What you should know by today

CPU basic architecture

- RISC, Units, Local Memory, Registers, Instructions

Registers

- Number, size, kind

Instructions

- Size, kind

Data representation

- Hexadecimal, little-big endian, one's and 2's representation and computation rules.

Useful web address to find support on assembly, simulator

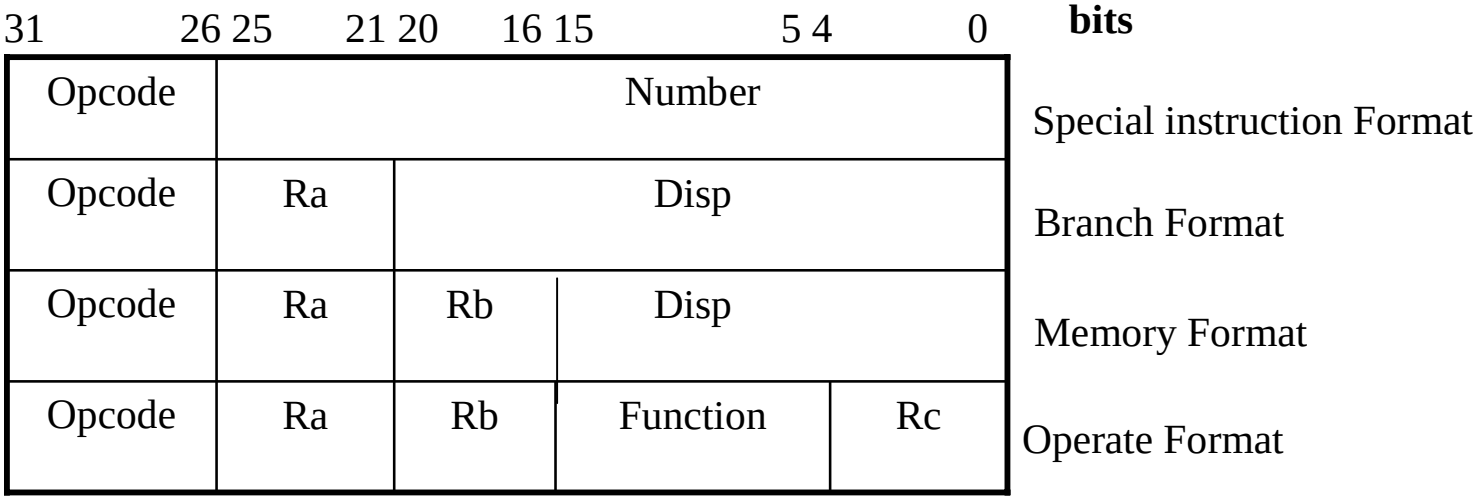
Tutorials:

- Architecture, registers, instructions format
- Using the simulator

Demonstrator:

- Familiar with the alpha simulator

Instruction Format



All the instructions have a 6 bit opcode stored in bits 26-31 which provides the instruction type. From there the CPU knows how to decode the instruction.

Branch instructions: *bne $1, loop*

- Test the value of register Ra
  - Do nothing
  - Do something: modify the PC (signed 21-bit PC-relative longword target displacement)

Integer instructions: *addq $0, $1, $2*

- Arithmetic and logical operations on registers
- Use Ra and Rb or a 8-bit litteral as source operand and write the result in Rc

Memory instructions:

- Move bytes,…, quadwords between Ra and memory, using Rb plus a signed 16-bit displacement as the memory adress
- Load data from memory into a register
- Store data from register into memory

ldq $2, 0($1) *stl $3, 12($2)*

Registers

Alpha has 32 integer registers and the same number of floating point registers. We will only focus on the integer registers.
The registers are referred to by "$" sign followed by a number between 0 and 31 (inclusive), e.g. $0, $1, etc.
Each register is given a software name, which is normally used in programs. When referring to a register, a "$" sign should precede the software name of the register, e.g. `$T0, $S0`, etc.

