CS330 - Computer Organization and Assembly Language Programming

Lecture 13

-Review-

Professor : Mahmut Unan – UAB CS

Agenda

- **Review**
  You are responsible for all the topics
  This is a rough review

- **Midterm Exam**
  February 24th, 2022 Thursday
  Lecture Time

Exam 1

- 16 multiple choice questions
- ~4 calculations/conversion type
- ~3 open ended questions

- 50+2 points
- 75 minutes
- Closed books/notes/etc

Review

Hello World !

A typical *C program* basically consists of the following parts;

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comment

```
#include <stdio.h>
int main()
{
 /* the first program in CS330 */
 printf("hello, world\n");
 return 0;
}
```

Information is Bits + Context

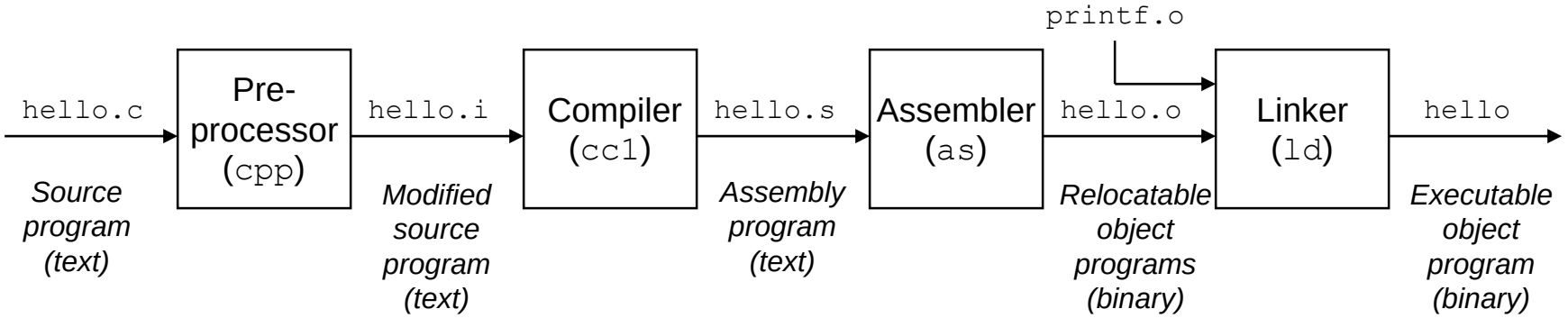
- "hello.c" is a source code
  - Sequence of bits (0 or 1)
  - 8-bit data chunks are called bytes
  - Each byte represents some text character in the program

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>	w	o	r	l	d	\	n	"	)	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

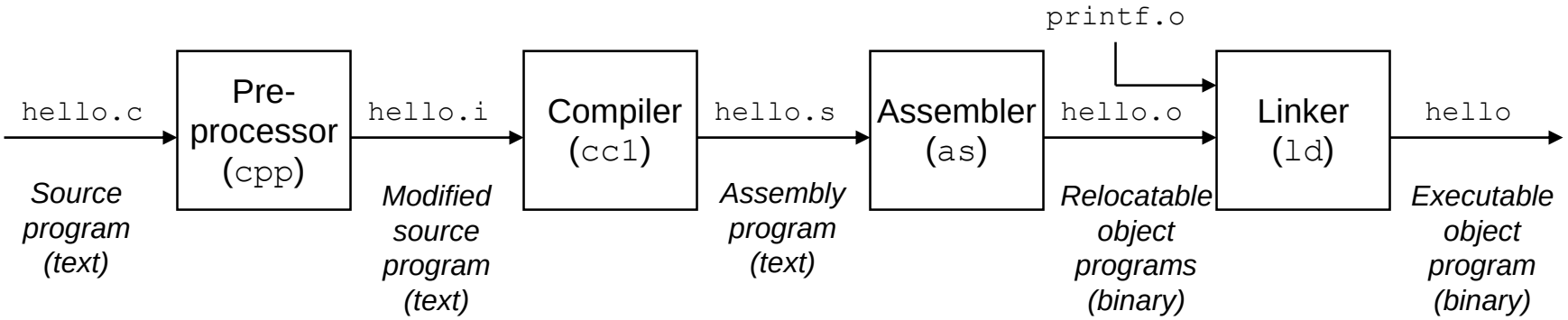
Figure 1.2 **The ASCII text representation of `hello.c`.**

Programs translated by other programs

- `unix> gcc –o hello hello.c`



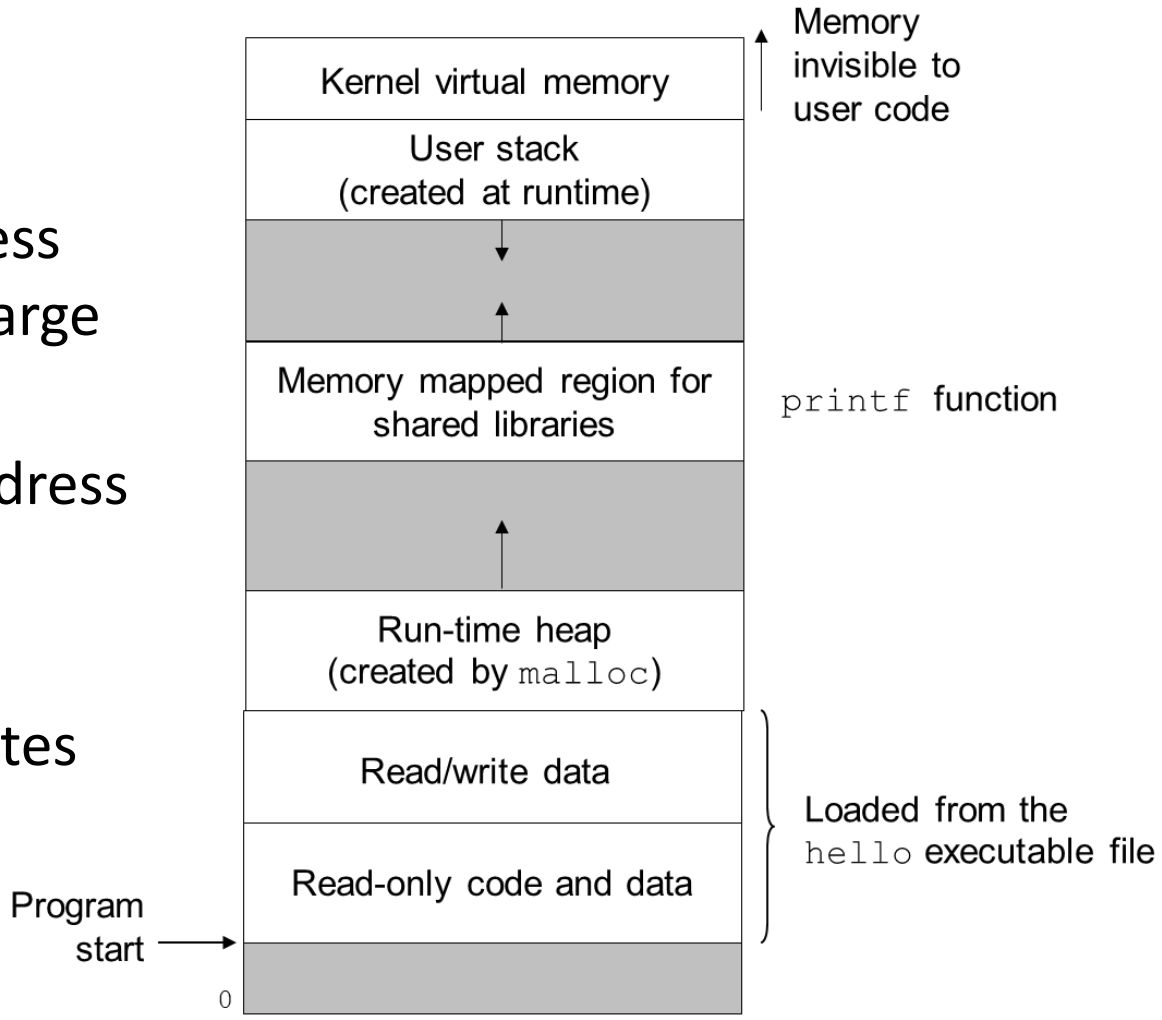
Compilation System



- Pre-processing
  - E.g., `#include<stdio.h>` is inserted into hello.i
- Compilation (.s)
  - Each statement is an assembly language program
- Assembly (.o)
  - A binary file whose bytes encode mach. language instructions
- Linking
  - Get printf() which resides in a separate precompiled object file

Virtual memory

- Illusion that each process has exclusive use of a large main memory
  - Example: Virtual address space for Linux
- **Files:** A sequence of bytes



Amdahl's Law

- Effectiveness of improving the performance of one part of system
- Speed up one part → Effect on the overall system performance?
- $T_{new} = (1 - \alpha)T_{old} + \frac{\alpha T_{old}}{k}$
- $= T_{old}[(1 - \alpha) + \frac{\alpha}{k}]$

- $S = \frac{T_{old}}{T_{new}}$
- $S = \frac{1}{((1-\alpha)+(\alpha/k))}$

Amdahl's Law / Example

- Consider a system;
  - A part of the system initially consumed 60% of the time (α = 0.6)
  - It is sped up by a factor of **3** (k=3)

- Overall improvement ?

Amdahl's Law / Example

- Consider a system;
  - A part of the system initially consumed 60% of the time (α = 0.6)
  - It is sped up by a factor of **3** (k=3)

- Overall improvement ?
- $S = \frac{1}{((1-\alpha)+(\alpha/k))}$

- = 1 / [0.4 + 0.6/3] = **1.67 times**

Amdahl's Law / Example 2

- Calculate the following improvements on a current system and decide which one is better

- **1)** if we make 90% of a program run 10 times faster.

- **2)** if we make 80% of a program run 20% faster

Amdahl's Law / Example 2

- **1)** if we make 90% of a program run 10 times faster.

$$S = \frac{1}{((1-\alpha)+(\alpha/k))} = \frac{1}{((1-0.9)+(0.9/10))} = \mathbf{5.26}$$

- **2)** if we make 80% of a program run 20% faster

$$S = \frac{1}{((1-\alpha)+(\alpha/k))} = \frac{1}{((1-0.8)+(0.8/1.2))} = \mathbf{1.153}$$

Data Measurement Chart

Data Measurement	Size
Bit	Single Binary Digit (1 or 0)
Byte	8 bits
Kilobyte (KB)	1,024 Bytes
Megabyte (MB)	1,024 Kilobytes
Gigabyte (GB)	1,024 Megabytes
Terabyte (TB)	1,024 Gigabytes
Petabyte (PB)	1,024 Terabytes
Exabyte (EB)	1,024 Petabytes

The Decimal System

- System based on decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) to represent numbers
- For example the number 83 means eight tens plus three:

$$83 = (8 * 10) + 3$$

- The number 4728 means four thousands, seven hundreds, two tens, plus eight:

$$4728 = (4 * 1000) + (7 * 100) + (2 * 10) + 8$$

- The decimal system is said to have a ***base***, or ***radix***, of 10. This means that each digit in the number is multiplied by 10 raised to a power corresponding to that digit's position:

$$83 = (8 * 10^1) + (3 * 10^0)$$

$$4728 = (4 * 10^3) + (7 * 10^2) + (2 * 10^1) + (8 * 10^0)$$

The Binary System

- Only two digits, 1 and 0
- Represented to the base 2
- The digits 1 and 0 in binary notation have the same meaning as in decimal notation:

$$0_2 = 0_{10}$$
$$1_2 = 1_{10}$$

- To represent larger numbers each digit in a binary number has a value depending on its position:

$$10_2 = (1 * 2^1) + (0 * 2^0) = 2_{10}$$
$$11_2 = (1 * 2^1) + (1 * 2^0) = 3_{10}$$
$$100_2 = (1 * 2^2) + (0 * 2^1) + (0 * 2^0) = 4_{10}$$

For representing numbers in base 2, there are two possible digits (0, 1) in which column values are a power of two:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	**1**	**1**	**0**	**0**	**0**	**1**	**1**

0 + 64 + 32 + 0 + 0 + 0 + 2 + 1 = **99**

Although values represented in base 2 are significantly longer than those in base 10, **binary representation is used in digital computing because of the resulting simplicity of hardware design**

Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write $FA1D37B_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

1100 1001 0111 1011 → 0xC97B

Machine Words

- Machine has “word size”
  - Nominal size of integer-valued data
  - More importantly – a virtual address is encoded by such a word
    - Hence, it determines max size of virtual address space
  - Most current machines are 32 bits (4 bytes)
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - Newer systems are 64 bits (8 bytes)
    - Potentially address ≈ 1.84 X 10^19 bytes
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

Data Sizes

- Each computer has a word size
  - For a machine with w-bit word size
  - The virtual address can range from 0 to 2^w-1
  - The program access to at most 2^w bytes

- 32 bit vs 64 bit

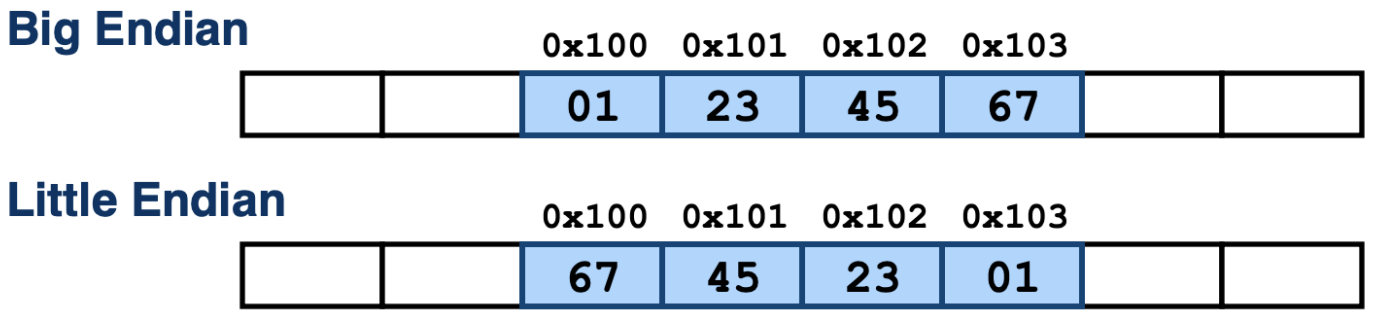
Addressing and Byte Ordering

- For objects that span multiple bytes (e.g. integers), we need to agree on two things
  - what would be the address of the object?
  - how would we order the bytes in memory?

