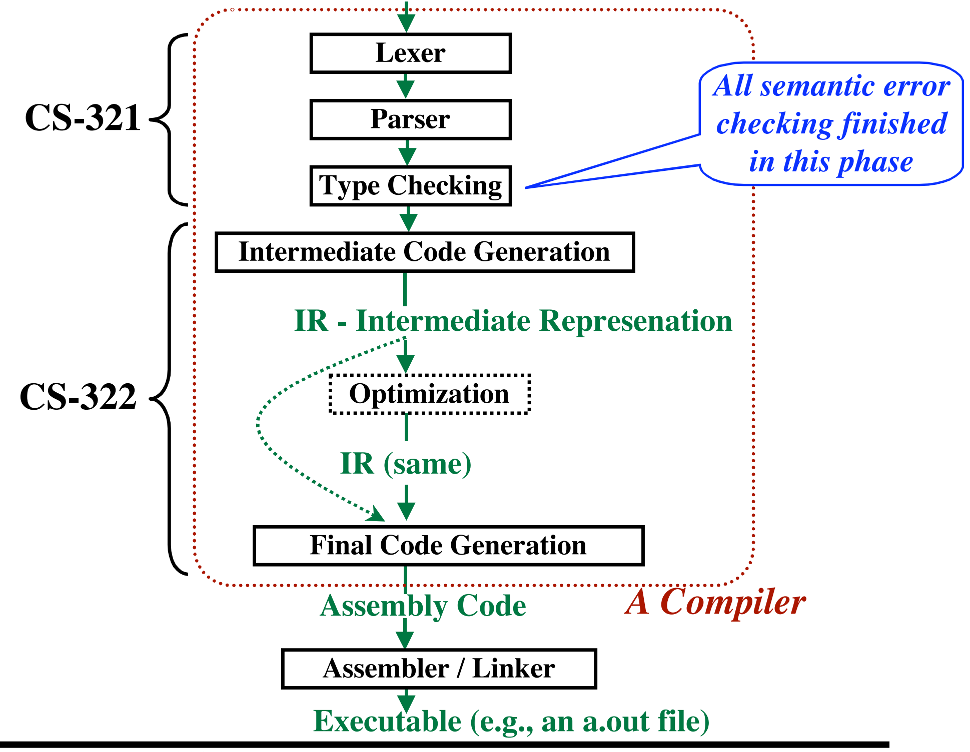
The SPARC Computer Architecture

Harry Porter
Portland State University



CS-321

CS-322

Lexer

Parser

Type Checking

All semantic error checking finished in this phase

Intermediate Code Generation

IR - Intermediate Represenation

Optimization

IR (same)

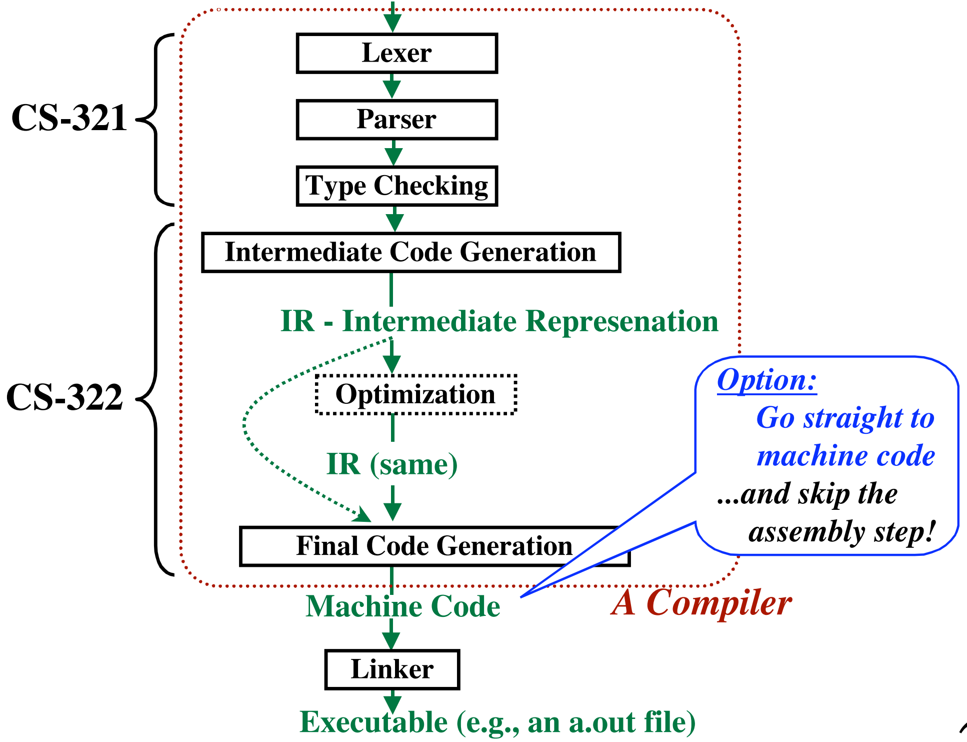
Final Code Generation

Assembly Code

A Compiler

Assembler / Linker

Executable (e.g., an a.out file)



CS-321

Lexer

Parser

Type Checking

CS-322

Intermediate Code Generation

IR - Intermediate Represenation

Optimization

IR (same)

Final Code Generation

Machine Code

Linker

Executable (e.g., an a.out file)

A Compiler

Option:
Go straight to machine code
...and skip the assembly step!

