CS 61C
Summer 2019

RISC-V Instructions

Discussion 4: July 8, 2019

1 RISC-V Instruction Formats

Instructions in RISC-V can be turned into binary numbers that the machine actually reads. There are different formats to the instructions, based on what information is needed. Each of the fields above is filled in with binary that represents the information. Each of the registers takes a 5 bit number that is the numeric name of the register (i.e. zero = 0, ra = 1, s1 = 9). See your reference card to know which register corresponds to which number.

31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

2 Addressing in RISC-V

2.1 What is range of 32-bit instructions that can be reached from the current PC using a branch instruction?

The immediate field of the branch instruction is 12 bits. This field only references addresses that are divisible by 2, so the immediate is multiplied by 2 before being added to the PC. Thus, the branch immediate can move the reference 2-byte instructions that are within $[-2^{11}, 2^{11} - 1]$ instructions of the current PC. The instructions we use, however, are 4 bytes so they reside at addresses that are divisible by 4 not 2. Therefore, we can only reference half as many 4-byte instructions as before, and the range of 4-byte instructions is $[-2^{10}, 2^{10} - 1]$

2.2 What is the range of 32-bit instructions that can be reached from the current PC using a jump instruction?

The immediate field of the jump instruction is 20 bits. Similar to above, this immediate is multiplied by 2 before added to the PC to get the final address. Since the immediate is signed, the range of 2-byte instructions that can be referenced is $[-2^{19}, 2^{19} - 1]$. As we actually want the number of 4-byte instructions, we actually can reference those within $[-2^{18}, 2^{18} - 1]$ instructions of the current PC.

2.3 Given the following RISC-V code (and instruction addresses), fill in the blank fields for the following instructions (you'll need your RISC-V green card!).

```
0x002cff00: loop: add t1, t2, t0         |________|________|________|________|________|__0x33__|
0x002cff04:       jal ra, foo            |__________________________|________________|__0x6F__|
0x002cff08:       bne t1, zero, loop     |________|________|________|________|________|__0x63__|
...
0x002cff2c: foo:  jr ra                 ra=___________________
```

```
0x002cff00: loop: add t1, t2, t0         |    0    |     5    |     7     |  0  |  6  |  0x33  |
0x002cff04:       jal ra, foo            |  0  |   0x14   |  0  |     0     |     1     |  0x6F  |
0x002cff08:       bne t1, zero, loop     |  1  |  0x3F  |  0  |  6  |  1  |  0xC  |  1  |  0x63  |
...
0x002cff2c: foo:  jr ra                 ra=__0x002cff08___
```

3 RISC-V with Arrays and Lists

Comment what each code block does. Each block runs in isolation. Assume that there is an array, `int arr[6] = {3, 1, 4, 1, 5, 9}`, which starts at memory address `0xBFFFFF00`, and a linked list struct (as defined below), `struct ll* lst`, whose first element is located at address `0xABCD0000`. Let `s0` contain `arr`'s address `0xBFFFFF00`, and let `s1` contain `lst`'s address `0xABCD0000`. You may assume integers and pointers are 4 bytes and that structs are tightly packed. Assume that `lst`'s last node's `next` is a NULL pointer to memory address `0x00000000`.

```
struct ll {
    int val;
    struct ll* next;
}
```

3.1
```
lw  t0, 0(s0)
lw  t1, 8(s0)
add t2, t0, t1
sw  t2, 4(s0)
```

Sets `arr[1]` to `arr[0]` + `arr[2]`

3.2
```
loop: beq  s1, x0, end
      lw   t0, 0(s1)
      addi t0, t0, 1
      sw   t0, 0(s1)
      lw   s1, 4(s1)
      jal  x0, loop
 end:
```

Increments all values in the linked list by 1.

3.3
```
       add  t0, x0, x0
loop:  slti t1, t0, 6
       beq  t1, x0, end
```

```
        slli t2, t0, 2
        add  t3, s0, t2
        lw   t4, 0(t3)
        sub  t4, x0, t4
        sw   t4, 0(t3)
        addi t0, t0, 1
        jal  x0, loop
 end:
```

Negates all elements in `arr`

4 RISC-V Calling Conventions

4.1 How do we pass arguments into functions?

Use the 8 arguments registers `a0 - a7`

4.2 How are values returned by functions?

Use `a0` and `a1` as the return value registers as well

4.3 What is `sp` and how should it be used in the context of RISC-V functions?

`sp` stands for stack pointer. We subtract from `sp` to create more space and add to free space. The stack is mainly used to save (and later restore) the value of registers that may be overwritten.

