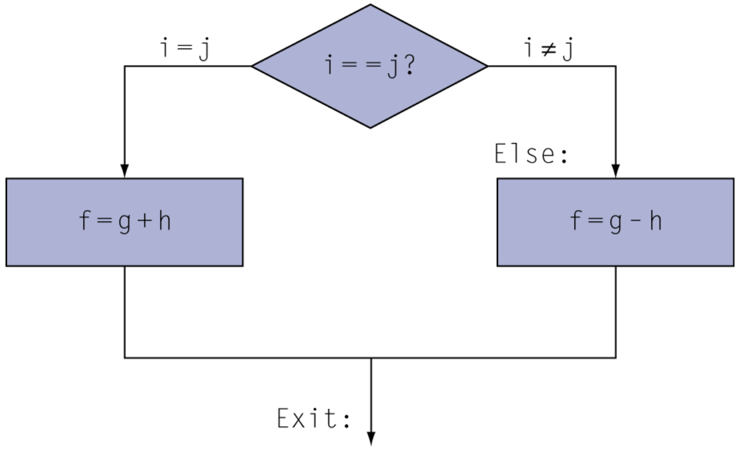
Example: Compiling If Statements

- C code:

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

  - i, j in X22, X23,
  - f, g, h, in X19, X20, X21
- Compiled LEGv8 code:

?



X9 will be zero if i=j

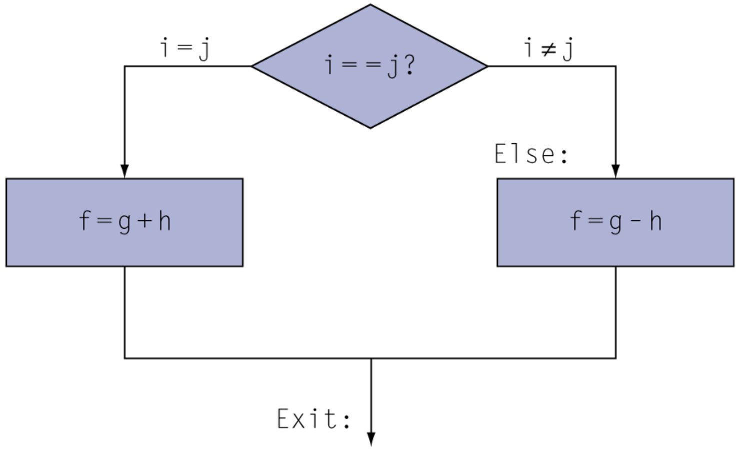
Example: Compiling If Statements

- C code:

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

  - i, j in X22, X23,
  - f, g, h, in X19, X20, X21
- Compiled LEGv8 code:

```
        SUB X9,X22,X23
```



X9 will be zero if i=j

Example: Compiling If Statements

- C code:

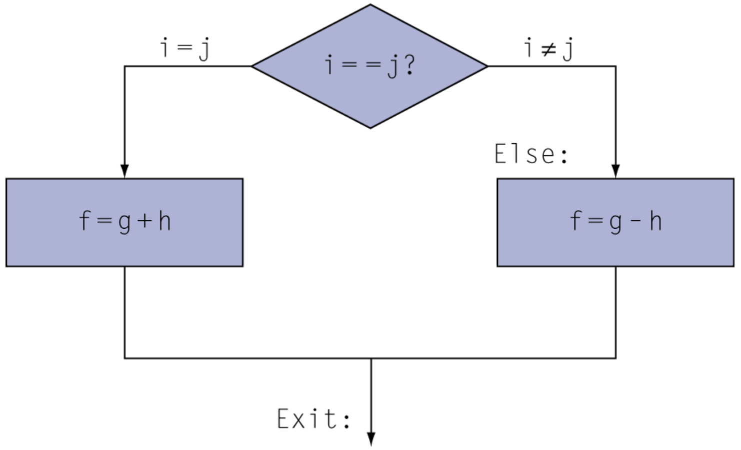
```
if ( i==j )
     f = g+h;
else
     f = g-h;
```