Machine Code

```
10000010000000000100000000000011
11000110001000000010000001010101
...
```

Assembly Code

```
add    %g1,%g2,%g3
st     %g3,myLocation
...
```

Each architecture has its own Machine Code / Assembly Code
Close correspondence
...almost 1-to-1

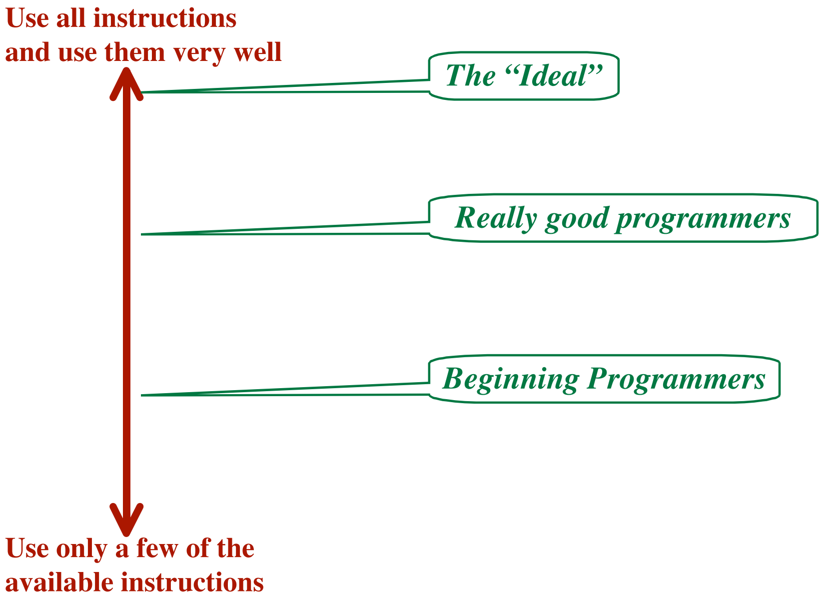
Machine code:

The bits that get loaded into memory
The CPU interprets these bits as instructions

Assembly code:

Easier for humans to work with
The "*assembler*" translates assembly code into machine code

A Spectrum



Use all instructions
and use them very well

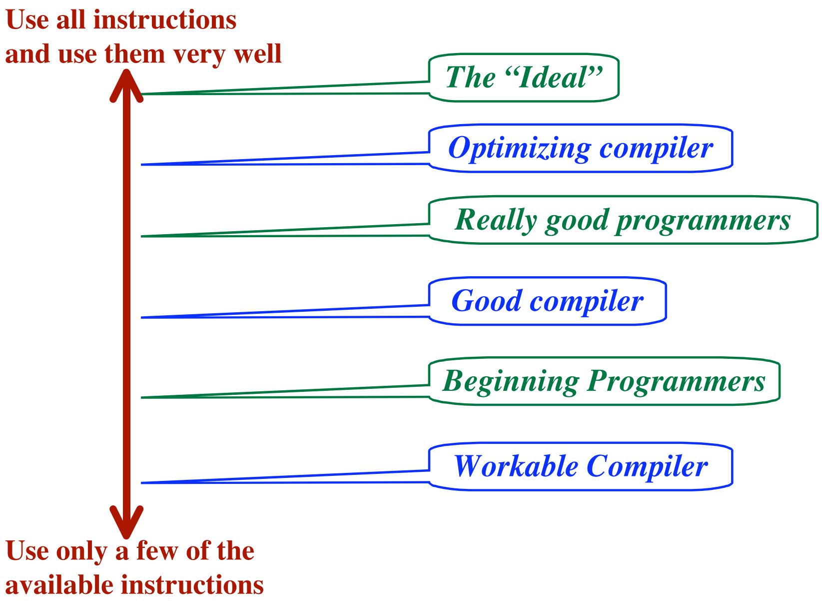
The "Ideal"

Really good programmers

Beginning Programmers

Use only a few of the
available instructions

A Spectrum



Use all instructions
and use them very well

The "Ideal"

Optimizing compiler

Really good programmers

Good compiler

Beginning Programmers

Workable Compiler

Use only a few of the
available instructions

SPARC Basics

Memory is byte addressable

Instructions are 4 bytes (32 bits)

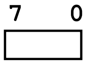
Addresses are 4 bytes (32 bits)

The CPU is always executing in either
- “System Mode” (or “supervisor mode”)
  Special instructions (load page table, perform I/O, other OS stuff...)
- “User Mode” (or “program mode”)
  Compiler-generated code does not include system instructions

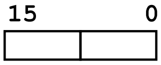
We’ll cover a subset of the SPARC instructions.