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Byte Ordering

- How to order bytes within multi-byte word in memory
- Conventions
  - (most) Sun's, IBMs are "Big Endian" machines
    - Least significant byte has highest address (comes last)
  - (most) Intel's are "Little Endian" machines
    - Least significant byte has lowest address (comes first)
- Example
  - Variable x has 4-byte representation 0x01234567
  - Address given by &x is 0x100 0x100 0x101



http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Boolean Variables and Operations

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - ⟨{0,1}, |, &, ~, 0, 1⟩
  - | is “sum” operation, & is “product” operation
  - ~ is “complement” operation (not additive inverse)
  - 0 is identity for sum, 1 is identity for product
- Makes use of variables and operations
  - Are logical
  - A variable may take on the value 1 (TRUE) or 0 (FALSE)
  - Basic logical operations are AND, OR, XOR and NOT

Boolean Variables and Operations / 2

- AND
  - Yields true (binary value 1) if and only if both of its operands are true
  - In the absence of parentheses the AND operation takes precedence over the OR operation
  - When no ambiguity will occur the AND operation is represented by simple concatenation instead of the dot operator
- OR
  - Yields true if either or both of its operands are true
- NOT
  - Inverts the value of its operand

Table: Boolean Operators

(a) Boolean Operators of Two Input Variables

P	Q	NOT P ($\overline{P}$)	P AND Q ($P \cdot Q$)	P OR Q ($P + Q$)	P NAND Q ($\overline{P \cdot Q}$)	P NOR Q ($\overline{P + Q}$)	P XOR Q ($P \oplus Q$)
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

(b) Boolean Operators Extended to More than Two Inputs (A, B, . . .)

Operation	Expression	Output = 1 if
AND	$A \cdot B \cdot \ldots$	All of the set {A, B, …} are 1.
OR	$A + B + \ldots$	Any of the set {A, B, …} are 1.
NAND	$\overline{A \cdot B \cdot \ldots}$	Any of the set {A, B, …} are 0.
NOR	$\overline{A + B + \ldots}$	All of the set {A, B, …} are 0.
XOR	$A \oplus B \oplus \ldots$	The set {A, B, …} contains an odd number of ones.

Table: Basic Identities of Boolean Algebra

Basic Postulates		
$A \cdot B = B \cdot A$	$A + B = B + A$	Commutative Laws
$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$	Distributive Laws
$1 \cdot A = A$	$0 + A = A$	Identity Elements
$A \cdot \overline{A} = 0$	$A + \overline{A} = 1$	Inverse Elements
Other Identities		
$0 \cdot A = 0$	$1 + A = 1$	
$A \cdot A = A$	$A + A = A$	
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$	$A + (B + C) = (A + B) + C$	Associative Laws
$\overline{A \cdot B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \cdot \overline{B}$	DeMorgan's Theorem

Exercise 1

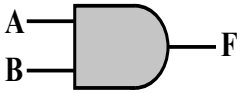
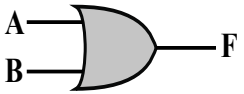
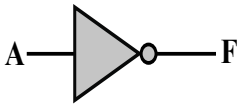
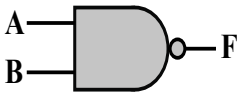
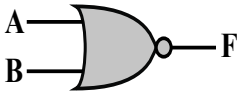
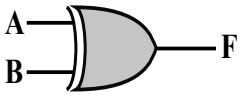
Evaluate the following expression when A =0, B =1, and C= 1

F= B + $\overline{\mathrm{C}}\mathrm{A} + \mathrm{B}\overline{\mathrm{A}}$ + $\mathrm{A}\overline{\mathrm{B}}$

Simplify the following functions;

F= AB + BC+$\overline{\mathrm{B}}\mathrm{C}$

F = A +$\overline{\mathrm{A}}$ B

Name	Graphical Symbol	Algebraic Function	Truth Table
AND		$F = A \cdot B$ or $F = AB$	A B | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = A + B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT		$F = \overline{A}$ or $F = A'$	A | F 0 | 1 1 | 0
NAND		$F = \overline{AB}$	A B | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = \overline{A + B}$	A B | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
XOR		$F = A \oplus B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0

General Boolean Algebras

- Boolean operations can be extended to work on bit vectors
  - Operations applied bitwise

```
  01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101    ~ 01010101
  --------      --------      --------      --------
  01000001      01111101      00111100      10101010
```

- All of the properties of Boolean algebra apply
- Now, Boolean |, & and ~ correspond to set union, intersection and complement

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Bit-Level Operations in C

- Operations **&, |, ~, ^** Available in C
  - Apply to any "integral" data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Arguments applied bit-wise
- Examples (Char data type)
  - ~0x41 → 0xBE
    - $\sim 01000001_2 \rightarrow 10111110_2$
  - ~0x00 → 0xFF
    - $\sim 00000000_2 \rightarrow 11111111_2$
  - 0x69 & 0x55 → 0x41
    - $01101001_2 \;\&\; 01010101_2 \rightarrow 01000001_2$
  - 0x69 | 0x55 → 0x7D
    - $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - **&&, ||, !**
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination
- Examples (char data type)
  - !0x41 → 0x00
  - !0x00 → 0x01
  - !!0x41 → 0x01
  - 0x69 && 0x55 → 0x01
  - 0x69 || 0x55 → 0x01
  - p && *p (avoids null pointer access)

Watch out for && vs. & (and || vs. |)…
one of the more common oopsies in C programming

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
    - Useful in two's compliment
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument **x**	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument **x**	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Boolean Algebra ≈ Integer Ring

Commutative	A | B = B | A A & B = B & A	A + B = B + A A * B = B * A
Associativity	(A | B) | C = A | (B | C) (A & B) & C = A & (B & C)	(A + B) + C = A + (B + C) (A * B) * C = A * (B * C)
Product distributes over sum	A & (B | C) = (A & B) | (A & C)	A * (B + C) = A * B + B * C
Sum and product identities	A | 0 = A A & 1 = A	A + 0 = A A * 1 = A
Zero is product annihilator	A & 0 = 0	A * 0 = 0
Cancellation of negation	~ (~ A) = A	– (– A) = A

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Boolean Algebra ≠ Integer Ring

Boolean: Sum distributes over product	A | (B & C) = (A | B) & (A | C)	A + (B * C) ≠ (A + B) * (B + C)
Boolean: Idempotency	A | A = A A & A = A	A + A ≠ A A * A ≠ A
Boolean: Absorption	A | (A & B) = A A & (A | B) = A	A + (A * B) ≠ A A * (A + B) ≠ A
Boolean: Laws of Complements	A | ~A = 1	A + –A ≠ 1
Ring: Every element has additive inverse	A | ~A ≠ 0	A + –A = 0

http://www.aqualab.cs.northwestern.edu/component/attachments/download/167

Properties of & and ^

- Boolean ring
  - ⟨{0,1}, ^, &, I, 0, 1⟩
  - Identical to integers mod 2
  - I is identity operation: I (A) = A
    - A ^ A = 0
- Property: Boolean ring
  - Commutative sum — A ^ B = B ^ A
  - Commutative product — A & B = B & A
  - Associative sum — (A ^ B) ^ C = A ^ (B ^ C)
  - Associative product — (A & B) & C = A & (B & C)
  - Prod. over sum — A & (B ^ C) = (A & B) ^ (B & C)
  - 0 is sum identity — A ^ 0 = A
  - 1 is prod. identity — A & 1 = A
  - 0 is product annihilator — A & 0 = 0
  - Additive inverse — A ^ A = 0

Unsigned Encodings

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

(Binary To Unsigned)

e.g. B2U ([1011]) = $1 * 2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 11$

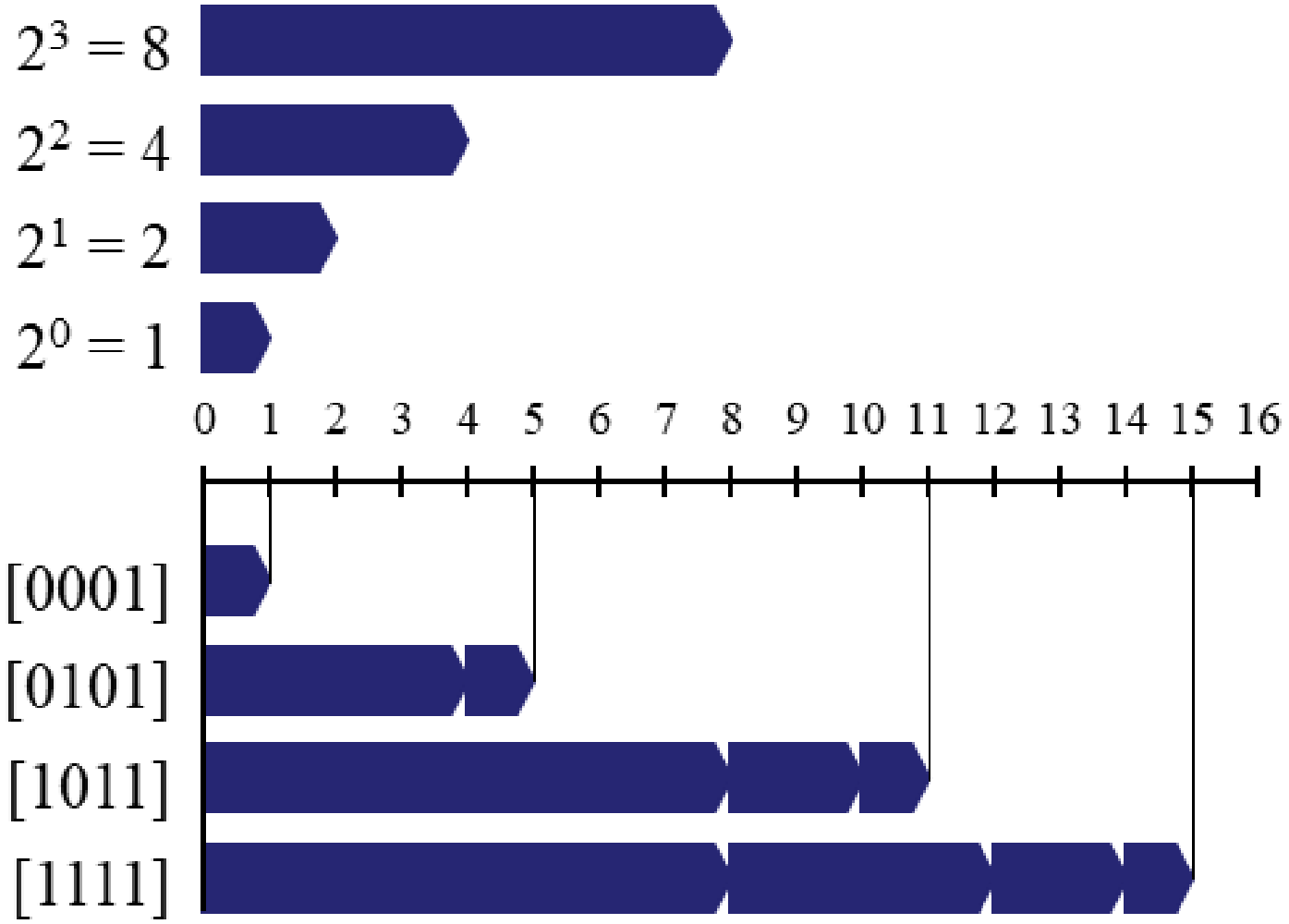
– C `short` 2 bytes long

```
short int x = 15213;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101

http://www.aqualab.cs.northwestern.edu/component/attachments/download/168

Examples


$2^3 = 8$
$2^2 = 4$
$2^1 = 2$
$2^0 = 1$
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
[0001]
[0101]
[1011]
[1111]

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit

- e.g. B2T ([1011]) = $-1 * 2^3 + 0*2^2 + 1*2^1 + 1*2^0 = -5$
- C `short` 2 bytes long

```
short int y = -15213;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

- Sign bit
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative; 1 for negative

http://www.aqualab.cs.northwestern.edu/component/attachments/download/168

Two's Complement

- Invert and add **one**

Suppose we're working with 8 bit quantities and suppose we want to find how **-28** would be expressed in two's complement notation.

