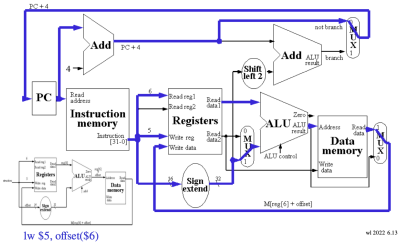
Calculating performance

- CPI: average clock cycles per instruction
- number of cycles for program P = number of instr. for P x CPI
- execution time for P = clock cycle time x number of cycles for P
  = 1 / clock speed x number of cycles for P
- average exe. time for P1, P2...Pn = 1/n (exe. time for P1 +···+ exe. time for Pn)
  (assume equal workload)

execution time equation: exe. time = instr. count x CPI x cycle time

wl 2023 2.3

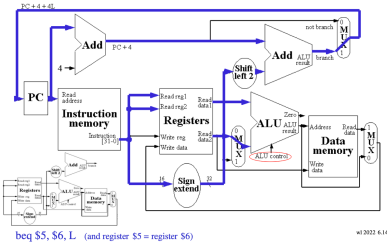
Combined datapath for load



lw $5, offset($6)

wl 2022 6.13

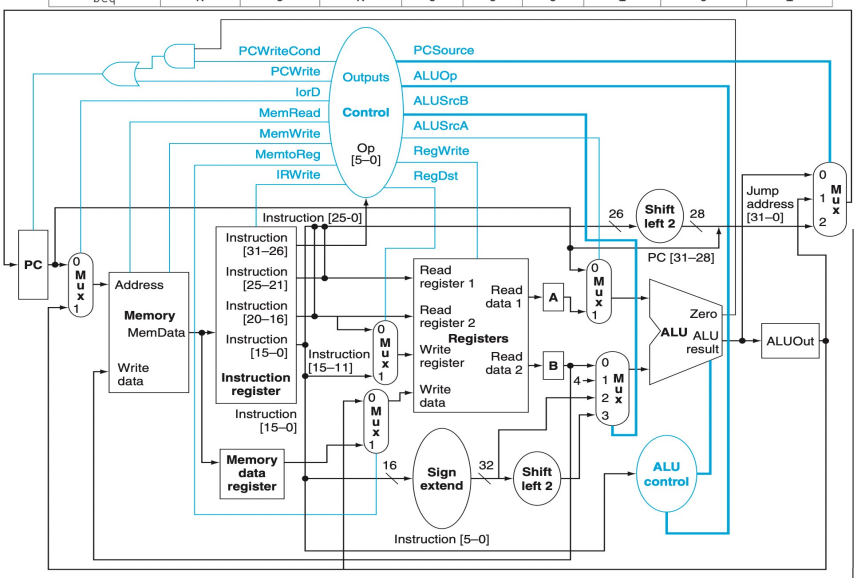
Combined datapath for branch



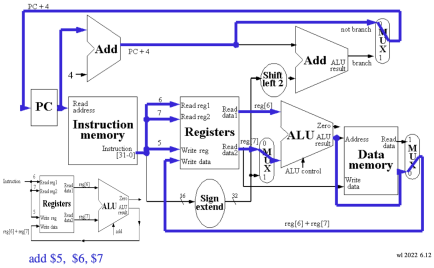
beq $5, $6, L (and register $5 = register $6)

wl 2022 6.14

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1



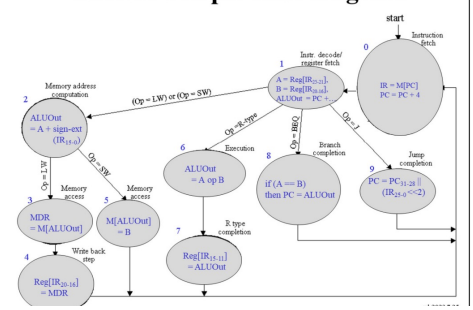
Combined datapath for R-type



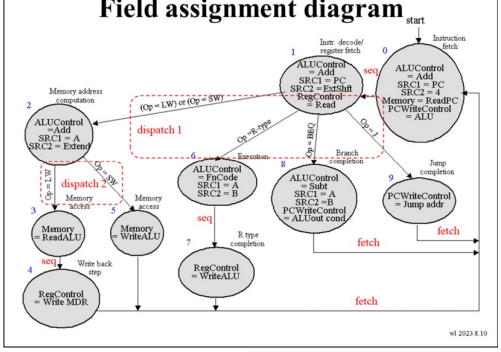
add $5, $6, $7

wl 2022 6.12

Execution steps: state diagram



Field assignment diagram



wl 2023 8.10

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]),2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Comparing the third algorithm and Booth's algorithm for positive numbers