Basic Concepts

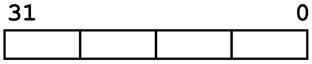
Byte
8 bits



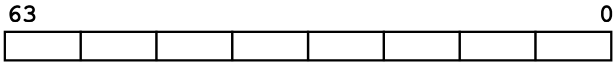
Halfword
16 bits = 2 bytes



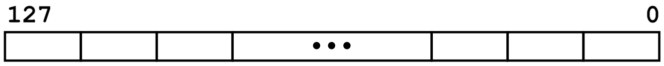
Word
32 bits = 4 bytes



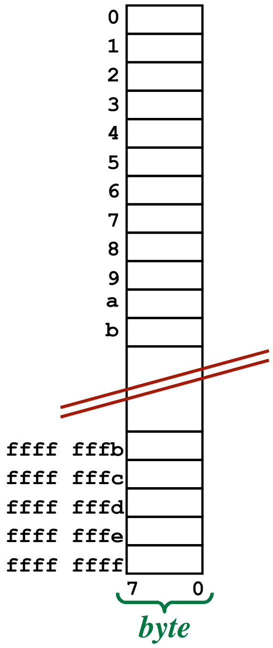
Doubleword
64 bits = 8 bytes



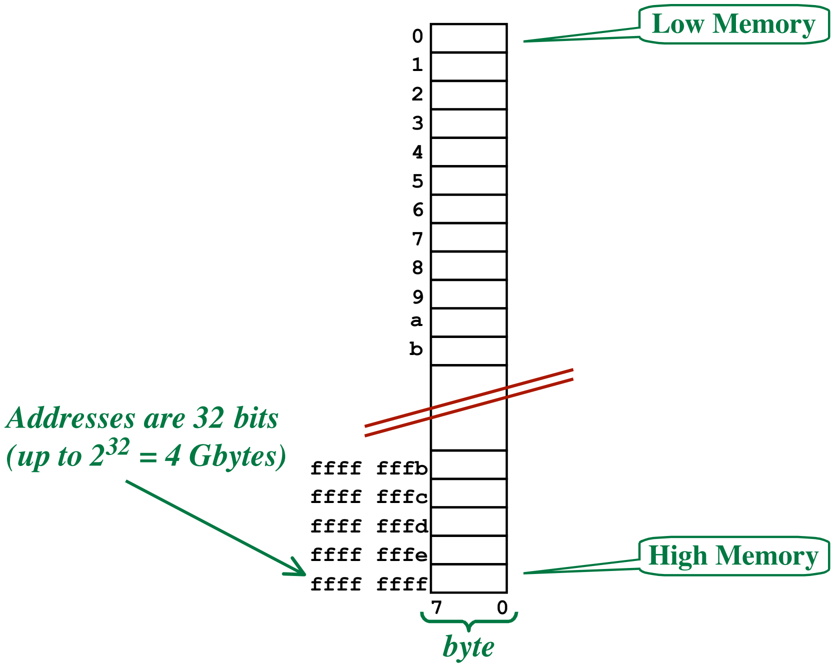
Quadword
128 bits = 16 bytes



Main Memory Organization

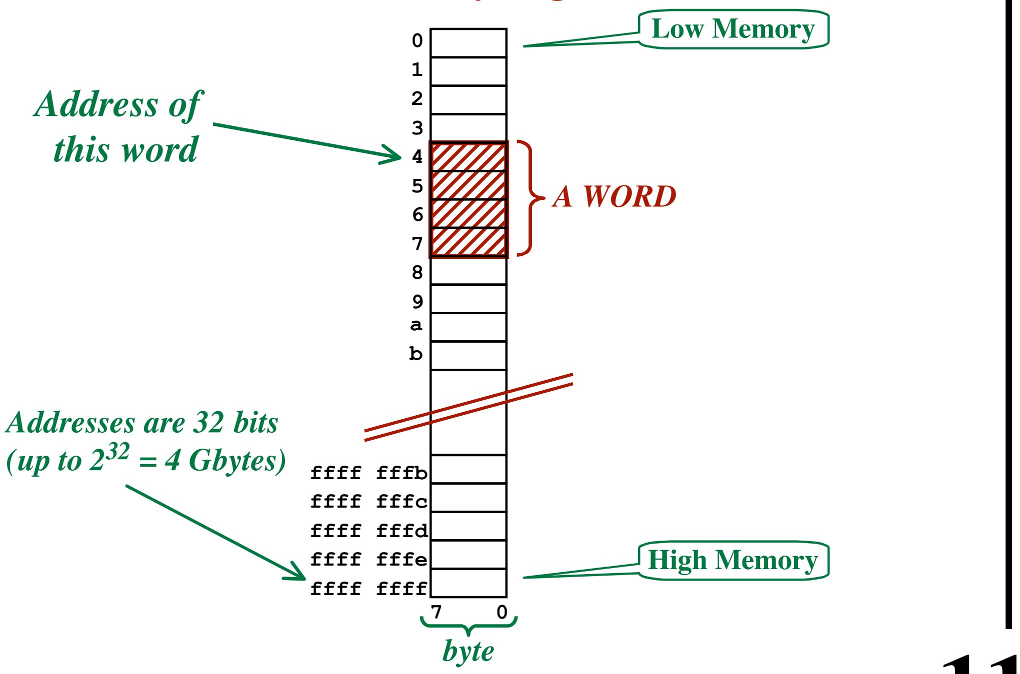


Main Memory Organization



Addresses are 32 bits (up to 2^{32} = 4 Gbytes)