- First we write out 28 in **binary form.**
  00011100
- Then we **invert the digits**. 0 becomes 1, 1 becomes 0.
  11100011
- Then we **add 1.**
  11100100

That is how one would write -28 in 8 bit binary.

https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html

Numeric Ranges

- **Unsigned Values**
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1
- **Two's Complement Values**
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
- **Other Values**
  - Minus 1
    111…1

Values for *W* = 16

	Decimal	Hex	Binary
UMax	**65535**	**FF FF**	**11111111 11111111**
TMax	**32767**	**7F FF**	**01111111 11111111**
TMin	**-32768**	**80 00**	**10000000 00000000**
-1	**-1**	**FF FF**	**11111111 11111111**
0	**0**	**00 00**	**00000000 00000000**

Signed vs. Unsigned in C

- Constants
  - By default are considered to be signed integers
  - Unsigned if have “U” as suffix

    `0U, 4294967259U`
- Casting
  - Explicit casting between signed & unsigned same as U2T and T2U

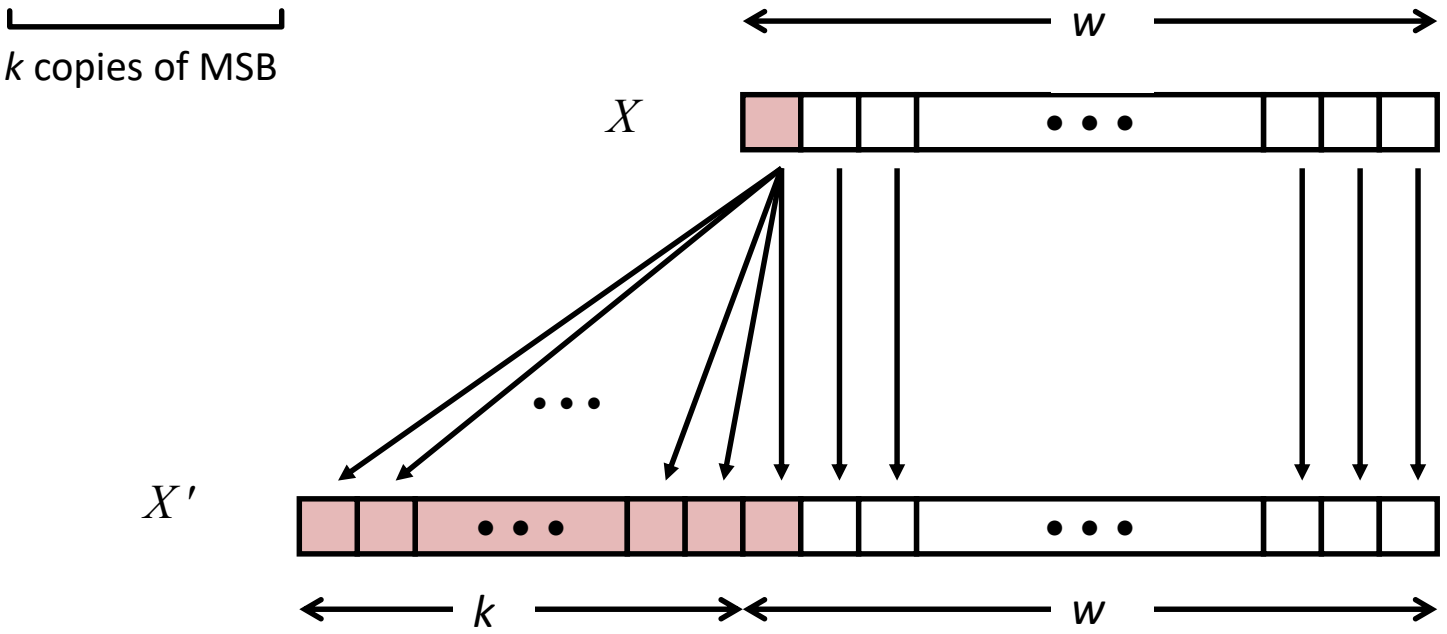
```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

  - Implicit casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty;
```

Sign Extension

- **Task:**
  - Given *w*-bit signed integer *x*
  - Convert it to *w*+*k*-bit integer with same value
- **Rule:**
  - Make *k* copies of sign bit:
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$



Truncating Numbers

- Reduce the number of bits representing the number
- Truncating *w-bit* number to a *k* bit number, we drop the high order *w-k* bits
  - Can alter its value
    - A form of overflow

Summary: Expanding, Truncating: Basic Rules

- **Expanding (e.g., short int to int)**
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- **Truncating (e.g., unsigned to unsigned short)**
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behavior

OVERFLOW RULE:

- If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

Multiplication

- Goal: Computing Product of *w*-bit numbers *x*, *y*
  - Either signed or unsigned
- But, exact results can be bigger than *w* bits
  - Unsigned: up to 2*w* bits
    - Result range: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to 2*w*-1 bits
    - Result range: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to 2*w* bits, but only for $(TMin_w)^2$
    - Result range: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
- So, maintaining exact results…
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by "arbitrary precision" arithmetic packages

Unsigned Binary Multiplication

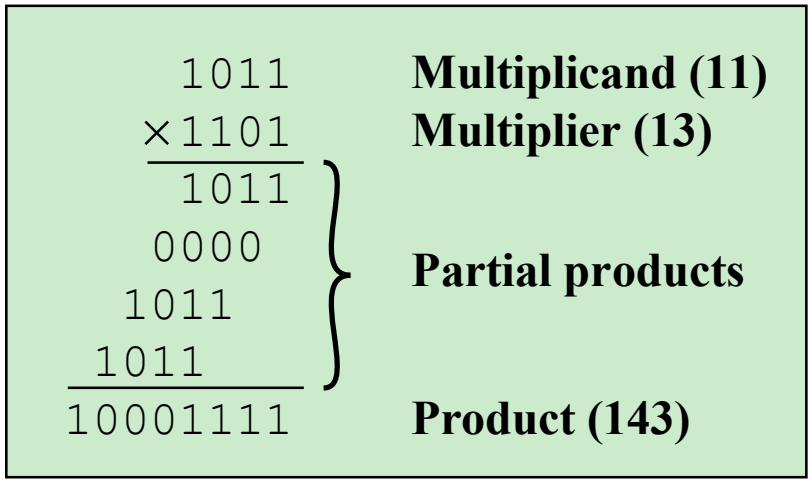
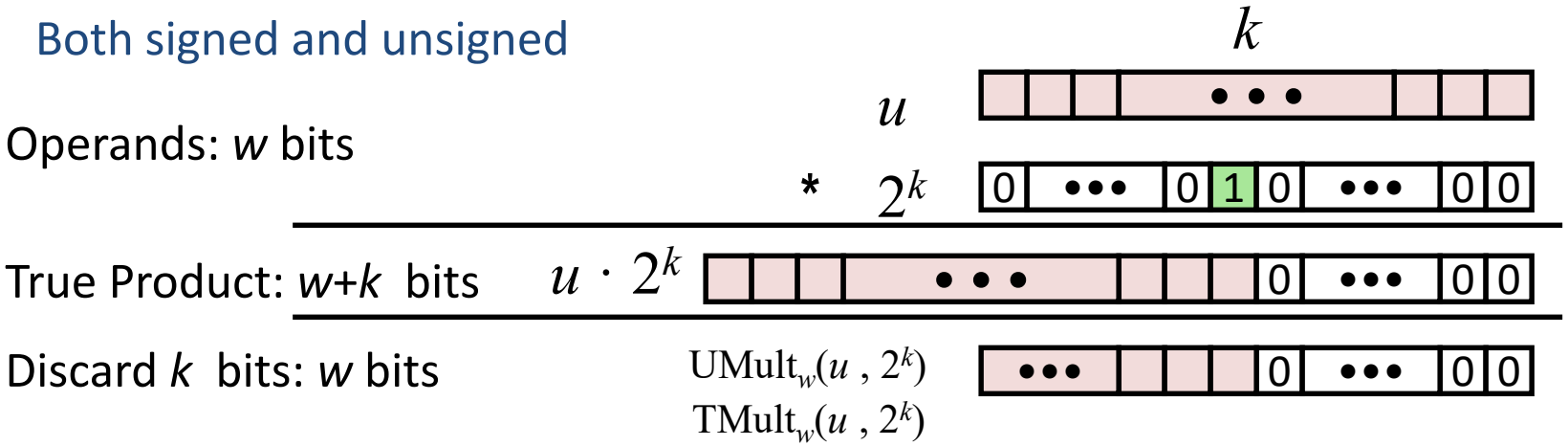


Figure 10.7 Multiplication of Unsigned Binary Integers

Power-of-2 Multiply with Shift

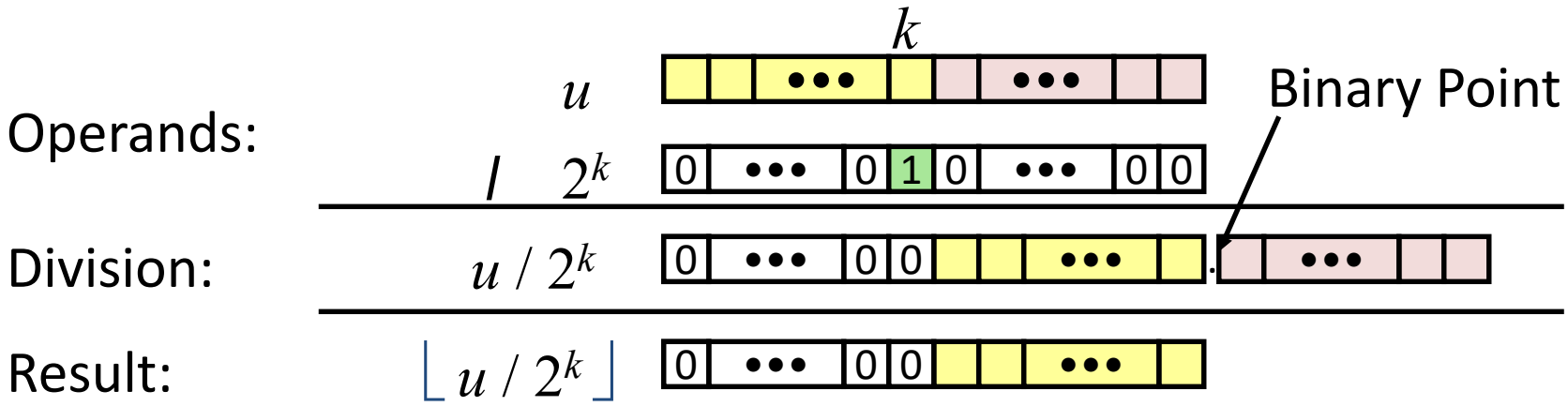
- **Operation**
  - `u << k` gives `u * 2^k`
  - Both signed and unsigned



- **Examples**
  - `u << 3 == u * 8`
  - `(u << 5) - (u << 3) == u * 24`
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically

Unsigned Power-of-2 Divide with Shift

- **Quotient of Unsigned by Power of 2**
  - `u >> k` gives $\lfloor u \;/\; 2^k \rfloor$
  - Uses logical shift



	Division	Computed	Hex	Binary
`x`	15213	15213	3B 6D	00111011 01101101
`x >> 1`	7606.5	7606	1D B6	00011101 10110110
`x >> 4`	950.8125	950	03 B6	00000011 10110110
`x >> 8`	59.4257813	59	00 3B	00000000 00111011