Register number	software name	usage
0	V0	Used for expression evaluations and to hold the integer function results.
1-8	T0-T7	Temporary registers used for expression evaluations
9-14	S0-S5	Registers for holding values of variables.
15	FP	Contains the frame pointer (if needed).
16-21	A0-A5	Used to pass the first six-integer type actual arguments.
22-25	T8-T11	Temporary registers used for expression evaluations.
26	RA	Contains the return address.
27	PV	Contains the function value and used for expression evaluation.
28	AT	Reserved for the assembler.
29	GP	Contains the global pointer.
30	SP	Contains the stack pointer.
31	ZERO	Always has the value 0.

Memory

Memory is accessed via 64-bit addressed (quadword), using either the little-endian, or optionally the big-endian byte numbering convention.

Why are longword addresses not enough: 2^{32} = 4*1024*1024*1024
Memory addressing would be limited to 4Gbytes of memory

The memory of Alpha is byte addressable (The basic addressable unit in the Alpha architecture is the 8-bit byte).

Data in memory needs to be aligned according to its size (e.g data size).

An aligned datum of size 2^n is stored in memory at a byte address that is a multiple of 2^n (One that has n-low order zeros). That is, the last n bits of the address of the byte are zeros.

- Significant performance penalty when accessing longword operands that are not naturally aligned.
- Instructions are represented by longwords: they need to be stored at addresses multiple of 4 (bytes) which requires the memory addresses to be longword aligned.

A word 0x1234 stored in memory must be at a word-aligned address
Word 0x1234 cannot be at address 0x1, since word is a 2 bytes long entity (0x1 is not multiple of 2 !!)

- Can store a word at addresses such as 0x……4, 0x.....8
- Did I forget something? ->

Word:

- (least significant bit of address at 0: multiple of 2)

Longword:

- (2 least significant bits of address at 0: multiple of 4)

Quadword:

- (3 least significant bits of address at 0: multiple of 8)

All machines: multi-byte object is stored as a continuous sequence of byte, with the address of the object given by the smallest address of the bytes used.

Let's suppose we have a longword (of hexadecimal value 0x01234567) stored at a starting address 0x1000 (it is a quadword).
The memory should look like:

Big endian

	0x1000	0x1001	0x1002	0x1003	
	01	23	45	67	

Little endian

	0x1000	0x1001	0x1002	0x1003	
	67	45	23	01	

Signed integer:
Most significant bit carries the sign :

- positive: 0
  hexadecimal: from 0x00.. to 0x7ff….
- negative: 1
  hexadecimal: from 0x800.. to Oxff…

16-bit representation: 32-bit representation:
-12345 (digital): 0xcfc7 0xffffcfc7

Little and Big Endian

On modern machines, Memory is byte addressable
Individual bytes can be referred to by an address.
Values (word, longword, quadword) which need more than one byte can be referred to by the address of their low byte.

When several bytes are necessary to represent (store) a number:
Little endian (least significant byte first)
Big endian (Most significant byte first)

Writing numbers:

Everyday you write decimal values in big endian (most significant number first) in the base 10:
For example, twelve hundreds and thirty four is represented as:

1234 = 1*1000 + 2*100 + 3*10 + 4*1

You could decide to write decimal values in little endian format (least significant part of the value first):
For example, twelve hundreds and thirty four could be represented as:
“1234” = 1234 = 4*1 + 3*10 + 2*100 + 1*1000

Values of the numbers are the same, it is just the way they are represented (stored) which defer.

For big endian data:
The value of a longword stored at address base is computed as
byte[base] << 24 + byte[base + 1] << 16
+ byte[base + 2] << 8 + byte[base + 3]

For little endian data:

The value of a longword stored at address base is computed as
byte[base] + byte[base + 1] << 8 +
+ byte[base + 2] << 16 + byte[base + 3] << 24

- Expand the previous formulae for quadwords.

Examples

Store the quadword 0x0123456789abcdef in memory at starting memory address 0x1000000 (*it is a quadword address*)

	Big endian	**Little endian**
0x1000000		
0x1000001		
0x1000002		
0x1000003		
0x1000004		
0x1000005		
0x1000006		
0x1000007		

- No consistency among different computer architectures regarding storage of numbers in big endian or little endian format.
- Choose the format on the Alpha on startup, but the choice tends to be little endian
- Alpha simulator only supports this alternative.

Strings are stored as a sequence of bytes, with one character per byte.

- The end of the string is indicated by a null (zero) byte.
- Strings are always stored in big endian format
  - The first character (stored first) at the low address end.

