Numerical Representation

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

Figure 2.2 **Hexadecimal notation.** Each hex digit encodes one of 16 values.

Binary

- **1+1 = 10**
- **1+11 = 100**

Binary	0001	0111	0011	1010	0100	1100

Hexadecimal

- Looks normal until 1+9 = a

Hexadecimal	1	7	3	A	4	C

Conversions

- 1111 = 0xf = 15
- Every 4 bits is one Hexadecimal place
- Easy mode conversion

Hexadecimal	1	7	3	A	4	C
Binary	0001	0111	0011	1010	0100	1100

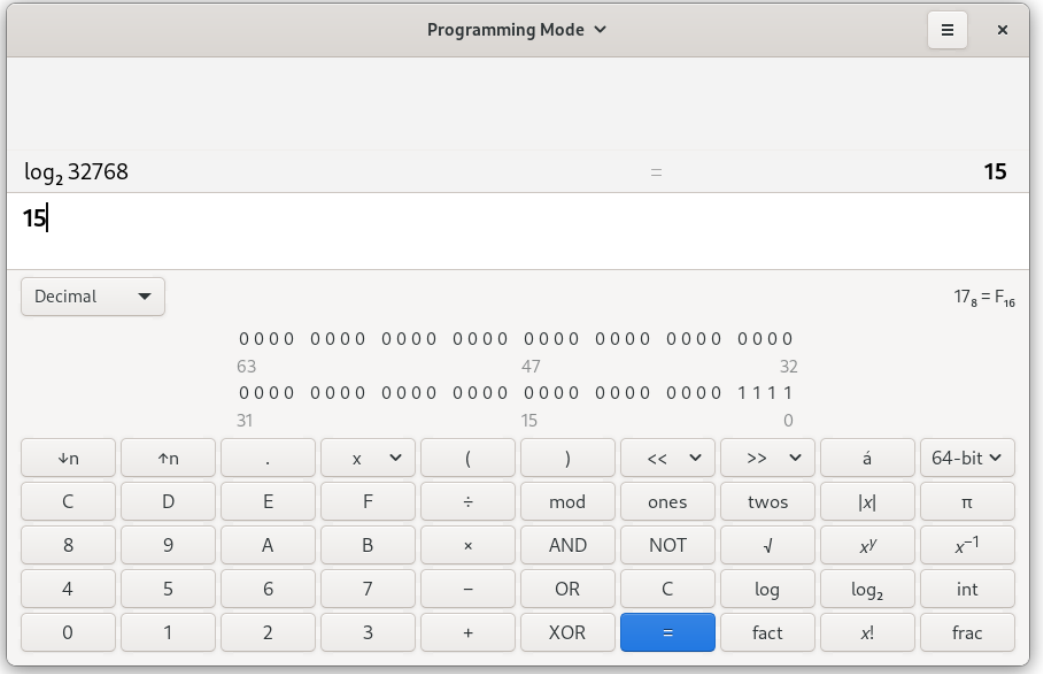
Just a few mental stretches

- Learn powers of 2 up to 2^32
- Remember how to log2

n	2^n (decimal)	2^n (hexadecimal)
5	32	0x20
23		
	32,768	
		0x2000
12		
	64	
		0x100