Arithmetic: Basic Rules

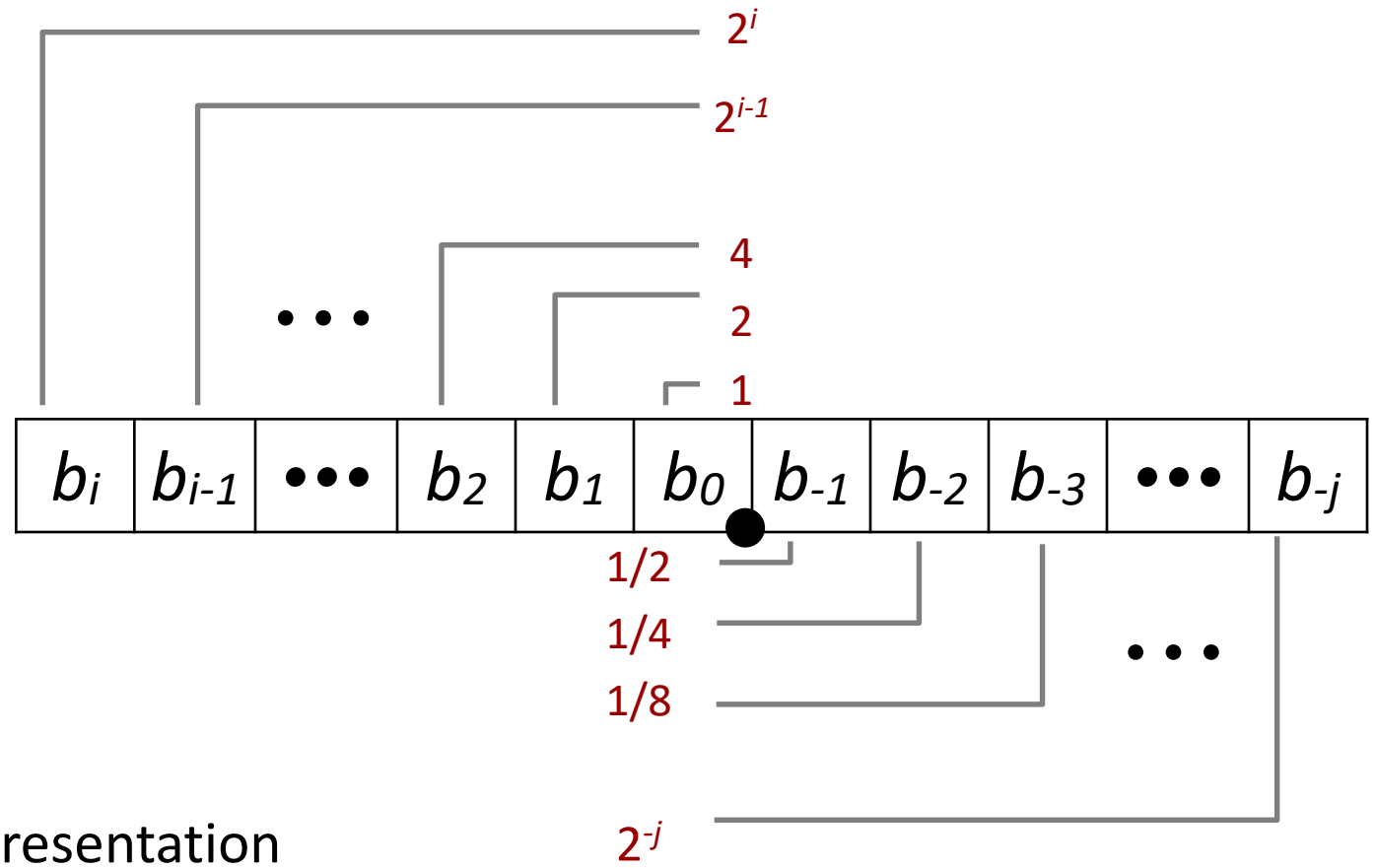
- **Addition:**
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod 2^w
    - Mathematical addition + possible subtraction of 2^w
  - Signed: modified addition mod 2^w (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2^w

- **Multiplication:**
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2^w
  - Signed: modified multiplication mod 2^w (result in proper range)

Fractional binary numbers

- What is 1011.101_2?

Fractional Binary Numbers



- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: $\sum_{k=-j}^{i} b_k \times 2^k$

Fractional Binary Numbers: Examples

Value	Representation
5 3/4	101.11_2
2 7/8	10.111_2
1 7/16	1.0111_2

- **Observations**
  - Divide by 2 by shifting right (unsigned)
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Representable Numbers

- **Limitation #1**
  - Can only exactly represent numbers of the form $x/2^k$
    - Other rational numbers have repeating bit representations
  - **Value** **Representation**
    - 1/3 $0.0101010101[01]\ldots_2$
    - 1/5 $0.001100110011[0011]\ldots_2$
    - 1/10 $0.0001100110011[0011]\ldots_2$

- **Limitation #2**
  - Just one setting of binary point within the *w* bits
    - Limited range of numbers (very small values? very large?)

IEEE Floating Point

- **IEEE Standard 754**
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs

- **Driven by numerical concerns**
  - Nice standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard

Floating Point Representation

- Numerical Form:

$$(-1)^s \, M \, 2^E$$

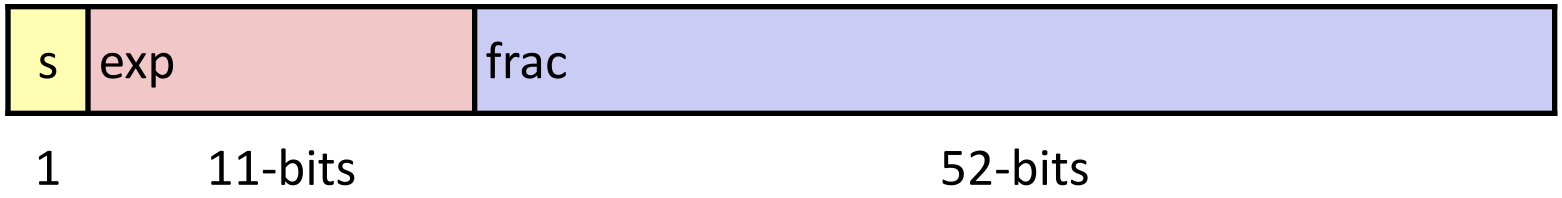
  - **Sign bit *s*** determines whether number is negative or positive
  - **Significand *M*** normally a fractional value in range [1.0,2.0).
  - **Exponent *E*** weights value by power of two

- Encoding
  - MSB s is sign bit ***s***
  - exp field encodes ***E*** (but is not equal to E)
  - frac field encodes ***M*** (but is not equal to M)

Precision options

- Single precision: 32 bits

- Double precision: 64 bits



- The value encoded by a given bit representation can be divided into three different cases, depending on the value of **exp**.
- Case 1: Normalized Values
- Case 2: Denormalized Values
- Case 3: Special Values

Case 1: "Normalized" Values

$$v = (-1)^s M 2^E$$

- *When: exp ≠ 000…0 and exp ≠ 111…1*

- Exponent coded as a ***biased*** value: ***E = Exp – Bias***
  - ***Exp*****:** unsigned value of exp field
  - ***Bias*** **= $2^{k-1} - 1$**, where *k* is number of exponent bits
    - Single precision: 127 (Exp: 1…254, E: -126…127)
    - Double precision: 1023 (Exp: 1…2046, E: -1022…1023)

- Significand coded with implied leading **1: *M* = $1.xxx…x_2$**
  - xxx…x: bits of frac field
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)
  - Get extra leading bit for "free"

Case 2: Denormalized Values

$v = (-1)^s M 2^E$
$E = 1 - Bias$

- **Condition: exp = 000…0**

- Exponent value: $E = 1 - \text{Bias}$ (instead of $E = 0 - Bias$)
- Significand coded with implied leading 0: $M = 0.xxx\ldots x_2$
  - **`xxx`**…**`x`**: bits of **`frac`**
- Cases
  - **`exp`** = `000…0`, **`frac`** = `000…0`
    - Represents zero value
    - Note distinct values: +0 and –0 (why?)
  - **`exp`** = `000…0`, **`frac`** ≠ `000…0`
    - Numbers closest to 0.0
    - Equispaced

Case 3: Special Values

- Condition: **`exp`** = **`111…1`**

- Case: **`exp`** = **`111…1`**, **`frac`** = **`000…0`**
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = −1.0/−0.0 = $+\infty$, 1.0/−0.0 = $-\infty$

- Case: **`exp`** = **`111…1`**, **`frac`** $\neq$ **`000…0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), $\infty - \infty$, $\infty \times 0$

Special Properties of the IEEE Encoding

- **FP Zero Same as Integer Zero**
  - All bits = 0

- **Can (Almost) Use Unsigned Integer Comparison**
  - Must first compare sign bits
  - Must consider −0 = 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Floating Point Operations: Basic Idea

- $x +_f y = \mathrm{Round}(x + y)$
- $x \times_f y = \mathrm{Round}(x \times y)$

- **Basic idea**
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into **`frac`**

Rounding

- **Rounding Modes (illustrate with $ rounding)**

	$1.40	**$1.60**	**$1.50**	**$2.50**	**–$1.50**
– Towards zero	$1	$1	$1	$2	–$1
– Round down (−∞)	$1	$1	$1	$2	–$2
– Round up (+∞)	$2	$2	$2	$3	–$1
– Nearest Even (default)	$1	$2	$2	$2	–$2

x86-64 Integer Registers

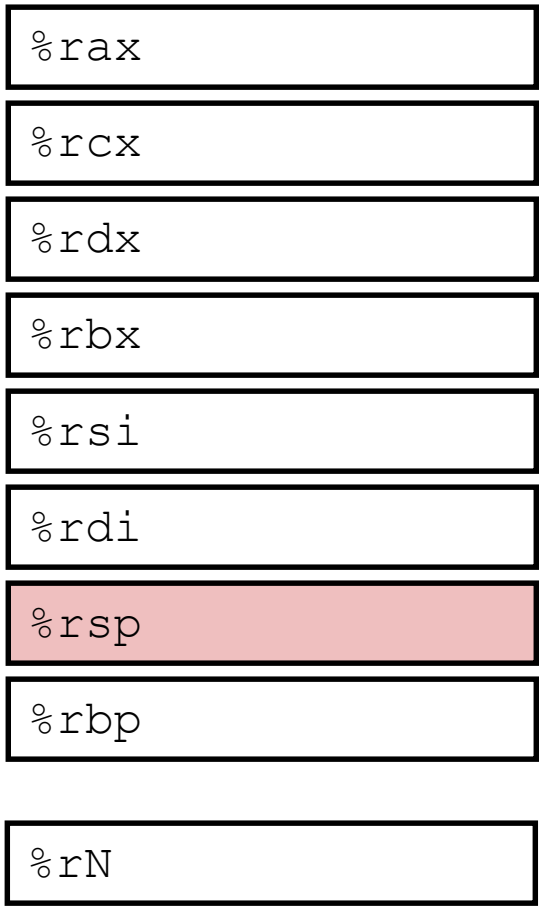
%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

– Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Some History: IA32 Registers

Moving Data

- Moving Data
  `movq` *Source*, *Dest*:
- Operand Types
  - ***Immediate:*** Constant integer data
    - Example: **`$0x400`, `$-533`**
    - Like C constant, but prefixed with **`'$'`**
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 16 integer registers
    - Example: **`%rax`, `%r13`**
    - But **`%rsp`** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example: **`(%rax)`**
    - Various other "address modes"