4.4 Which values need to saved by the caller, before jumping to a function using `jal`?

Registers `a0 - a7`, `t0 - t6`, and `ra`

4.5 Which values need to be restored by the callee, before using `jalr` to return from a function?

Registers `sp`, `gp` (global pointer), `tp` (thread pointer), and `s0 - s11`. Important to note that we don't really touch `gp` and `tp`

5 Writing RISC-V Functions

5.1 Write a function `sumSquare` in RISC-V that, when given an integer `n`, returns the summation below. If `n` is not positive, then the function returns 0.

$$n^2 + (n-1)^2 + (n-2)^2 + \ldots + 1^2$$

For this problem, you are given a RISC-V function called `square` that takes in a single integer and returns its square. Implement `sumSquare` using `square` as a subroutine. Be sure to follow RISC-V caller/callee convention. (Hints: for `sumSquare`, in what register can we expect the parameter `n`? What registers should hold `square`'s parameter and return value? In what register should we place the return value of `sumSquare`? What needs to go in `sumSquare`'s prologue and epilogue?)

```
sumSquare: addi sp, sp -12   # Make space for 3 words on the stack
           sw   ra, 0(sp)    # Store the return address
           sw   s0, 4(sp)    # Store register s0
           sw   s1, 8(sp)    # Store register s1
           add  s0, a0, x0   # Set s0 equal to the parameter n
           add  s1, x0, x0   # Set s1 (accumulator) equal to 0
     loop: bge  x0, s0, end  # Branch if s0 is not positive
           add  a0, s0, x0   # Set a0 to the value in s0, setting up
                             # args for call to function square
           jal  ra, square   # Call the function square
           add  s1, s1, a0   # Add the returned value into s1
           addi s0, s0, -1   # Decrement s0 by 1
           jal  x0, loop     # Jump back to the loop label
     end:  add  a0, s1, x0   # Set a0 to s1 (desired return value)
           lw   ra, 0(sp)    # Restore ra
           lw   s0, 4(sp)    # Restore s0
           lw   s1, 8(sp)    # Restore s1
           addi sp, sp, 12   # Free space on the stack for the 3 words
           jr   ra           # Return to the caller
```

6 More Translating between C and RISC-V

6.1 Translate between the RISC-V code to C. You may want to use the RISC-V Green Card on the next page as a reference. What is this RISC-V function computing? Assume no stack or memory-related issues, and assume no negative inputs.

C	RISC-V
`// a0 -> x, a1 -> y,`	`Func: addi t0 x0 1`
`// t0 -> result`	`Loop: and t1 a1 a1`
`// Function computes pow(x,y)`	`      beq t1 x0 Done`
`// Direct translation:`	`      mul t0 t0 a0`
`int power(int x, int y) {`	`      addi a1 a1 -1`
`  int result = 1;`	`      jal x0 Loop`
`  while (y & y != 0) {`	`Done: add a0 t0 x0`
`    result *= x;`	`      jr ra`
`    y--;`	
`  }`	
`  return result;`	
`}`	
`// Note the loop condition could`	
`// be simplified.`	

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
`add,addw`	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
`addi,addiw`	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
`and`	R	AND	R[rd] = R[rs1] & R[rs2]	
`andi`	I	AND Immediate	R[rd] = R[rs1] & imm	
`auipc`	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	

`beq`	SB	Branch EQual	if(R[rs1]==R[rs2)
			PC=PC+{imm,1b'0}	

`bge`	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2)
			PC=PC+{imm,1b'0}	

`bgeu`	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2)
			PC=PC+{imm,1b'0}
				2)