Programming Mode Calc

- Shows 64bits
- Octal and Hex



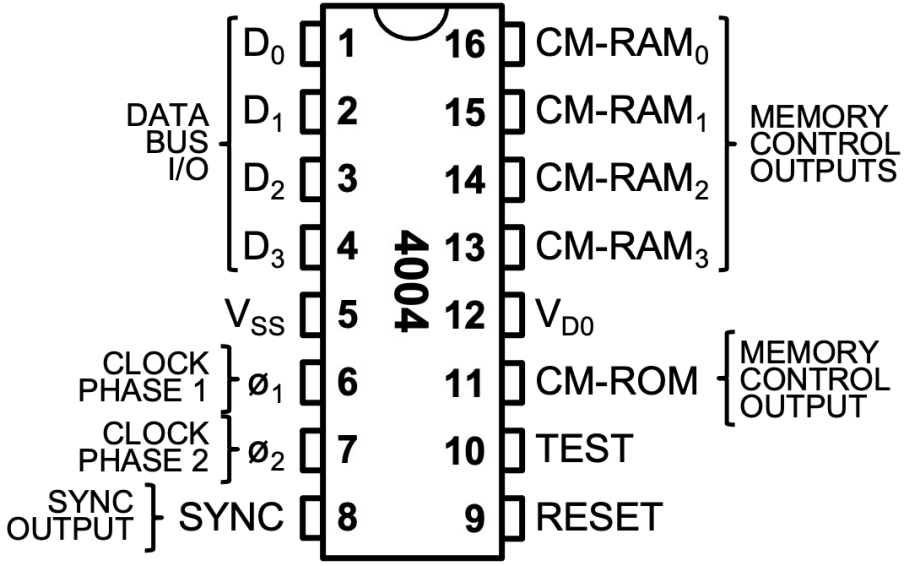
Converting to Binary and Hex

- 0x75 == N*16 + M
  - N = 7
  - M = 5
- 0x75 = (7 in binary) (5 in binary)
  - 7 = 0111 0101

Decimal	Binary	Hexadecimal
_____	_____	0x75
_____	_____	0xBD
_____	_____	0xF5

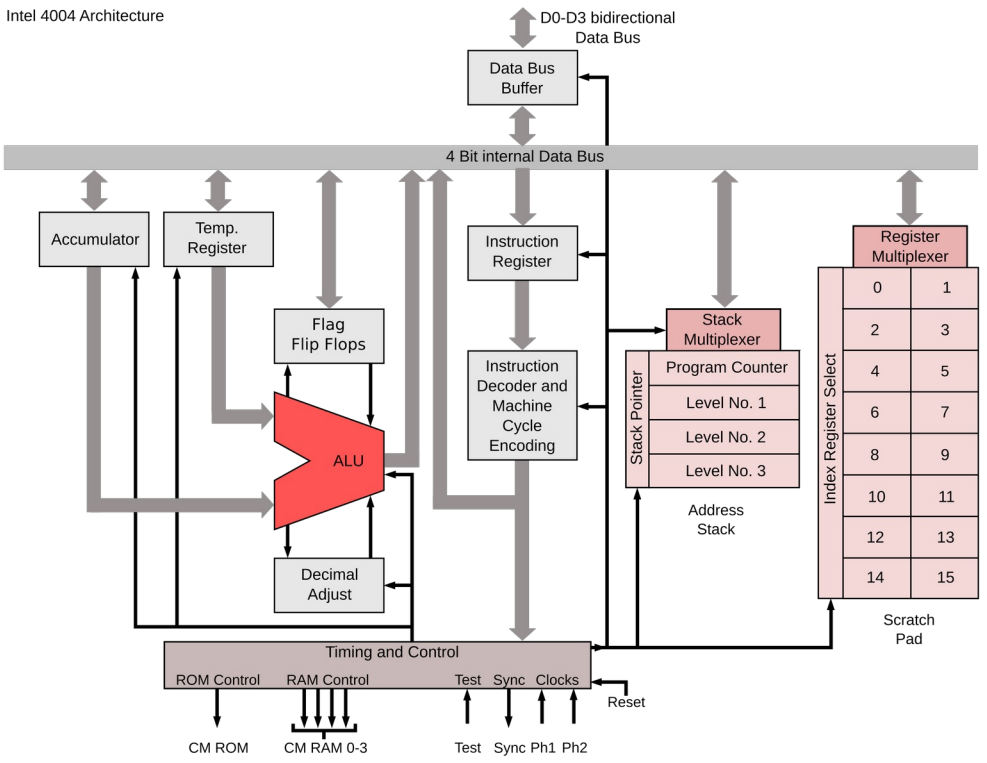
Overflow

- Size of variables
  - Hardware limitations
  - Intel 4004
  - 4 bit addressing



This should look familiar

- Data bus is 4 bits
- 0xff = two clock cycles



What happens?

- If memory is 4 bits wide:
  - 1111 + 1
  - 0000 - 1

32bit and 64bit worlds

C declaration		Bytes	
Signed	Unsigned	32-bit	64-bit
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

Figure 2.3 Typical sizes (in bytes) of basic C data types. The number of bytes allocated varies with how the program is compiled. This chart shows the values typical of 32-bit and 64-bit programs.

Size of things

- “/usr/include/bits/types.h”
- Defines common types from C into more useable types for linux
- Use sizeof() to determine size in bytes.