Main Memory Organization



Low Memory

Address of this word

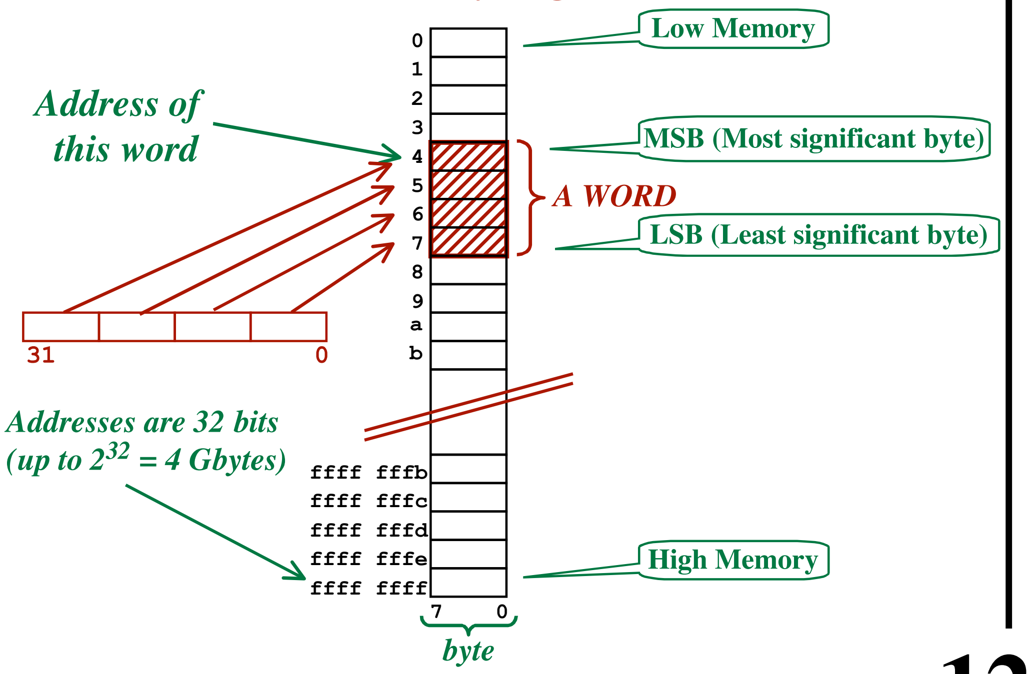
A WORD

Addresses are 32 bits (up to $2^{32} = 4$ Gbytes)

High Memory

byte

Main Memory Organization



Low Memory

Address of this word

MSB (Most significant byte)

A WORD

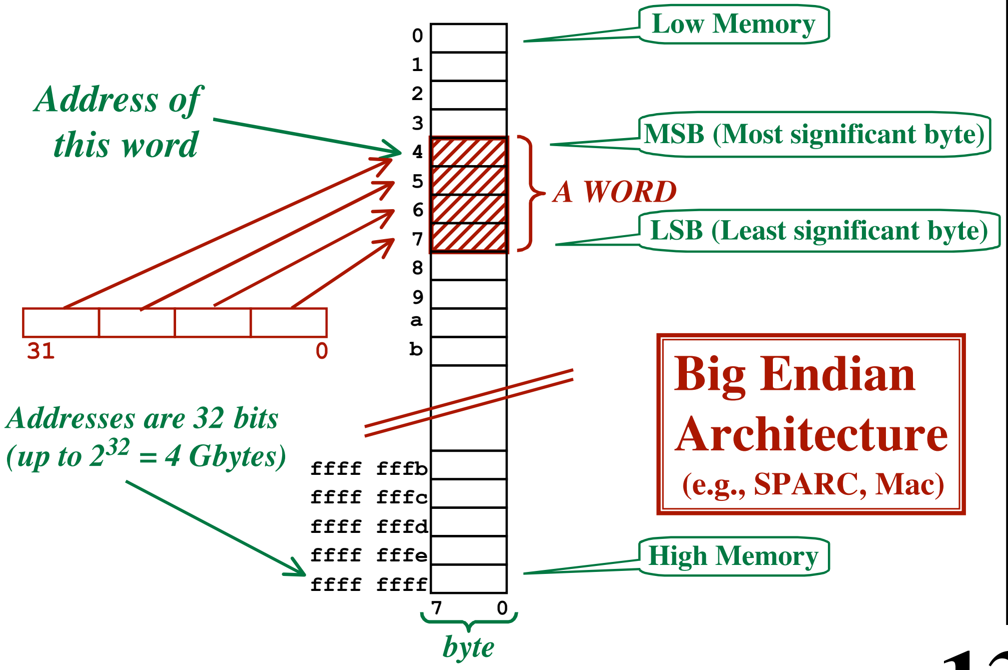
LSB (Least significant byte)

Addresses are 32 bits (up to $2^{32} = 4$ Gbytes)

High Memory

byte

Main Memory Organization



Address of this word

Addresses are 32 bits (up to 2^{32} = 4 Gbytes)

Low Memory

MSB (Most significant byte)

A WORD

LSB (Least significant byte)