68	**65**	**6c**	**6c**	**6f**	**21**	**00**	
‘h’	**‘e’**	**‘l’**	**‘l’**	**‘o’**	**‘!’**	**‘\0’**	

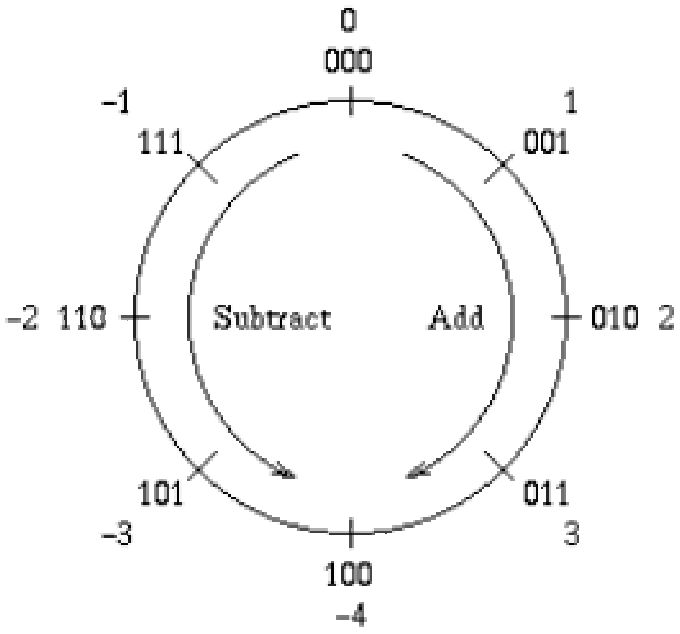
Textual Characters:

The characters with ASCII value 0x21 to 0x7e represent textual characters.

The full ASCII character set is as follows

00	NUL	01	SOH	02	STX	03	ETX
04	EOT	05	ENQ	06	ACK	07	BEL (\a)
08	BS (\b)	09	HT (\t)	0A	LF (\n)	0B	VT (\v)
0C	FF (\f)	0D	CR (\r)	0E	SO	0F	SI
10	DLE	11	DC1	12	DC2	13	DC3
14	DC4	15	NAK	16	SYN	17	ETB
18	CAN	19	EM	1A	SUB	1B	ESC
1C	FS	1D	GS	1E	RS	1F	US
20	SP	21	!	22	"	23	#
24	$	25	%	26	&	27	'
28	(	29	)	2A	*	2B	+
2C	,	2D	-	2E	.	2F	/
30	0	31	1	32	2	33	3
34	4	35	5	36	6	37	7
38	8	39	9	3A	:	3B	;
3C	<	3D	=	3E	>	3F	?
40	@	41	A	42	B	43	C
44	D	45	E	46	F	47	G
48	H	49	I	4A	J	4B	K
4C	L	4D	M	4E	N	4F	O
50	P	51	Q	52	R	53	S
54	T	55	U	56	V	57	W
58	X	59	Y	5A	Z	5B	[
5C	\	5D	]	5E	^	5F	_
60	`	61	a	62	b	63	c
64	d	65	e	66	f	67	g
68	h	69	i	6A	j	6B	k
6C	l	6D	m	6E	n	6F	o
70	p	71	q	72	r	73	s
74	t	75	u	76	v	77	w
78	x	79	y	7A	z	7B	{
7C	|	7D	}	7E	~	7F	DEL

The 2's complement representation



To add, subtract, or multiply two complement numbers, we just perform the operation as if the numbers are unsigned, and throw away any additional bits generated. To perform negation, form the ones complement (subtract from all ones (the representation of -1), then add 1.

Suppose we have n bits. We can represent numbers between -2^n-1 and $+(2^n$-1 - 1). For example if n is 8, we represent numbers between -128 and +127.

- A positive number x is represented as itself, x.
- 0 is represented as itself, 0.
- A negative number -x is represented as:
  ((1 << n) - 1) - x + 1 (i.e. $(2^n$ -1) - x + 1).

Now (1 << n) - 1 or 2^n - 1 is the bit pattern 1111...11, and it is easy to subtract a number from this value in binary, because it is just a matter of interchanging 0's and 1's.