  - i, j in X22, X23,
  - f, g, h, in X19, X20, X21
- Compiled LEGv8 code:

```
             SUB X9,X22,X23




Else:        SUB X19,X20,x21
```



X9 will be zero if i=j

Example: Compiling If Statements

- C code:

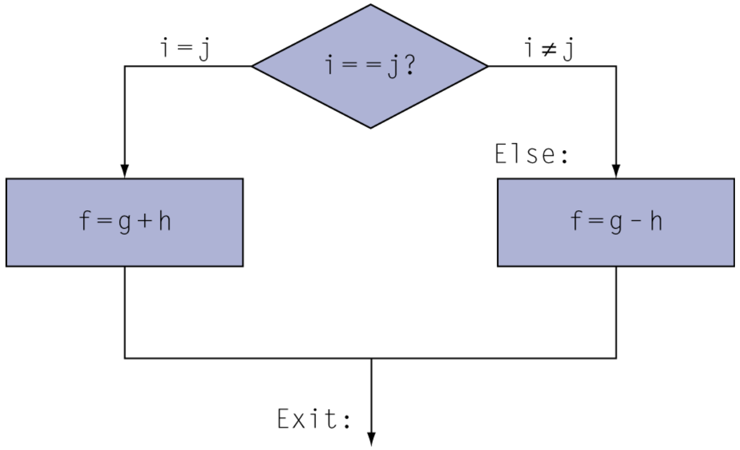
```
if ( i==j )
    f = g+h;
else
    f = g-h;
```

  – i, j in X22, X23,
  – f, g, h, in X19, X20, X21

- Compiled LEGv8 code:

```
         SUB X9,X22,X23
         CBNZ X9,Else


Else:    SUB X19,X20,x21
```



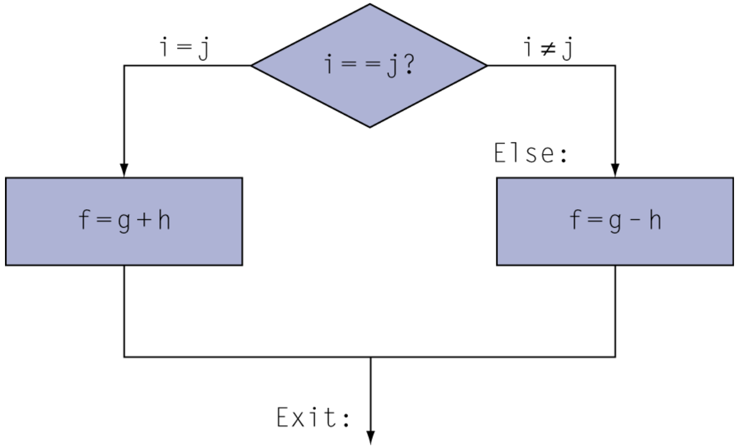
X9 will be zero if i=j

Example: Compiling If Statements

- C code:

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

  - i, j in X22, X23,
  - f, g, h, in X19, X20, X21



- Compiled LEGv8 code:

```
          SUB X9,X22,X23
          CBNZ X9,Else
          ADD X19,X20,X21

Else:     SUB X19,X20,x21
```

X9 will be zero if i=j

Example: Compiling If Statements

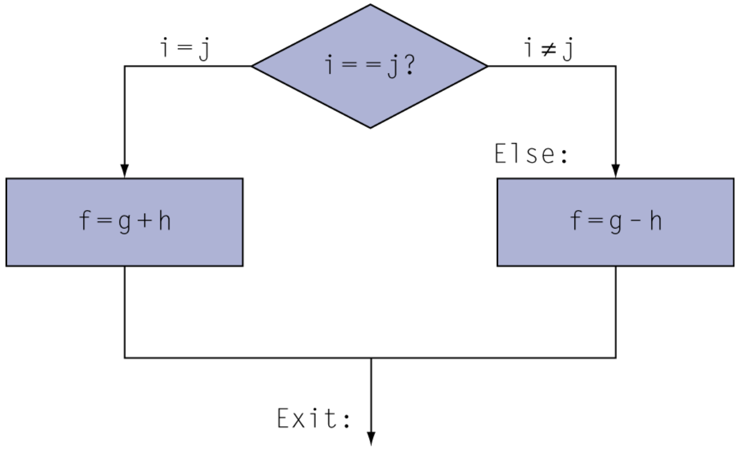
- C code:

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

  - i, j in X22, X23,
  - f, g, h, in X19, X20, X21
- Compiled LEGv8 code:

```
           SUB X9,X22,X23
           CBNZ X9,Else
           ADD X19,X20,X21

Else:      SUB X19,X20,x21
Exit: …
```



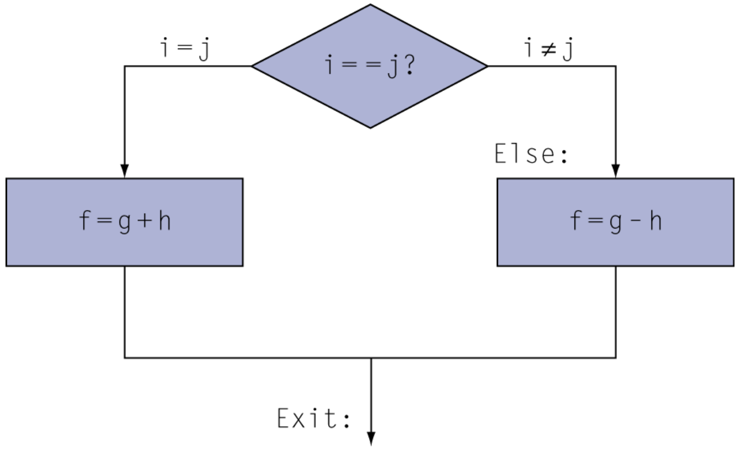
X9 will be zero if i=j

Example: Compiling If Statements

- C code:

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

  - i, j in X22, X23,
  - f, g, h, in X19, X20, X21



- Compiled LEGv8 code:

```
              SUB X9,X22,X23
              CBNZ X9,Else
              ADD X19,X20,X21
              B Exit
Else:         SUB X19,X20,x21
Exit: …
```

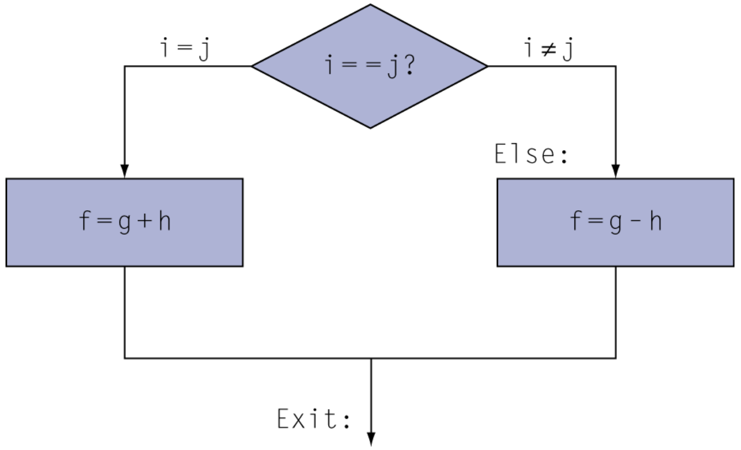
X9 will be zero if i=j

Example: Compiling If Statements

- C code:

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

  - i, j in X22, X23,
  - f, g, h, in X19, X20, X21



- Compiled LEGv8 code:

```
          SUB X9,X22,X23
          CBNZ X9,Else
          ADD X19,X20,X21
          B Exit
Else:     SUB X19,X20,x21
Exit: …
```

X9 will be zero if i=j

Compiling Loop Statements

- C code:

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

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:
  ?