Aside: time64_t

- Is this bad?
- Why did y2k happen?
  - 1980 stored as “80” not “xx80”
  - Not strictly overflow
- Epochalypse/2038 Problem

Byte Ordering

- Big Endian
  - Older hardware you don’t care about
  - The Network, which you do care about
- Little Endian
  - X86, ARM, most things

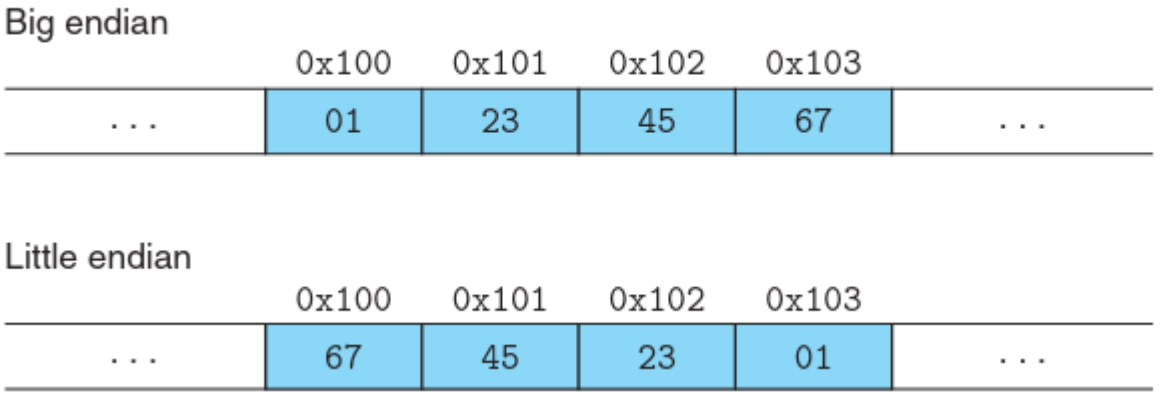
Byte Ordering: Continued

- Big Endian Sun
- 32bit Little Endian
- 64bit Little endian

Machine	Value	Type	Bytes (hex)
Linux 32	12,345	`int`	`39 30 00 00`
Windows	12,345	`int`	`39 30 00 00`
Sun	12,345	`int`	`00 00 30 39`
Linux 64	12,345	`int`	`39 30 00 00`
Linux 32	12,345.0	`float`	`00 e4 40 46`
Windows	12,345.0	`float`	`00 e4 40 46`
Sun	12,345.0	`float`	`46 40 e4 00`
Linux 64	12,345.0	`float`	`00 e4 40 46`
Linux 32	`&ival`	`int *`	`e4 f9 ff bf`
Windows	`&ival`	`int *`	`b4 cc 22 00`
Sun	`&ival`	`int *`	`ef ff fa 0c`
Linux 64	`&ival`	`int *`	`b8 11 e5 ff ff 7f 00 00`

Byte Ordering: Continued

- Ntohs()
- byteorder(3)



Byte Ordering: Continued

BYTEORDER(3) Library Functions Manual BYTEORDER(3)

NAME

htonl, htons, ntohl, ntohs - convert values between host and network byte order

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);

uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

DESCRIPTION

The **htonl**() function converts the unsigned integer hostlong from host byte order to network byte order.

The **htons**() function converts the unsigned short integer hostshort from host byte order to network byte order.

The **ntohl**() function converts the unsigned integer netlong from network byte order to host byte order.

The **ntohs**() function converts the unsigned short integer netshort from network byte order to host byte order.

On the i386 the host byte order is Least Significant Byte first, whereas the network byte order, as used on the Internet, is Most Significant Byte first.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
htonl(), **htons**(), **ntohl**(), **ntohs**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2008.

HISTORY

POSIX.1-2001.

SEE ALSO

bswap(3), **endian**(3), **gethostbyname**(3), **getservent**(3)

Bits and Boolean Algebra

- NOT() operator
- Simple operation
- Inverts data
- Bar = ~foo;

~	
0	1
1	0

Boolean Algebra Continued

- Logical AND (&)
- Used to check flags
- (foo & 1)
- (foo & 2)
- (foo & 3)

&	0	1
0	0	0
1	0	1

Boolean Algebra Continued

- Logical OR (|)
- Used to check/set flags
- (foo | 1)
- (foo | 2)
- (foo | 3)

|	0	1
0	0	1
1	1	1

Boolean Algebra Continued

- Logical XOR (^)
- Used to clear values
- (foo ^ 1)
- (foo ^ 2)
- (foo ^ 3)

^	0	1
0	0	1
1	1	0

Boolean Algebra Continued

- Combine NOT w/XOR
- (foo ^ (!1))
- (foo ^ (!2))
- (foo ^ (!3))
- GNU Documentation