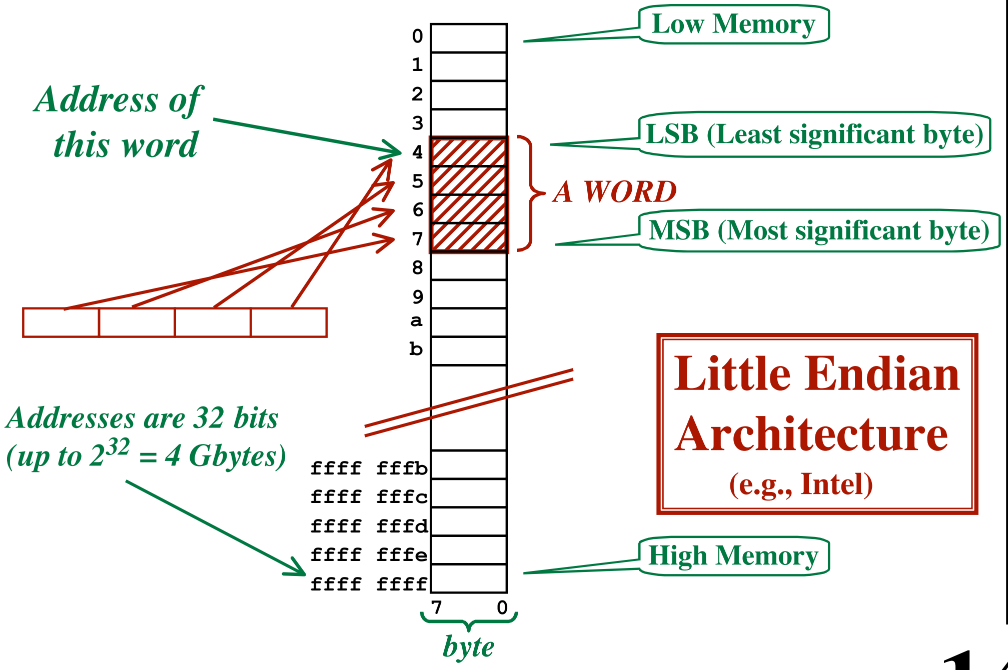
High Memory

byte

Big Endian Architecture

(e.g., SPARC, Mac)

Main Memory Organization



Address of this word

Addresses are 32 bits (up to 2^{32} = 4 Gbytes)

Low Memory

LSB (Least significant byte)

A WORD

MSB (Most significant byte)

High Memory

byte

Little Endian Architecture

(e.g., Intel)

Data Alignment

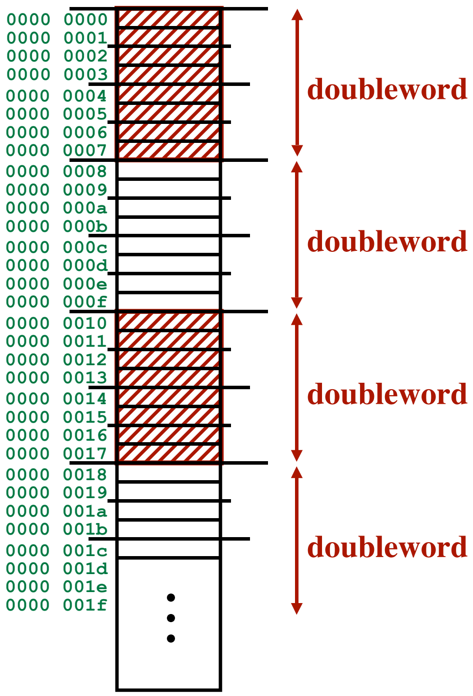
Data stored in memory must be “aligned”
according to the length of the data

Byte Data
can go at any address

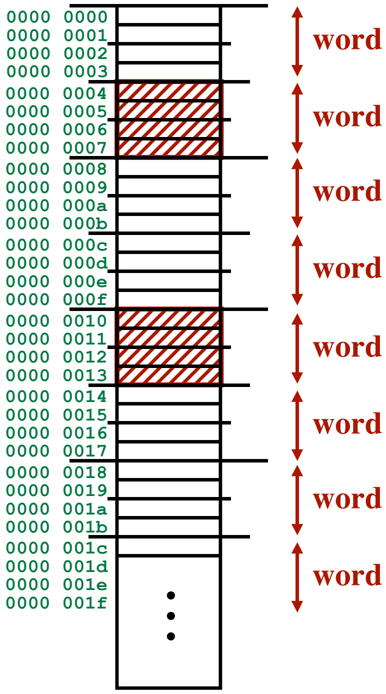
Halfword Data
must be “halfword aligned”
addresses must be even numbers

Word Data
must be “word aligned”
addresses must be divisible by 4

Doubleword Data
must be “doubleword aligned”
addresses must be divisible by 8



Doubleword addresses end in a number divisible by 8:
0, 8
In binary:
---- ---- ---- ---- ---- ---- ---- -000

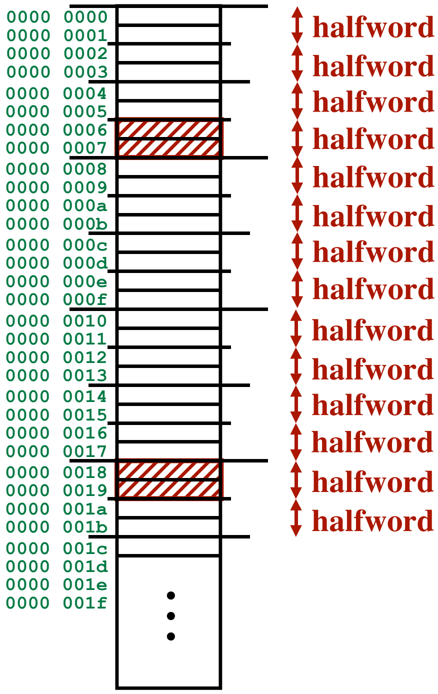


Word addresses end in a number divisible by 4:

0, 4, 8, c

In binary:

---- ---- ---- ---- ---- ---- ---- --00



Halfword addresses end in an even number:

0, 2, 4, 6, 8, a, c, e

In binary:

---- ---- ---- ---- ---- ---- ---- ---0

Decimal Number Representation

Example:

4037

$= 4000 + 30 + 7$

$= \ldots + 0\cdot 10000 + 4\cdot 1000 + 0\cdot 100 + 3\cdot 10 + 7\cdot 1$

$= \ldots + 0\cdot 10^4 + 4\cdot 10^3 + 0\cdot 10^2 + 3\cdot 10^1 + 7\cdot 10^0$

Base 10:

$\ldots + X\cdot 10^4 + X\cdot 10^3 + X\cdot 10^2 + X\cdot 10^1 + X\cdot 10^0$

Set of numerals (the “digits”):

{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

Hexadecimal Number Representation

Base 16:

$\ldots + X\cdot 16^4 + X\cdot 16^3 + X\cdot 16^2 + X\cdot 16^1 + X\cdot 16^0$

$\ldots + X\cdot 65536 + X\cdot 4096 + X\cdot 256 + X\cdot 16 + X\cdot 1$

Set of numerals:

{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F }

Example:

3A0F

$= \ldots + 0\cdot 16^4 + 3\cdot 16^3 + A\cdot 16^2 + 0\cdot 16^1 + F\cdot 16^0$

$= \ldots + 0\cdot 65536 + 3\cdot 4096 + A\cdot 256 + 0\cdot 16 + F\cdot 1$

$= \ldots + 0\cdot 65536 + 3\cdot 4096 + 10\cdot 256 + 0\cdot 16 + 15\cdot 1$

= 12,288 + 2,560 + 15 = 14,863 (in decimal)

Binary Number Representation

Base 2:

$\ldots + X \cdot 2^5 + X \cdot 2^4 + X \cdot 2^3 + X \cdot 2^2 + X \cdot 2^1 + X \cdot 2^0$

$\ldots + X \cdot 32 + X \cdot 16 + X \cdot 8 + X \cdot 4 + X \cdot 2 + X \cdot 1$

Set of numerals:

{ 0, 1}

Example:

110101

$= \ldots + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

$= \ldots + 1 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1$

$= 32 + 16 + 4 + 1$

$= 53$ (in decimal)

One-to-one correspondence between hex and binary;

3	A	0	F
0011	1010	0000	1111

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

Byte (8 bits)

Hex: `3A`

Binary: `0011 1010`

Halfword (16 bits)

Hex: `3A0F`

Binary: `0011 1010 0000 1111`

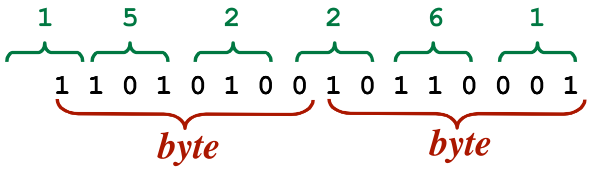
Word (32 bits)

Hex: `3AOF 12D8`

Binary: `0011 1010 0000 1111 0001 0010 1101 1000`

Octal Notation

Bad match with byte alignment



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

The numbers get too long.
  Word (32 bits)
    Octal: `12305570426`
    Hex: `3A0F 12D8`

Every octal looks like a decimal number (and often they get confused).

$263_8 = 179_{10}$
$263_{10} = 263_{10}$
$263_{16} = 611_{10}$

C Notation for octals (leading zero is significant!)

`0263`

Unsigned Number Representation

Example: 8-bits

Always non-negative
  0,1,2, ... 255
  0,1,2, ... 2^8-1

Value (in decimal)	Binary	Hex
0	0000 0000	00
1	0000 0001	01
2	0000 0010	02
3	0000 0011	03
4	0000 0100	04
5	0000 0101	05
6	0000 0110	06
7	0000 0111	07
...	...	...
252	1111 1100	FC
253	1111 1101	FD
254	1111 1110	FE
255	1111 1111	FF

Unsigned Number Representation

Example: 32-bits

Always non-negative

0,1,2, ... 4,294,967,295

0,1,2, ... 2^{32}-1

Value (in decimal)	Binary	Hex
0	0000 0000 0000 0000 0000 0000 0000 0000	0000 0000
1	0000 0000 0000 0000 0000 0000 0000 0001	0000 0001
2	0000 0000 0000 0000 0000 0000 0000 0010	0000 0002
3	0000 0000 0000 0000 0000 0000 0000 0011	0000 0003
4	0000 0000 0000 0000 0000 0000 0000 0100	0000 0004
5	0000 0000 0000 0000 0000 0000 0000 0101	0000 0005
6	0000 0000 0000 0000 0000 0000 0000 0110	0000 0006
7	0000 0000 0000 0000 0000 0000 0000 0111	0000 0007
...	...	...
4,294,967,292	1111 1111 1111 1111 1111 1111 1111 1100	FFFF FFFC
4,294,967,293	1111 1111 1111 1111 1111 1111 1111 1101	FFFF FFFD
4,294,967,294	1111 1111 1111 1111 1111 1111 1111 1110	FFFF FFFE
4,294,967,295	1111 1111 1111 1111 1111 1111 1111 1111	FFFF FFFF