`blt`	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	
`bltu`	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	2)
`bne`	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}	
`ebreak`	I	Environment BREAK	Transfer control to debugger	
`ecall`	I	Environment CALL	Transfer control to operating system	
`jal`	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
`jalr`	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
`lb`	I	Load Byte	R[rd] = {56'bM[](7),M[R[rs1]+imm](7:0)}	4)
`lbu`	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm](7:0)}	
`ld`	I	Load Doubleword	R[rd] = M[R[rs1]+imm](63:0)	
`lh`	I	Load Halfword	R[rd] = {48'bM[](15),M[R[rs1]+imm](15:0)}	4)
`lhu`	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm](15:0)}	
`lui`	U	Load Upper Immediate	R[rd] = {32b'imm<31>, imm, 12'b0}	
`lw`	I	Load Word	R[rd] = {32'bM[](31),M[R[rs1]+imm](31:0)}	4)
`lwu`	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}	
`or`	R	OR	R[rd] = R[rs1] | R[rs2]	
`ori`	I	OR Immediate	R[rd] = R[rs1] | imm	
`sb`	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
`sd`	S	Store Doubleword	M[R[rs1]+imm](63:0) = R[rs2](63:0)	
`sh`	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
`sll,sllw`	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
`slli,slliw`	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
`slt`	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
`slti`	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
`sltiu`	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
`sltu`	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
`sra,sraw`	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
`srai,sraiw`	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
`srl,srlw`	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
`srli,srliw`	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
`sub,subw`	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
`sw`	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
`xor`	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
`xori`	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
`lb`	I	0000011	000		03/0
`lh`	I	0000011	001		03/1
`lw`	I	0000011	010		03/2
`ld`	I	0000011	011		03/3
`lbu`	I	0000011	100		03/4
`lhu`	I	0000011	101		03/5
`lwu`	I	0000011	110		03/6
`addi`	I	0010011	000		13/0
`slli`	I	0010011	001	0000000	13/1/00
`slti`	I	0010011	010		13/2
`sltiu`	I	0010011	011		13/3
`xori`	I	0010011	100		13/4
`srli`	I	0010011	101	0000000	13/5/00
`srai`	I	0010011	101	0100000	13/5/20
`ori`	I	0010011	110		13/6
`andi`	I	0010011	111		13/7
`auipc`	U	0010111			17
`addiw`	I	0011011	000		1B/0
`slliw`	I	0011011	001	0000000	1B/1/00
`srliw`	I	0011011	101	0000000	1B/5/00
`sraiw`	I	0011011	101	0100000	1B/5/20
`sb`	S	0100011	000		23/0
`sh`	S	0100011	001		23/1
`sw`	S	0100011	010		23/2
`sd`	S	0100011	011		23/3
`add`	R	0110011	000	0000000	33/0/00
`sub`	R	0110011	000	0100000	33/0/20
`sll`	R	0110011	001	0000000	33/1/00
`slt`	R	0110011	010	0000000	33/2/00
`sltu`	R	0110011	011	0000000	33/3/00
`xor`	R	0110011	100	0000000	33/4/00
`srl`	R	0110011	101	0000000	33/5/00
`sra`	R	0110011	101	0100000	33/5/20
`or`	R	0110011	110	0000000	33/6/00
`and`	R	0110011	111	0000000	33/7/00
`lui`	U	0110111			37
`addw`	R	0111011	000	0000000	3B/0/00
`subw`	R	0111011	000	0100000	3B/0/20
`sllw`	R	0111011	001	0000000	3B/1/00
`srlw`	R	0111011	101	0000000	3B/5/00
`sraw`	R	0111011	101	0100000	3B/5/20
`beq`	SB	1100011	000		63/0
`bne`	SB	1100011	001		63/1
`blt`	SB	1100011	100		63/4
`bge`	SB	1100011	101		63/5
`bltu`	SB	1100011	110		63/6
`bgeu`	SB	1100011	111		63/7
`jalr`	I	1100111	000		67/0
`jal`	UJ	1101111			6F
`ecall`	I	1110011	000	000000000000	73/0/000
`ebreak`	I	1110011	000	000000000001	73/0/001

Notes: 1) *The Word version only operates on the rightmost 32 bits of a 64-bit registers*
2) *Operation assumes unsigned integers (instead of 2's complement)*
3) *The least significant bit of the branch address in jalr is set to 0*
4) *(signed) Load instructions extend the sign bit of data to fill the 64-bit register*
5) *Replicates the sign bit to fill in the leftmost bits of the result during right shift*
6) *Multiply with one operand signed and one unsigned*
7) *The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register*
8) *Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0,+0, +inf, denorm, ...)*
9) *Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location*