`movq` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

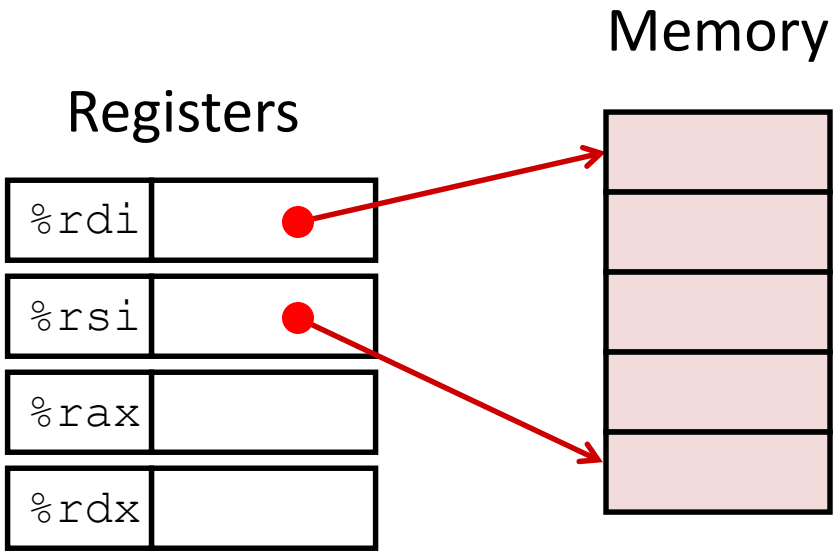
Example of Simple Addressing Modes

```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

Understanding `Swap()`

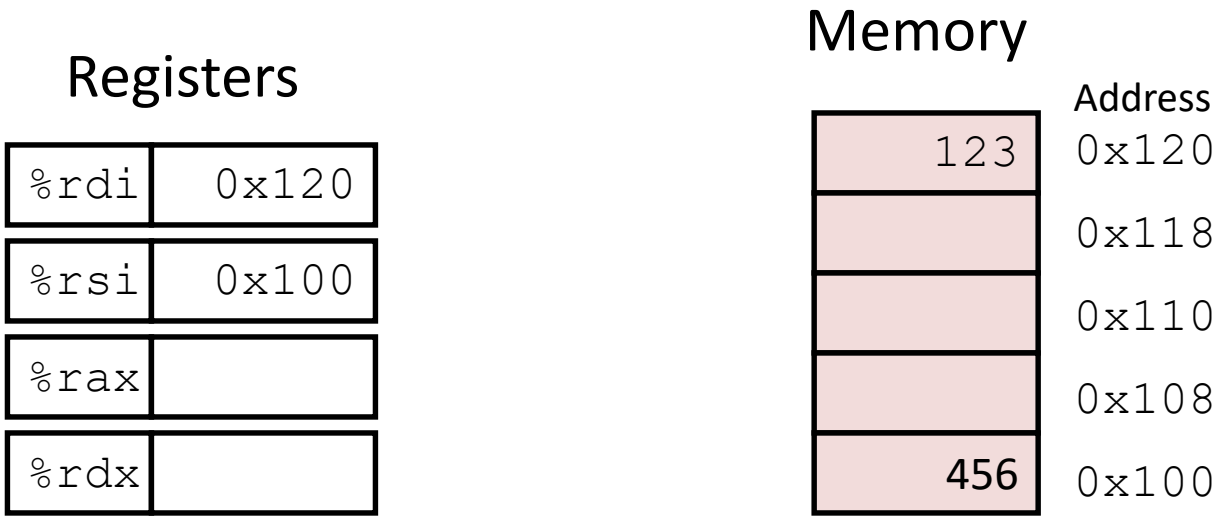
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

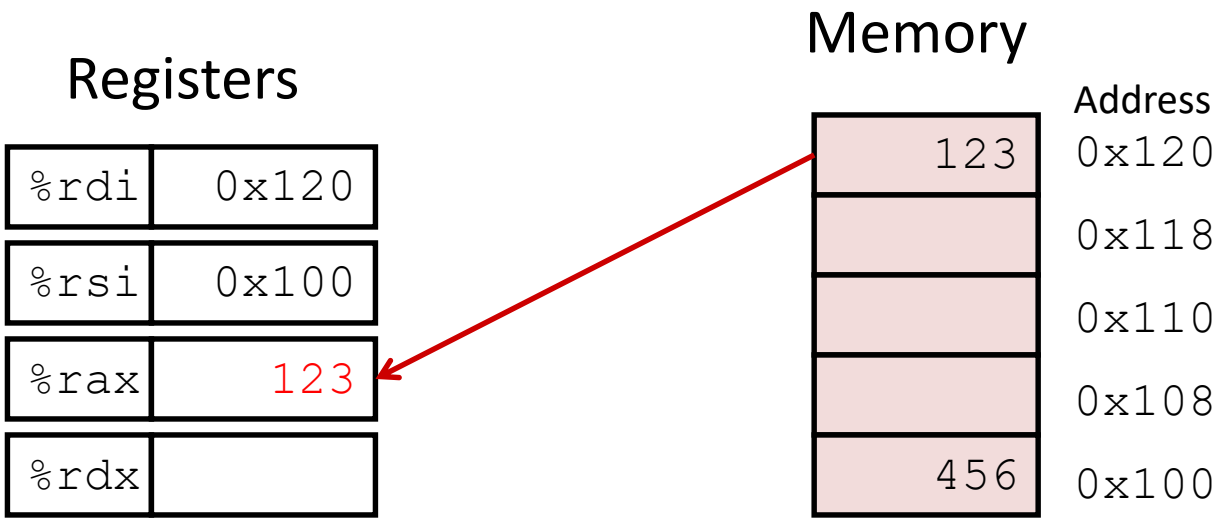
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`



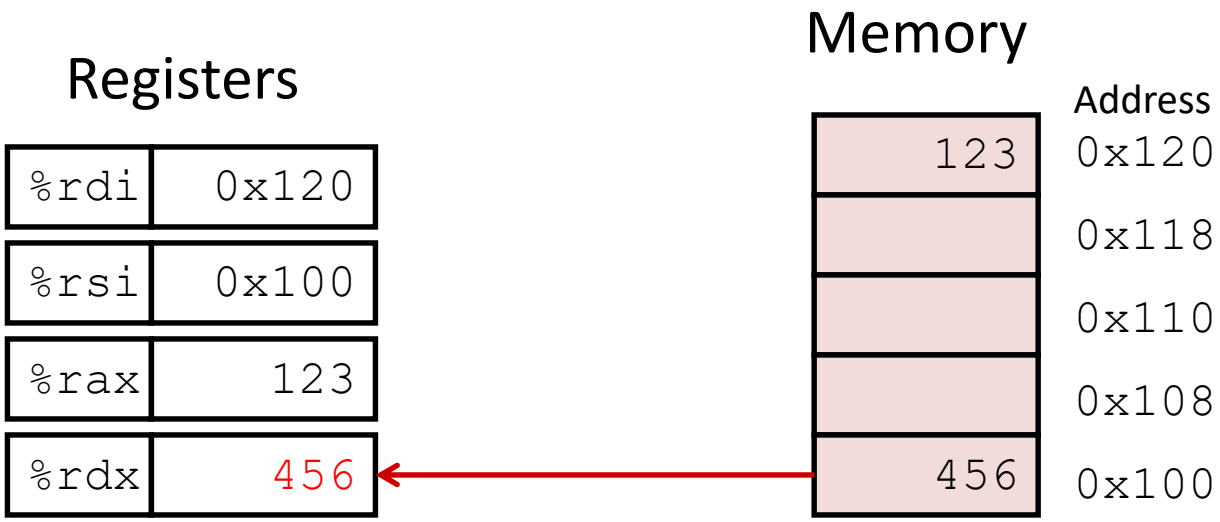

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Understanding `Swap()`



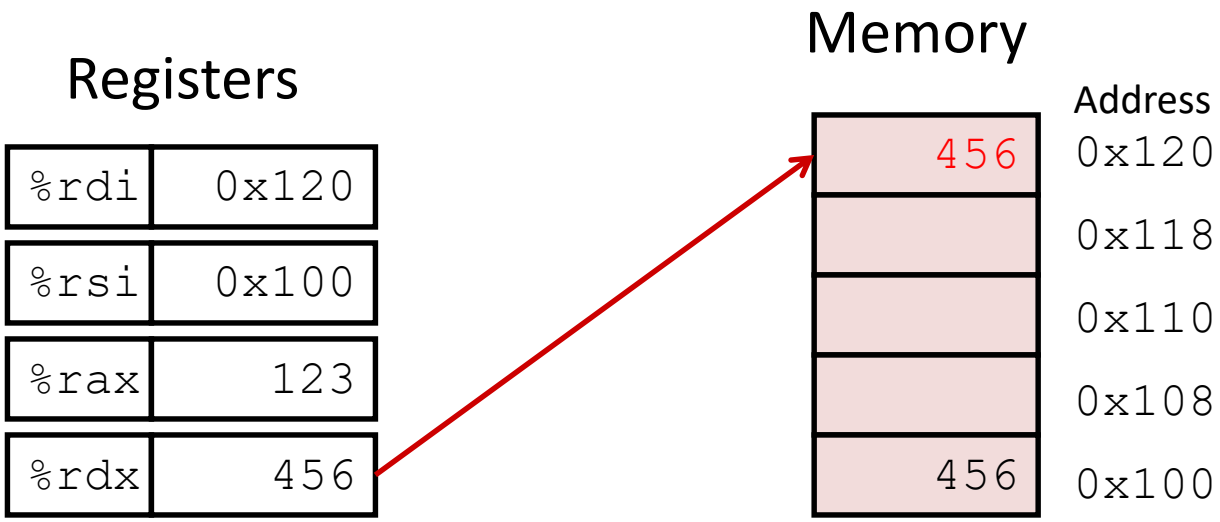

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`



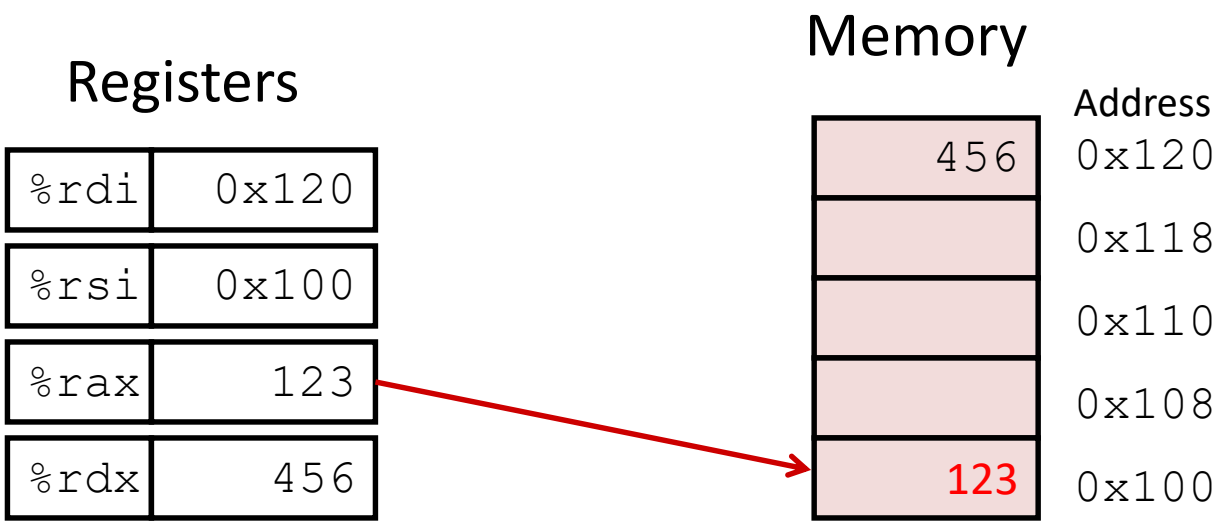

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

Complete Memory Addressing Modes

- **Most General Form**

  D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

  – D: Constant "displacement" 1, 2, or 4 bytes

  – Rb: Base register: Any of 16 integer registers

  – Ri: Index register: Any, except for `%rsp`

  – S: Scale: 1, 2, 4, or 8

- **Special Cases**

(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]
D(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]+D]
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]

AT&T Syntax vs Intel Syntax

- AT&T Sytanx

  movl %esp, %ebp

  Instruction source destination

- Intel Syntax

  movq ebp, esp

  Instruction destination source

  No percent signs

Pushing and Popping Stack Data

- Stack plays a vital role in the handling of procedure calls.
- Stack = a data structure where values can be added or deleted → according to last in first out discipline

- Add data to stack → push
- Remove data from stack → pop

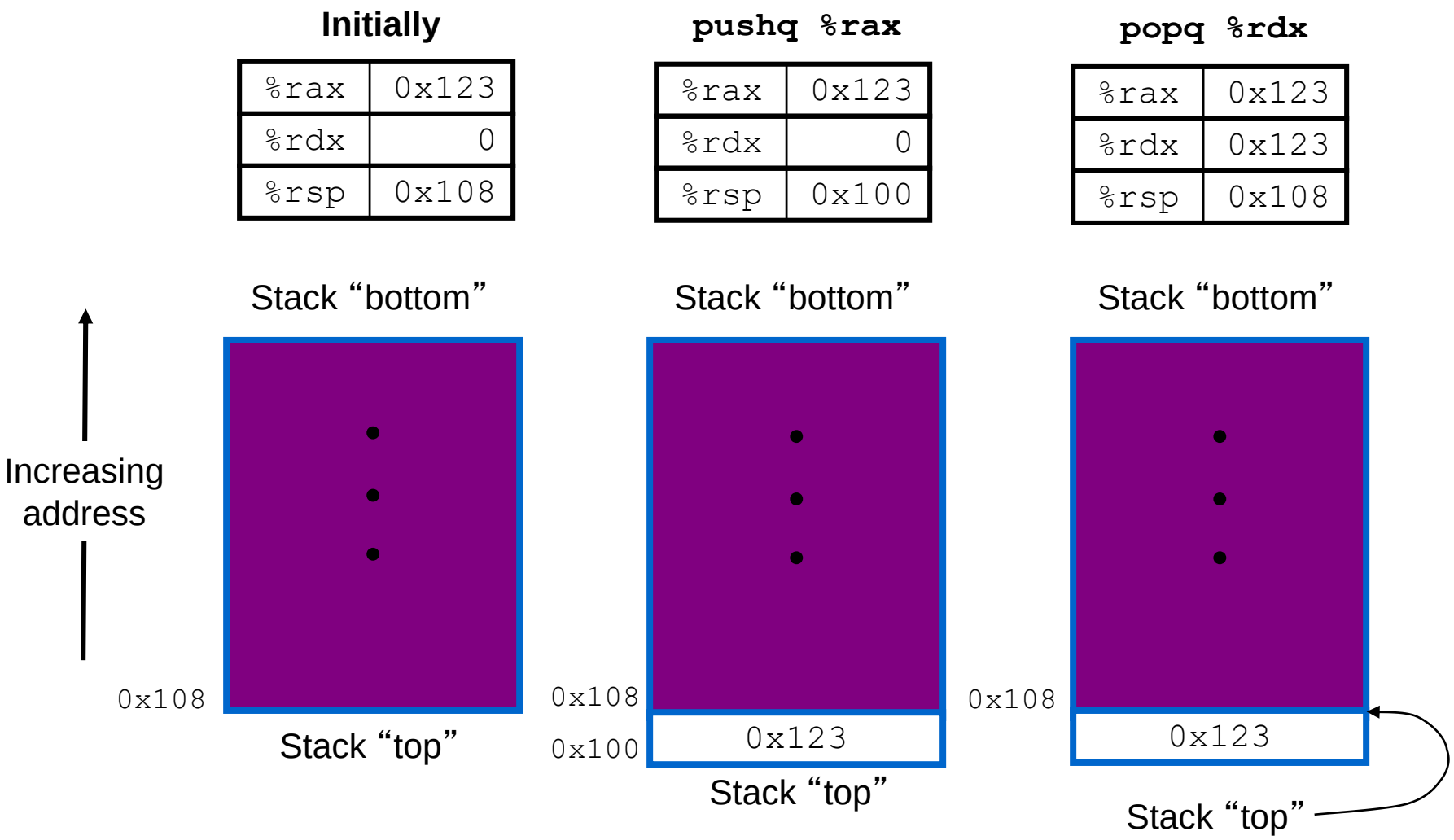
Push Source Push source onto stack

Pop Destination Pop top of stack into destination

EXAMPLES:

pushq %rbp

Popq M[R[%rsp]]


Initially
%rax 0x123
%rdx 0
%rsp 0x108
pushq %rax
%rax 0x123
%rdx 0
%rsp 0x100
popq %rdx
%rax 0x123
%rdx 0x123
%rsp 0x108
Stack "bottom"
Stack "bottom"
Stack "bottom"
Increasing address
0x108
Stack "top"
0x108
0x100
0x123
Stack "top"
0x108
0x123
Stack "top"

- The stack pointer %rsp holds the address of the top stack element.
- Pushing a quad word value onto the stack involves first decrementing the stack pointer by 8 and then writing the value at the new top of stack address.
- `pushq %rbp` is equal to