Unsigned Number Representation

Largest Number Representable

Byte (8-bits)

2^{8}-1

= 255

= FF (in hex)

Halfword (16-bits)

2^{16}-1

= 65,535

= 64K - 1

= FFFF (in hex)

Word (32-bits)

2^{32}-1

= 4,294,967,295

= 4G -1

= FFFF FFFF (in hex)

“Two’s complement” number representation

Signed Number Representation

Example: 8-bits

Binary	Hex	Unsigned Value	Signed Value	
0000 0000	00	0	0	
0000 0001	01	1	1	
0000 0010	02	2	2	
...	...	...	...	
0111 1101	7D	125	125	2^7-3
0111 1110	7E	126	126	2^7-2
0111 1111	7F	127	127	2^7-1
1000 0000	80	128		
1000 0001	81	129		
1000 0010	82	130		
...	...	...		
1111 1101	FD	253		
1111 1110	FE	254		
1111 1111	FF	255		

“Two’s complement” number representation

Signed Number Representation

Example: 8-bits

Most significant bit
0 means ≥ zero (in hex: 0..7)
1 means < zero (in hex: 8..F)

Binary	Hex	Unsigned Value	Signed Value	
0000 0000	00	0	0	
0000 0001	01	1	1	
0000 0010	02	2	2	
...	...	...	...	
0111 1101	7D	125	125	2^7-3
0111 1110	7E	126	126	2^7-2
0111 1111	7F	127	127	2^7-1
1000 0000	80	128	-128	$-(2^7)$
1000 0001	81	129	-127	$-(2^7-1)$
1000 0010	82	130	-126	$-(2^7-2)$
...	...	...	...	
1111 1101	FD	253	-3	
1111 1110	FE	254	-2	
1111 1111	FF	255	-1	

“Two’s complement” number representation

Signed Number Representation

Example: 8-bits

Most significant bit
0 means ≥ zero (in hex: **0..7**)
1 means < zero (in hex: **8..F**)

Binary	Hex	Unsigned Value	Signed Value	
0000 0000	00	0	0	
0000 0001	01	1	1	
0000 0010	02	2	2	
...	...	...	...	
0111 1101	7D	125	125	2^7-3
0111 1110	7E	126	126	2^7-2
0111 1111	7F	127	127	2^7-1
1000 0000	80	128	-128	$-(2^7)$
1000 0001	81	129	-127	$-(2^7-1)$
1000 0010	82	130	-126	$-(2^7-2)$
...	...	...	...	
1111 1101	FD	253	-3	
1111 1110	FE	254	-2	
1111 1111	FF	255	-1	

Always one more negative number than positive numbers:

$$\underbrace{-128,\ \ldots\ ,\ -1,}_{2^7 = 128\text{ values}}\quad + \quad \underbrace{0,\ 1,\ \ldots\ +127}_{2^7 = 128\text{ values}}\quad = \quad 2^8 = 256\text{ values}$$

“Two’s complement” number representation

Signed Number Representation

Example: 32-bits

Binary	Hex	Unsigned Value	Signed Value	
0000...0000	0000 0000	0	0	
0000...0001	0000 0001	1	1	
0000...0010	0000 0002	2	2	
...	...	...	...	
0111...1101	7FFF FFFD	2,147,483,645	2,147,483,645	$2^{31}-3$
0111...1110	7FFF FFFE	2,147,483,646	2,147,483,646	$2^{31}-2$
0111...1111	7FFF FFFF	2,147,483,647	2,147,483,647	$2^{31}-1$
1000...0000	8000 0000	2,147,483,648	-2,147,483,648	$-(2^{31})$
1000...0001	8000 0001	2,147,483,649	-2,147,483,647	$-(2^{31}-1)$
1000...0010	8000 0002	2,147,483,650	-2,147,483,646	$-(2^{31}-1)$
...	...	...	...	
1111...1101	FFFF FFFD	4,294,967,294	-3	
1111...1110	FFFF FFFE	4,294,967,295	-2	
1111...1111	FFFF FFFF	4,294,967,296	-1	

Always one more negative number than positive numbers:

$$\underbrace{-2{,}147{,}483{,}648,\ \ldots\ ,\ -1,}_{2^{31}\text{ values}}\quad + \quad \underbrace{0,\ 1,\ \ldots\ +\ 2{,}147{,}483{,}647}_{2^{31}\text{ values}}\quad = \quad 2^{32}\text{ values}$$

Ranges of Numbers Using “Signed” Values

...in the “two’s complement” system of number representation:

	Total Number of Values	Largest Positive Number	Most Negative Number
Byte (8-bits)	2^8 256	2^7-1 127	-(2^7) -128
Halfword (16-bits)	2^{16} 64K 65,536	2^{15}-1 32K-1 32,767	-(2^{15}) -32K -32,768
Word (32-bits)	2^{32} 4G 4,294,967,296	2^{31}-1 2G-1 2,147,483,647	-(2^{31}) -2G -2,147,483,648