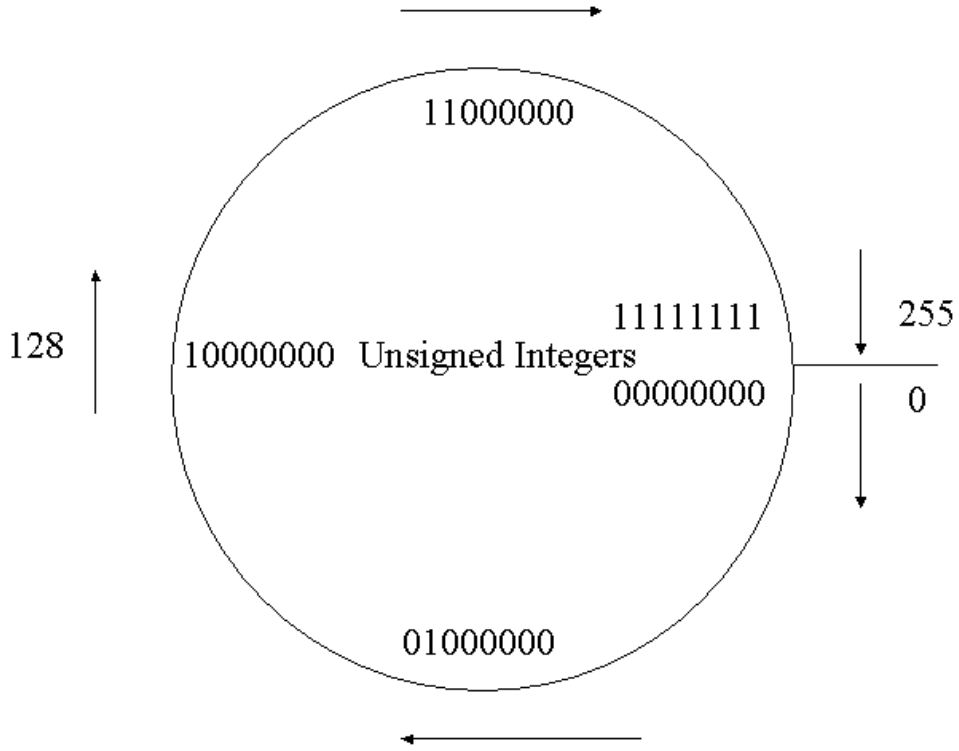
For example -30 is represented as 11111111 - 00011110 + 1 = 11100001 + 1 = 11100010.

Exercise

What is the bit pattern for decimal 42 and -42 as 8 bit two's complement numbers?

Unsigned integers representation

Given 8 bits, we can interpret the bit patterns 00000000, 00000001, 00000010, 00000011, ...11111110, 11111111 as unsigned integers 0, 1, 2, 3, ... 254, 255 (decimal). With this interpretation, we cannot represent integers outside the range 0 ... 255. We can have a similar representation if we have more bits.



Suppose we have n bits. We can represent numbers between 0 and 2^n - 1. For example if n is 8, we represent numbers between 0 and 255.
For example, decimal 30 is 16 + 8 + 4 + 2, and so is represented as the bit pattern 00011110.

Data representation examples

Find the hexadecimal representation for 12345:

12345 = 8192+4096+32+16+8+1

Decimal 12345	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
0x3039	0	0	1	1	0	0	0	0	0	0	1	1	1	0	0	1

Compute 2's representation for –12345:

0x1111111111111111 – 0x0011000000111001 + 1

```
  1111111111111111
-
  0011000000111001
---------------------------------
  1100111111000110
+
                             1
---------------------------------
1100|1111|1100|0111
```

0xcfc7

Extending an hexadecimal number to 64 bits

Represent the signed hexadecimal word 0x9876 as 3 bytes hexadecimal number

0x9876: 1001 1000 0111 0110: -2^15 + 2^12+……
0xff9876: 1111 1111 1001 1000 0111 0110:

-2^23+2^22+2^21+2^20+2^19+…+2^15+2^12+…. =
-2^22 +2^21+…… =
-2^21 +2^20+…….. =

-2^16+2^15+2^12+……. =
-2^15+2^12+……. =
0x9876

Extends this work to obtain the hexadecimal representation of the word 0x9876 as an hexadecimal longword