The immediate field is sign-extended in RISC-V

PSEUDO INSTRUCTIONS

MNEMONIC	NAME	DESCRIPTION	USES
`beqz`	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	`beq`
`bnez`	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	`bne`
`fabs.s,fabs.d`	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	`fsgnx`
`fmv.s,fmv.d`	FP Move	F[rd] = F[rs1]	`fsgnj`
`fneg.s,fneg.d`	FP negate	F[rd] = –F[rs1]	`fsgnjn`
`j`	Jump	PC = {imm,1b'0}	`jal`
`jr`	Jump register	PC = R[rs1]	`jalr`
`la`	Load address	R[rd] = address	`auipc`
`li`	Load imm	R[rd] = imm	`addi`
`mv`	Move	R[rd] = R[rs1]	`addi`
`neg`	Negate	R[rd] = –R[rs1]	`sub`
`nop`	No operation	R[0] = R[0]	`addi`
`not`	Not	R[rd] = ~R[rs1]	`xori`
`ret`	Return	PC = R[1]	`jalr`
`seqz`	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	`sltiu`
`snez`	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	`sltu`

ARITHMETIC CORE INSTRUCTION SET

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
`mul,mulw`	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
`mulh`	R	MULtiply High	R[rd] = (R[rs1] * R[rs2])(127:64)	
`mulhu`	R	MULtiply High Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
`mulhsu`	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
`div,divw`	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
`divu`	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
`rem,remw`	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
`remu,remuw`	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64A Atomtic Extension

`amoadd.w,amoadd.d`	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
`amoand.w,amoand.d`	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
`amomax.w,amomax.d`	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
`amomaxu.w,amomaxu.d`	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
`amomin.w,amomin.d`	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
`amominu.w,amominu.d`	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
`amoor.w,amoor.d`	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
`amoswap.w,amoswap.d`	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
`amoxor.w,amoxor.d`	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
`lr.w,lr.d`	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
`sc.w,sc.d`	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	Opcode
I	imm[11:0]		rs1	funct3	rd	Opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

REGISTER NAME, USE, CALLING CONVENTION

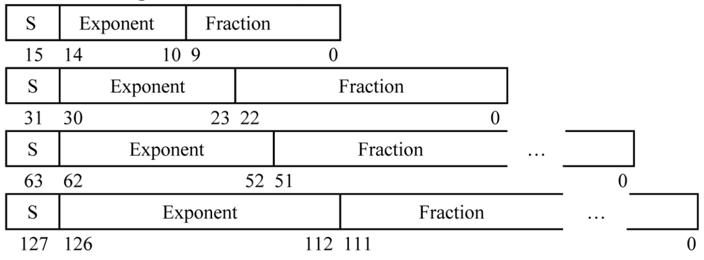
REGISTER	NAME	USE	SAVER
`x0`	`zero`	The constant value 0	N.A.
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	--
`x4`	`tp`	Thread pointer	--
`x5-x7`	`t0-t2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/Frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-x11`	`a0-a1`	Function arguments/Return values	Caller
`x12-x17`	`a2-a7`	Function arguments	Caller
`x18-x27`	`s2-s11`	Saved registers	Callee
`x28-x31`	`t3-t6`	Temporaries	Caller
`f0-f7`	`ft0-ft7`	FP Temporaries	Caller
`f8-f9`	`fs0-fs1`	FP Saved registers	Callee
`f10-f11`	`fa0-fa1`	FP Function arguments/Return values	Caller
`f12-f17`	`fa2-fa7`	FP Function arguments	Caller
`f18-f27`	`fs2-fs11`	FP Saved registers	Callee
`f28-f31`	`ft8-ft11`	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

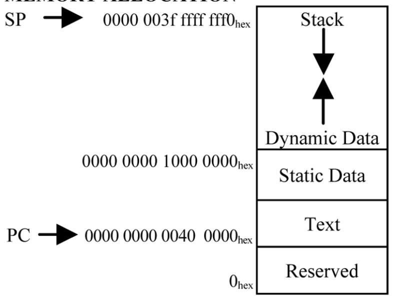
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

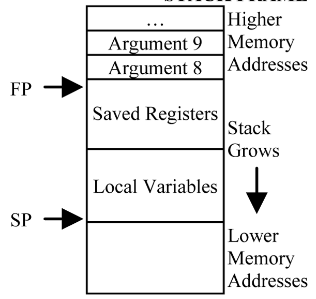
where Half-Precision Bias = 15, Single-Precision Bias = 127, Double-Precision Bias = 1023, Quad-Precision Bias = 16383

IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION **STACK FRAME**





SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi
10^{-3}	milli-	m	10^{-15}	femto-	f
10^{-6}	micro-	µ	10^{-18}	atto-	a
10^{-9}	nano-	n	10^{-21}	zepto-	z
10^{-12}	pico-	p	10^{-24}	yocto-	y