Addition

Decimal:

```
  1 1 1
    3 8 5 3
+   9 3 7 4
-----------
  1 3 2 2 7
```

Binary:

```
    1 1   1
    1 1 1 0 1 1 0 0
+   1 0 1 0 1 0 1 0
-------------------
  1 1 0 0 1 0 1 1 0
```

	0	1	2	3	4	5	6	7	8	9	10
0	0	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9		
2	2	3	4	5	6	7	8				
3	3	4	5	6	7	8					
4	4	5	6	7	8						
5	5	6	7	8				etc.			
6	6	7									
7	7										
8	8										
9											
10											

```
0 + 0 = 0
1 + 0 = 1
1 + 1 = 10
1 + 1 + 1 = 11
```

Addition:

The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

8-bit Unsigned:

```
     1110 1100   = 236
+    1010 1010   = 170
   1 1001 0110   = 406
```

8-bit Signed:

```
     1110 1100   = -20
+    1010 1010   = -86
   1 1001 0110   = -106
```

Overflow! (max value = 255)

Addition:

The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

8-bit Unsigned:

```
     1110 1100   = 236
+    1010 1010   = 170
   1 1001 0110   = 406
```

8-bit Signed:

```
     1110 1100   = -20
+    1010 1010   = -86
   1 1001 0110   = -106
```

Overflow! (max value = 255)

Subtraction:

The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

Addition:

The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

```
  8-bit Unsigned:                 8-bit Signed:
     1110 1100   = 236               1110 1100   = -20
 +   1010 1010   = 170           +   1010 1010   = -86
   1 1001 0110   = 406             1 1001 0110   = -106
```

Overflow! (max value = 255)

Subtraction:

The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

Multiplication:

Two algorithms.

```
8-bit Signed:                            8-bit Unsigned:
                1111 1110   = -2                        1111 1110   = 254
            x   1111 1110   = -2                    x   1111 1110   = 254
  0000 0000 0000 0100   = +4          1111 1100 0000 0100   = 64,516
```

(NOTE: Result may be twice as long as operands.)

Addition:

The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

8-bit Unsigned: 8-bit Signed:
     1110 1100   = 236   1110 1100   = -20
 +   1010 1010   = 170   +   1010 1010   = -86
   1 1001 0110   = 406   1 1001 0110   = -106

Overflow! (max value = 255)

Subtraction:

The algorithm is the same for SIGNED and UNSIGNED.
Overflow detection is slightly different.

Multiplication:

Two algorithms.

8-bit Signed: 8-bit Unsigned:
                1111 1110   = -2   1111 1110   = 254
            x   1111 1110   = -2   x   1111 1110   = 254
  0000 0000 0000 0100   = +4   1111 1100 0000 0100   = 64,516

(NOTE: Result may be twice as long as operands.)

Division:

Two algorithms.

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

```
                    0000 0010   =  2
complementing:
add 1:
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

```
                    0000 0010   =  2
complementing:      1111 1101
add 1:
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by "add 1"

Example:

```
                  0000 0010   =  2
complementing:    1111 1101
add 1:           +0000 0001
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by "add 1"

Example:

```
                  0000 0010   =  2
complementing:    1111 1101
add 1:           +0000 0001
                  1111 1110   = -2
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

```
                    0000 0010   =  2
   complementing:   1111 1101
   add 1:          +0000 0001
                    1111 1110   = -2
```

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

```
                    0000 0010   =  2
   complementing:   1111 1101
   add 1:          +0000 0001
                    1111 1110   = -2
```

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

8-Bit Example:

```
                    1000 0000   =  -128
   complementing:
   add 1:
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

```
                    0000 0010   =  2
   complementing:   1111 1101
   add 1:          +0000 0001
                    1111 1110   = -2
```

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

8-Bit Example:

```
                    1000 0000   =  -128
   complementing:   0111 1111
   add 1:
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by “add 1”

Example:

```
                    0000 0010   =  2
   complementing:   1111 1101
   add 1:          +0000 0001
                    1111 1110   = -2
```

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

8-Bit Example:

```
                    1000 0000   =  -128
   complementing:   0111 1111
   add 1:          +0000 0001
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by "add 1"

Example:

```
                      0000 0010   =  2
   complementing:     1111 1101
   add 1:            +0000 0001
                      1111 1110   = -2
```

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

8-Bit Example:

```
                      1000 0000   =  -128
   complementing:     0111 1111
   add 1:            +0000 0001
                      1000 0000   =  -128
```

Arithmetic Negation

The Algorithm to Negate a Signed Number:

Bitwise complement (i.e., logical NOT)
Followed by "add 1"

Example:

```
                      0000 0010   =  2
   complementing:     1111 1101
   add 1:            +0000 0001
                      1111 1110   = -2
```

Arithmetic negation can overflow!
Every signed number can be negated,
... except the most negative number.

8-Bit Example:

```
                      1000 0000   =  -128
   complementing:     0111 1111
   add 1:            +0000 0001
                      1000 0000   =  -128
```

The most negative 32-bit number, 0x80000000

```
Hex:       8    0    0    0       0    0    0    0
Binary:  1000 0000 0000 0000    0000 0000 0000 0000
Decimal: -2,147,483,648
```