```
subq &8, %rsp
movq %rbp, (%rsp)
```

recall -Complete Memory Addressing Modes

- **Most General Form**

  D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

  – D: Constant “displacement” 1, 2, or 4 bytes

  – Rb: Base register: Any of 16 integer registers

  – Ri: Index register: Any, except for `%rsp`

  – S: Scale: 1, 2, 4, or 8

- **Special Cases**

(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]
D(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]+D]
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`		
`(%rdx,%rcx)`		
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Arithmetic
&
Logical Operations

Address Computation Instruction

- **`leaq`** ***Source***, ***Destination***
  - ***Source*** is address mode expression
  - Set ***Destination*** to address denoted by expression

  leaq S, D D ← &S Load effective address

- Uses
  - Computing addresses without a memory reference
    - E.g., translation of **`p = &x[i];`**
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8
- Example

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

More on **LEAQ**

- The destination operand must be a register

```
leaq 7(%rdx, %rdx,4), %rax
```

→set register `%rax` to `5x+7`

If `%rax` has the value x, and `%rcx` holds the value y

```
leaq 6(%rax), %rdx
leaq (%rax, %rcx), %rdx
```

More on **LEAQ**

- The destination operand must be a register

```
leaq 7(%rdx, %rdx,4), %rax
```

→set register `%rax` to `5x+7`

If `%rax` has the value x, and `%rcx` holds the value y

```
leaq 6(%rax), %rdx      → 6 + x
leaq (%rax, %rcx), %rdx → x + y
```

Basic Arithmetic Operations

- ADD
- SUB
- IMUL
- IDIV

ADD

`ADDQ  %rbx,  %rax`

- adds %rbx to %rax, and overwrites the result in %rax

`ADDQ  &8,  %rsp`

adds 8 to the stack pointer %rsp, (incrementing)

- c = a + b;

```
MOVQ   a, %rax
MOVQ   b, %rbx
ADDQ   %rbx, %rax
MOVQ %rax, c
```

leaq for basic arithmetic operations

```
long scale(long x, long y, long z){
long t= x  + 4 * y + 12 * z;
return t;
}

scale:
leaq (%rdi, %rsi, 4), %rax
leaq (%rdx, %rdx, 2), %rdx
leaq (%rax, %rdx, 4), %rax
```

leaq for basic arithmetic operations

```
long scale(long x, long y, long z){
long t= x  + 4 * y + 12 * z;
return t;
}
x in %rdi, y in %rsi, z in %rdx
scale:
leaq (%rdi, %rsi, 4), %rax      x+4*y
leaq (%rdx, %rdx, 2), %rdx     z+2*z=3*z
leaq (%rax, %rdx, 4), %rax  (x+4*y)+4*(3*z)
```

Some Arithmetic Operations

- Two Operand Instructions:

Format	***Computation***	
`addq`	*Src,Dest*	Dest = Dest + Src
`subq`	*Src,Dest*	Dest = Dest – Src
`imulq`	*Src,Dest*	Dest = Dest * Src
`salq`	*Src,Dest*	Dest = Dest << Src ***Also called shlq***
`sarq`	*Src,Dest*	Dest = Dest >> Src ***Arithmetic***
`shrq`	*Src,Dest*	Dest = Dest >> Src ***Logical***
`xorq`	*Src,Dest*	Dest = Dest ^ Src
`andq`	*Src,Dest*	Dest = Dest & Src
`orq`	*Src,Dest*	Dest = Dest | Src

imulq

Imul S, D D ← D*S

```
return a*b + c*d;
```

*Assume a in %edi, b in %sil, c in %rdx, d in %ecx

```
movslq %ecx,%rcx
movsbl %sil,%esi
imulq %rdx, %rcx
imull %edi, %esi
leal (%rsi,%rcx), %eax
ret
```

salq, shrq, sarq, shlq….

- Left/right shift operations
- salq $4, %rax → left shift rax by 4 bits

- For example

```
movw    $ff00,%ax          # ax=1111.1111.0000.0000 (0xff00, unsigned 65280, signed -256)
shrw    $3,%ax             # ax=0001.1111.1110.0000 (0x1fe0, signed and unsigned 8160)
                           # (logical shifting unsigned numbers right by 3
                           #   is like integer division by 8)
shlw    $1,%ax             # ax=0011.1111.1100.0000 (0x3fc0, signed and unsigned 16320)
                           # (logical shifting unsigned numbers left by 1
                           #   is like multiplication by 2)
```


https://en.wikibooks.org/wiki/X86_Assembly/Shift_and_Rotate

Binary Operations

xorq, andq, orq, notq…

```
movl $0x1, %eax ; eax := 1
movl $0x0, %ebx ; ebx := 0
orl  %eax, %ebx ; ebx := eax V ebx
```

ebx would be 1 because 1 V 0 ⇔ 1

```
notl %ebx  ; ebx := 0
```

Unary Operations

- Single operand serving as both source and destination
- Operand can be either a register or memory location

```
incq(%rsp)
```

similar to ++ or – in C

```
decq (%rcx)
```

Other Arithmetic Operations

- One Operand Instructions

`incq`	*Dest*	*Dest = Dest + 1*
`decq`	*Dest*	*Dest = Dest – 1*
`negq`	*Dest*	*Dest = – Dest*
`notq`	*Dest*	*Dest = ~Dest*

- See book for more instructions

Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax
   addq    %rdx, %rax
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx
   leaq    4(%rdi,%rdx), %rcx
   imulq   %rcx, %rax
   ret
```

Interesting Instructions

- **`leaq`**: address computation
- **`salq`**: shift
- **`imulq`**: multiplication
  - But, only used once

Understanding Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax    # t1
   addq    %rdx, %rax           # t2
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx             # t4
   leaq    4(%rdi,%rdx), %rcx   # t5
   imulq   %rcx, %rax           # rval
   ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	**t1**, **t2**, **rval**
%rdx	**t4**
%rcx	**t5**

```
long arith(long x, long y, long z){
long t1 = x ^ y;
long t2= z*48;
long t3= t1 & 0x0F0F0F0F;
long t4= t2-t3;
return t4;
}  // Assume x in %rdi, y in %rsi , and z in %rdx
arith:
    xorq %rsi, %rdi      long t1 = x ^ y;
    leaq  (%rdx, %rdx,2), %rax       3*z
    salq   $4, %rax       t2 = 16*(3*z)=48*z
    andl   $252645135, %edi     t3 = t1 & 0x0F0F0F0F
    subq   %rdi, %rax          return t2-t3;
    ret
```

Machine Programming I: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- C, assembly, machine code
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- Assembly Basics: Registers, operands, move
  - The x86-64 move instructions cover wide range of data movement forms
- Arithmetic
  - C compiler will figure out different instruction combinations to carry out computation

Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax
   addq    %rdx, %rax
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx
   leaq    4(%rdi,%rdx), %rcx
   imulq   %rcx, %rax
   ret
```

Interesting Instructions

- **`leaq`**: address computation
- **`salq`**: shift
- **`imulq`**: multiplication
  - But, only used once

Understanding Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax    # t1
   addq    %rdx, %rax           # t2
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx             # t4
   leaq    4(%rdi,%rdx), %rcx   # t5
   imulq   %rcx, %rax           # rval
   ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	**t1**, **t2**, **rval**
%rdx	**t4**
%rcx	**t5**

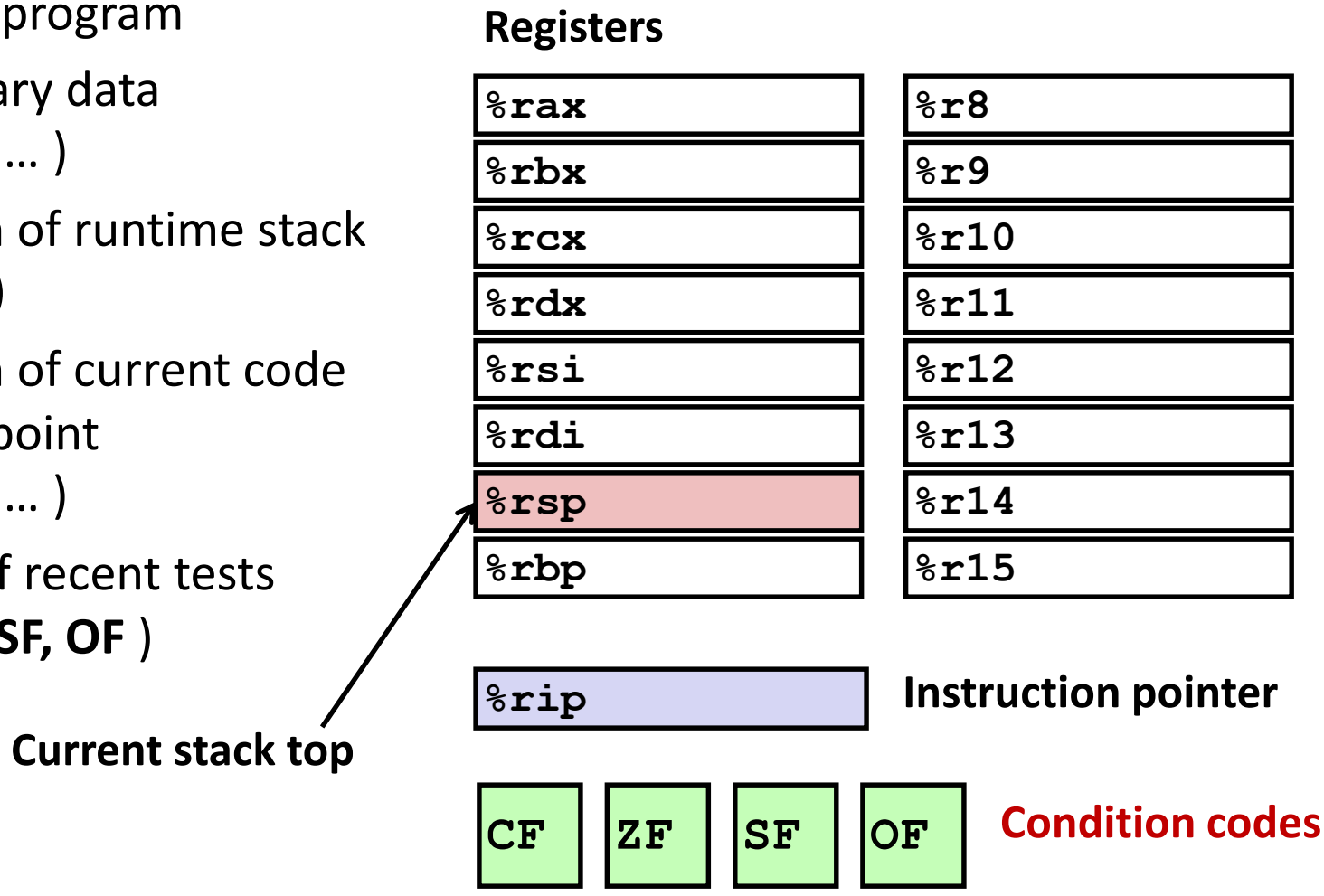
```
long arith(long x, long y, long z){
long t1 = x ^ y;
long t2= z*48;
long t3= t1 & 0x0F0F0F0F;
long t4= t2-t3;
return t4;
} // Assume x in %rdi, y in %rsi , and z in %rdx
arith:
    xorq %rsi, %rdi      long t1 = x ^ y;
    leaq  (%rdx, %rdx,2), %rax      3*z
    salq   $4, %rax      t2 = 16*(3*z)=48*z
    andl   $252645135, %edi     t3 = t1 & 0x0F0F0F0F
    subq   %rdi, %rax       return t2-t3;
    ret
```

Machine Programming I: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- C, assembly, machine code
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- Assembly Basics: Registers, operands, move
  - The x86-64 move instructions cover wide range of data movement forms
- Arithmetic
  - C compiler will figure out different instruction combinations to carry out computation

Processor State (x86-64, Partial)

- Information about currently executing program
  - Temporary data (**%rax**, …)
  - Location of runtime stack (**%rsp**)
  - Location of current code control point (**%rip**, …)
  - Status of recent tests (**CF, ZF, SF, OF**)



Condition Codes (Implicit Setting)

- **Single bit registers**
  - **CF** Carry Flag (for unsigned) **SF** Sign Flag (for signed)
  - **ZF** Zero Flag **OF** Overflow Flag (for signed)

- **Implicitly set (think of it as side effect) by arithmetic operations**

  Example: **`addq`** *Src*,*Dest* ↔ **`t = a+b`**

  CF set if carry out from most significant bit (unsigned overflow)

  ZF set if **`t == 0`**

  SF set if **`t < 0`** (as signed)

  OF set if two's-complement (signed) overflow

```
(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
```

- **Not set by `leaq` instruction**

Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - **`cmpq`** *Src2*, *Src1*
  - **`cmpq b,a`** like computing **`a-b`** without setting destination

  - **CF set** if carry out from most significant bit (used for unsigned comparisons)
  - **ZF set** if **`a == b`**
  - **SF set** if **`(a-b) < 0`** (as signed)
  - **OF set** if two's-complement (signed) overflow

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

Condition Codes (Explicit Setting: Test)

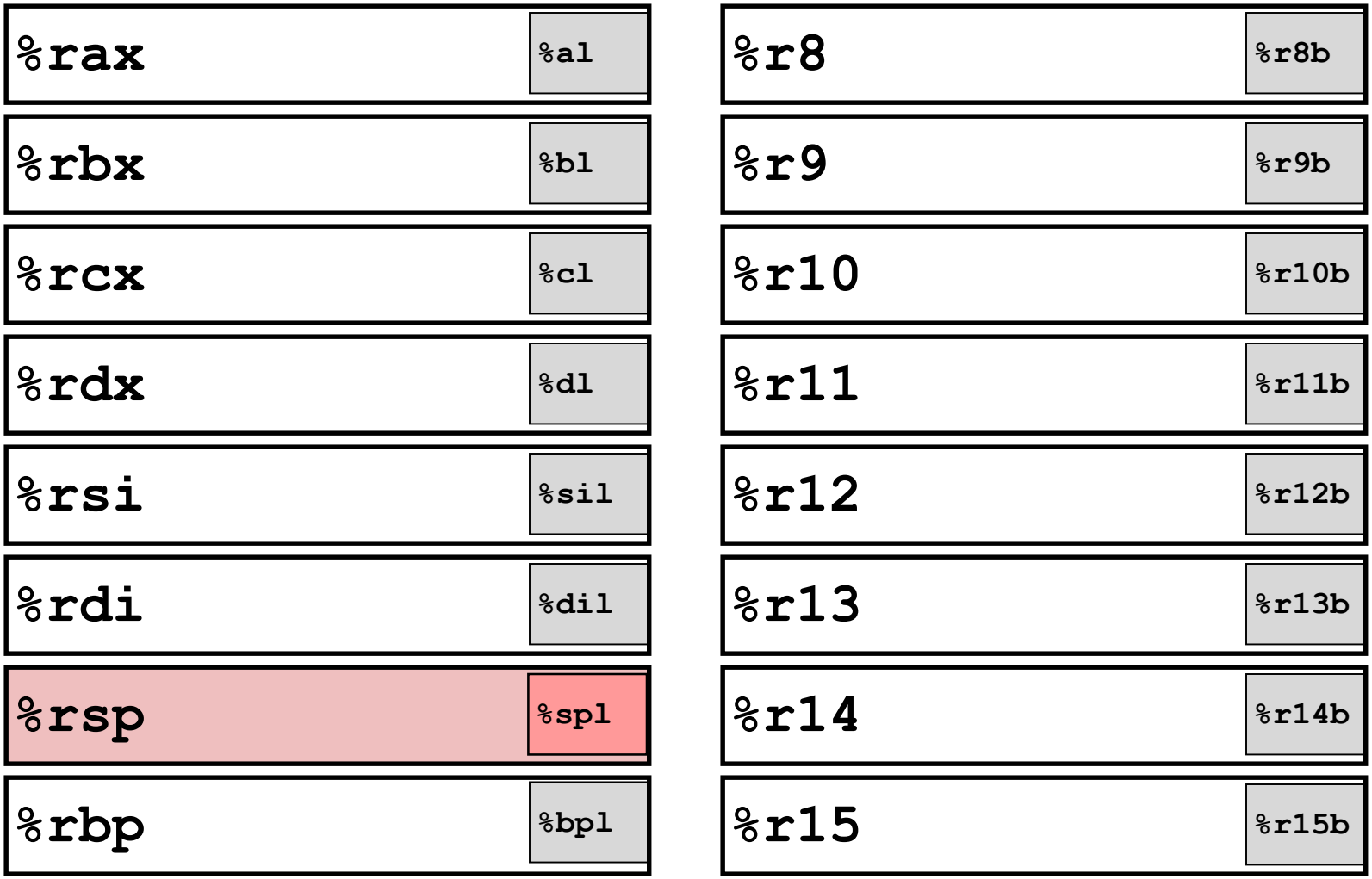
- **Explicit Setting by Test instruction**
  - **`testq`** *Src2*, *Src1*
    - **`testq b,a`** like computing **`a&b`** without setting destination
  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask
  - **ZF set** when **`a&b == 0`**
  - **SF set** when **`a&b < 0`**

Reading Condition Codes

- **SetX Instructions**
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

x86-64 Integer Registers



– Can reference low-order byte

Reading Condition Codes (Cont.)

- SetX Instructions:
  - Set single byte based on combination of condition codes
- One of addressable byte registers
  - Does not alter remaining bytes
  - Typically use **`movzbl`** to finish job
    - 32-bit instructions also set upper 32 bits to 0

```
int gt (long x, long y)
{
  return x > y;
}
```

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rax`	Return value

```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al           # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Conditional Branch Example (Old Style)

- Generation

```
gcc -Og -S -fno-if-conversion control.c
```

```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi   # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                     # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Expressing with Goto Code

- C allows **`goto`** statement
- Jump to position designated by label

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j
  (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

General Conditional Expression Translation (Using Branches)

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
  ntest = !Test;
  if (ntest) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

Using Conditional Moves

- Conditional Move Instructions
  - Instruction supports:
    if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be safe
- Why?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

C Code

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Goto Version

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

Conditional Move Example

```
long absdiff
   (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
absdiff:
    movq    %rdi, %rax  # x
    subq    %rsi, %rax  # result = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx  # eval = y-x
    cmpq    %rsi, %rdi  # x:y
    cmovle  %rdx, %rax  # if <=, result = eval
    ret
```

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

“Do-While” Loop Example

C Code

```
long pcount_do
   (unsigned long x) {
   long result = 0;
   do {
     result += x & 0x1;
     x >>= 1;
   } while (x);
   return result;
}
```

Goto Version

```
long pcount_goto
   (unsigned long x) {
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
   return result;
}
```

- Count number of 1’s in argument `x` (“popcount”)
- Use conditional branch to either continue looping or to exit loop

“Do-While” Loop Compilation

Goto Version

```
long pcount_goto
   (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```
   movl    $0, %eax    #  result = 0
.L2:                   # loop:
   movq    %rdi, %rdx
   andl    $1, %edx    #  t = x & 0x1
   addq    %rdx, %rax  #  result += t
   shrq    %rdi        #  x >>= 1
   jne     .L2         #  if (x) goto loop
   rep; ret
```

General "Do-While" Translation

C Code

```
do
   Body
   while (Test);
```

Goto Version

```
loop:
   Body
   if (Test)
      goto loop
```

- Body:

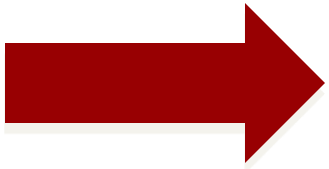
```
{
   Statement1;
   Statement2;
      …
   Statementn;
}
```

General “While” Translation #1

- “Jump-to-middle” translation
- Used with **-Og**

While version

```
while (Test)
  Body
```



Goto Version

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

While Loop Example #1

C Code

```
long pcount_while
   (unsigned long x) {
   long result = 0;
   while (x) {
     result += x & 0x1;
     x >>= 1;
   }
   return result;
}
```

Jump to Middle

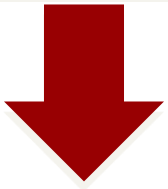
```
long pcount_goto_jtm
   (unsigned long x) {
   long result = 0;
   goto test;
 loop:
   result += x & 0x1;
   x >>= 1;
 test:
   if(x) goto loop;
   return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

General “While” Translation #2

While version

```
while (Test)
  Body
```



Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

- “Do-while” conversion
- Used with **–O1**

While Loop Example #2

C Code

```
long pcount_while
   (unsigned long x) {
   long result = 0;
   while (x) {
     result += x & 0x1;
     x >>= 1;
   }
   return result;
}
```

Do-While Version

```
long pcount_goto_dw
   (unsigned long x) {
   long result = 0;
   if (!x) goto done;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
 done:
   return result;
}
```

- Compare to do-while version of function
- Initial conditional guards entrance to loop

“For” Loop Form

General Form

```
for (Init; Test; Update)
    Body
```

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

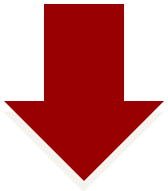
Body

```
{
  unsigned bit =
      (x >> i) & 0x1;
  result += bit;
}
```

“For” Loop → While Loop

For Version

```
for (Init; Test; Update)
    Body
```



While Version

```
Init;
while (Test) {
    Body
    Update;
}
```

For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
long pcount_for_while
   (unsigned long x)
{
   size_t i;
   long result = 0;
   i = 0;
   while (i < WSIZE)
   {
      unsigned bit =
         (x >> i) & 0x1;
      result += bit;
      i++;
   }
   return result;
}
```

“For” Loop Do-While Conversion

C Code

```
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

- Initial test can be optimized away

Goto Version

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               ! Test
    goto done;
 loop:
  {
    unsigned bit =
      (x >> i) & 0x1;             Body
    result += bit;
  }
  i++;                            Update
  if (i < WSIZE)                  Test
    goto loop;
 done:
  return result;
}
```

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

Jump Table Structure

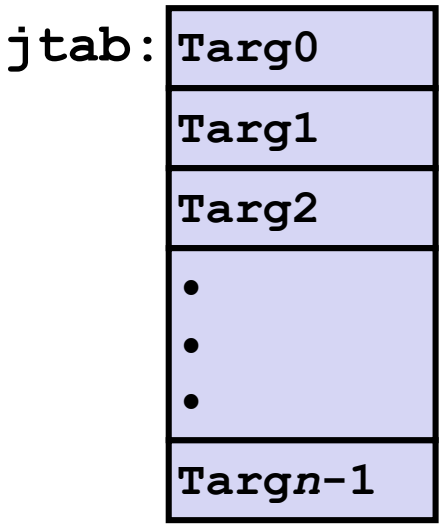
Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

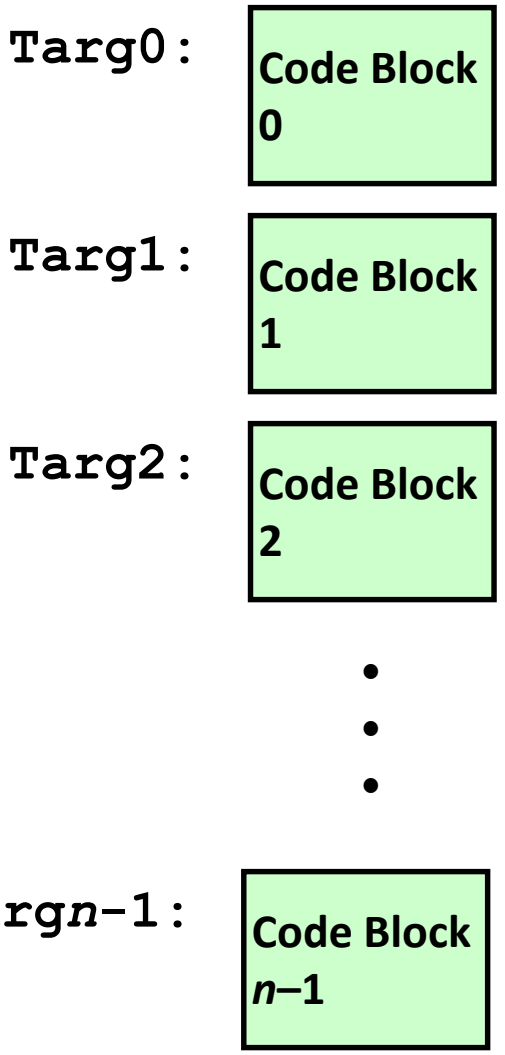
Translation (Extended C)

```
goto *JTab[x];
```

Jump Table



Jump Targets



Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja      .L8
    jmp     *.L4(,%rdi,8)
```

What range of values takes default?

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	Return value

Note that **w** not initialized here

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

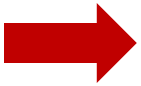
Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

Indirect jump



Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes
  - Base address at **`.L4`**

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

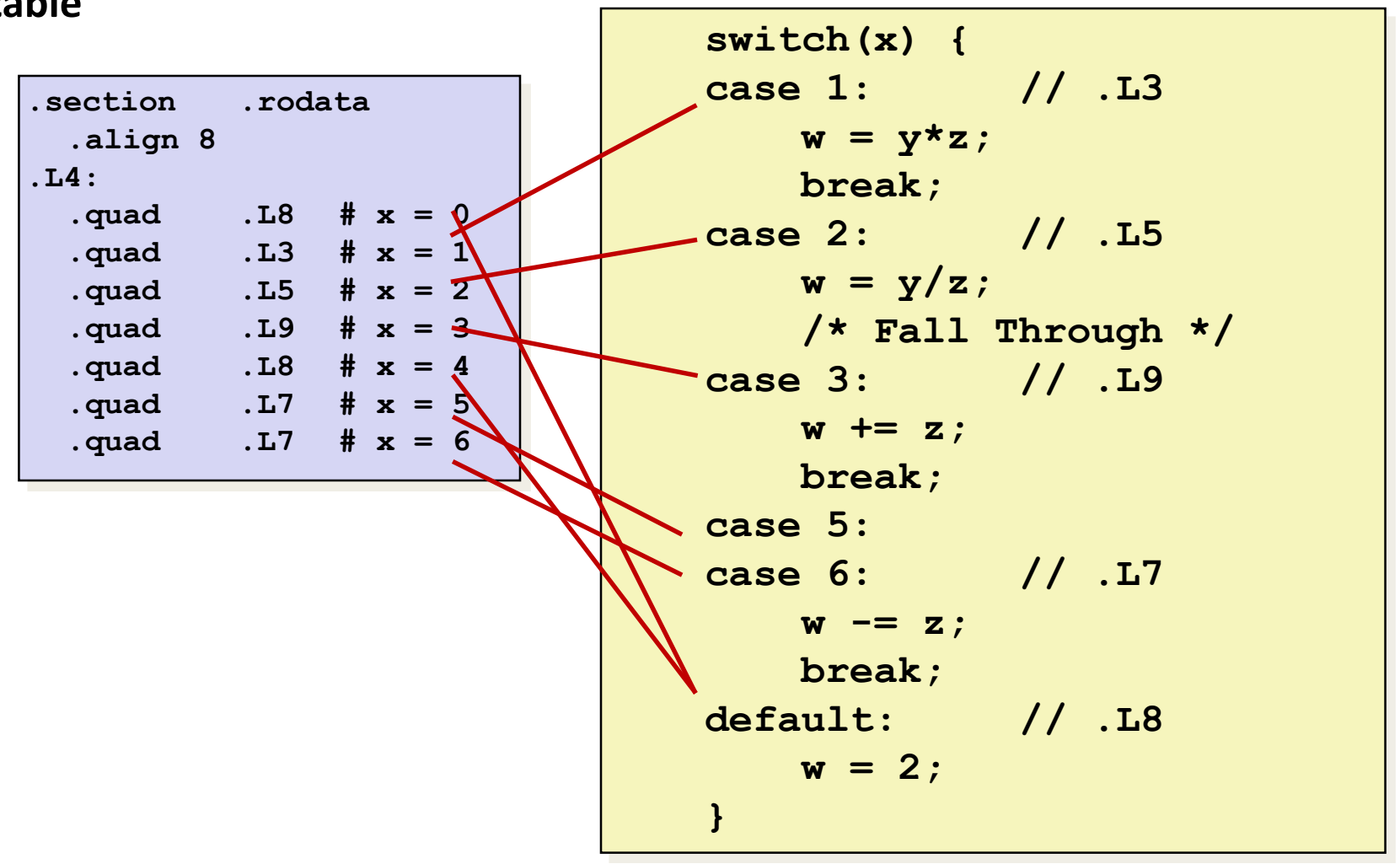
- Jumping
  - **Direct:** **`jmp .L8`**
  - Jump target is denoted by label **`.L8`**
  - **Indirect:** **`jmp *.L4(,%rdi,8)`**
  - Start of jump table: **`.L4`**
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address **`.L4 + x*8`**
    - Only for $0 \le \mathbf{x} \le 6$

Jump Table

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

```
    switch(x) {
    case 1:       // .L3
        w = y*z;
        break;
    case 2:       // .L5
        w = y/z;
        /* Fall Through */
    case 3:       // .L9
        w += z;
        break;
    case 5:
    case 6:       // .L7
        w -= z;
        break;
    default:      // .L8
        w = 2;
    }
```



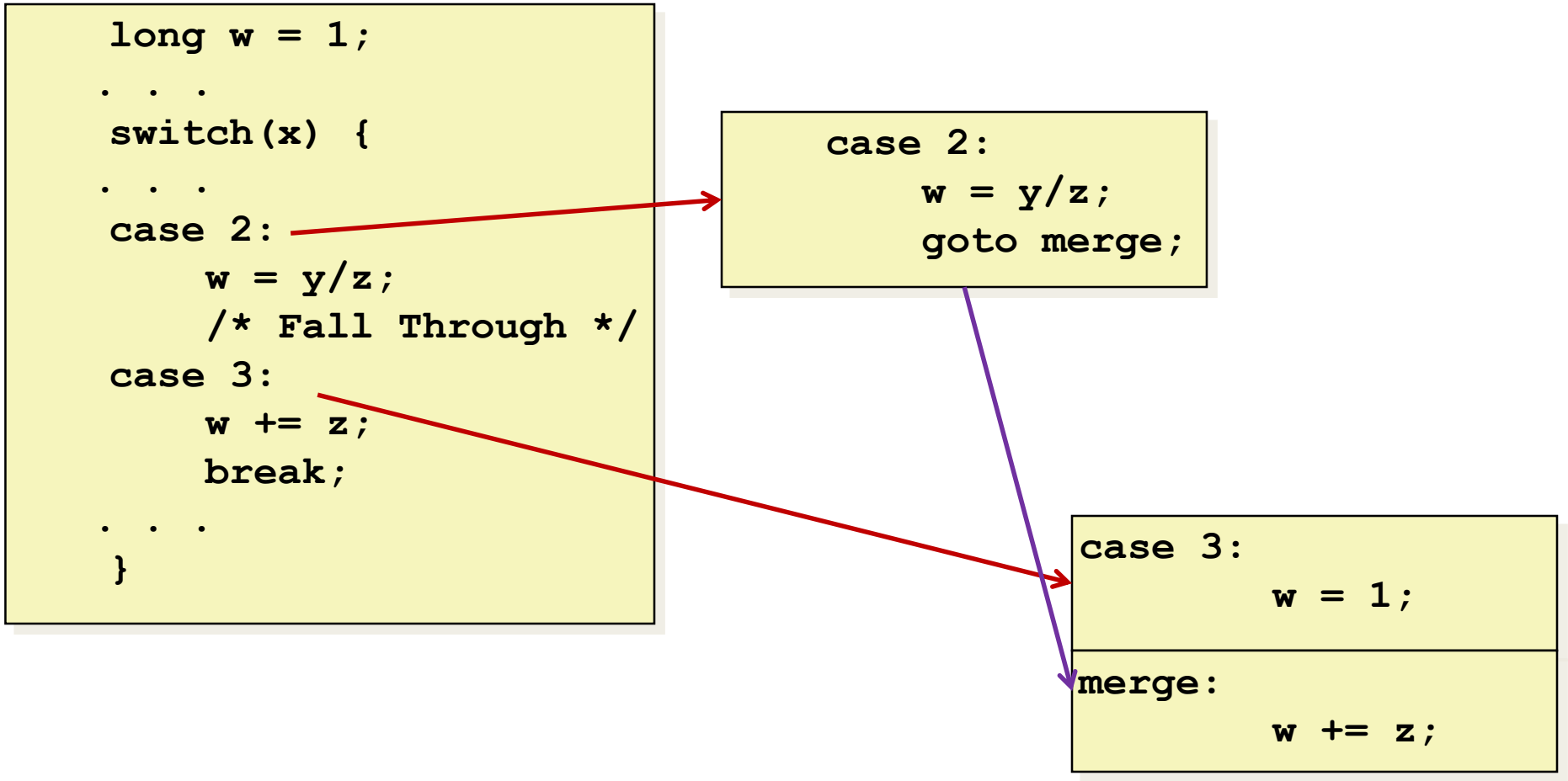
Code Blocks (x == 1)

```
  switch(x) {
  case 1:    // .L3
        w = y*z;
        break;
    . . .
}
```

```
.L3:
   movq     %rsi, %rax   # y
   imulq    %rdx, %rax   # y*z
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Handling Fall-Through



Code Blocks (x == 2, x == 3)

```
    long w = 1;
. . .
    switch(x) {
. . .
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
. . .
    }
```

```
.L5:                        # Case 2
   movq     %rsi, %rax
   cqto
   idivq    %rcx            #  y/z
   jmp      .L6             #  goto merge
.L9:                        # Case 3
   movl     $1, %eax        #  w = 1
.L6:                        # merge:
   addq     %rcx, %rax      #  w += z
   ret
```

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rdx`	Argument **z**
`%rax`	Return value

Code Blocks (x == 5, x == 6, default)

```
  switch(x) {
    . . .
    case 5:  // .L7
    case 6:  // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

```
.L7:                    # Case 5,6
  movl   $1, %eax       #  w = 1
  subq   %rdx, %rax     #  w -= z
  ret
.L8:                    # Default:
  movl   $2, %eax       #  2
  ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Summarizing

- C Control
  - if-then-else
  - do-while
  - while, for
  - switch
- Assembler Control
  - Conditional jump
  - Conditional move
  - Indirect jump (via jump tables)
  - Compiler generates code sequence to implement more complex control
- Standard Techniques
  - Loops converted to do-while or jump-to-middle form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (if-elseif-elseif-else)