Iteration	Multiplicand	Original algorithm Step	Original algorithm Product	Booth's algorithm Step	Booth's algorithm Product
0	0010	Initial Values	0000 0110	Initial Values	0000 0110 0
1	0010 0010	1: 0 => no operation 2: Shift right Product	0000 0110 0000 0011	1a: 00 => no operation 2: Shift right Product	0000 0110 0 0000 0011 0
2	0010 0010	1a: 1 => Prod=Prod+Mcand 2: Shift right Product	0010 0011 0001 0001	1c: 10 => Prod=Prod-Mcand 2: Shift right Product	1110 0011 0 1111 0001 1
3	0010 0010	1a: 1 => Prod=Prod+Mcand 2: Shift right Product	0011 0001 0001 1000	1d: 11 => no operation 2: Shift right Product	1111 0001 1 1111 0000 1
4	0010 0010	1: 0 => no operation 2: Shift right Product	0001 1000 0000 1100	1b: 01 => Prod=Prod+Mcand 2: Shift right Product	0001 1000 1 0000 1100 0

Division example using first algorithm

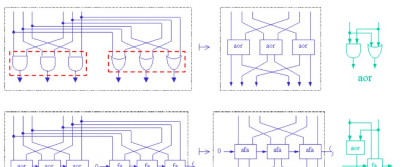
Iteration	Step	Quotient	Divisor	Remainder
0	Initial Values	0000	0010 0000	0000 0111
1	1: Rem = Rem - Div 2b: Rem<0 => +Div, sll Q, Q0=0 3: Shift Div right	0000 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111
2	1: Rem = Rem - Div 2b: Rem<0 => +Div, sll Q, Q0=0 3: Shift Div right	0000 0000 0000	0001 0000 0001 0000 0000 1000	1111 0111 0000 0111 0000 0111
3	1: Rem = Rem - Div 2b: Rem<0 => +Div, sll Q, Q0=0 3: Shift Div right	0000 0000 0000	0000 1000 0000 1000 0000 0100	1111 1111 0000 0111 0000 0011
4	1: Rem = Rem - Div 2a: Rem≥0 => sll Q, Q0=1 3: Shift Div right	0000 0001 0001	0000 0100 0000 0100 0000 0010	0000 0011 0000 0011 0000 0011
5	1: Rem = Rem - Div 2a: Rem≥0 => sll Q, Q0=1 3: Shift Div right	0001 0011 0011	0000 0010 0000 0010 0000 0001	0000 0001 0000 0001 0000 0001

dd=7 q=3
ds=2 r=1

wl 2023 5.21

Deriving ALU cell by interleaving components

- group components together to form larger repeated unit



- the dotted boxes have the same function and interface

wl 2022 4.8

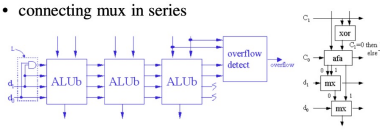
Comparison operations

- slt: set on less than, if a < b then 1 else 0
- if a < b, a-b < 0, so MSB of (a-b) is 1 (32 bits)
- implementation
  - provide additional input to each cell, left of a_i, b_i
  - LSB input from MSB ALUb output, other inputs set to 0
  - include additional mux in cell for selection
  - to select slt, $s=0$. d_1=1, d_0=1

wl 2022 4.10

Selecting ALU operation

- programmable inverter for b_i (using xor)
- connecting mux in series



- d_0d_1: 00 and, 01 or, 10 add, 11 subtract
- detecting overflow: exercise

wl 2022 4.9

Zero detection

- beq, bne: test a=b or a-b=0
- include another gate to test if output zero

- summary: $s\ d_0\ d_1$ function: 011 set on less than, 100 and, 101 or, 110 add, 111 subtract

wl 2022 4.11

Instructions

Data movement

`movq Src, Dest`	Dest = Src
`movsbq Src,Dest`	Dest (quad) = Src (byte), sign-extend
`movzbq Src,Dest`	Dest (quad) = Src (byte), zero-extend

Conditional move

`cmove Src, Dest`	Equal / zero
`cmovne Src, Dest`	Not equal / not zero
`cmovs Src, Dest`	Negative
`cmovns Src, Dest`	Nonnegative
`cmovg Src, Dest`	Greater (signed >)
`cmovge Src, Dest`	Greater or equal (signed ≥)
`cmovl Src, Dest`	Less (signed <)
`cmovle Src, Dest`	Less or equal (signed ≤)
`cmova Src, Dest`	Above (unsigned >)
`cmovae Src, Dest`	Above or equal (unsigned ≥)
`cmovb Src, Dest`	Below (unsigned <)
`cmovbe Src, Dest`	Below or equal (unsigned ≤)

Control transfer

`cmpq Src2, Src1`	Sets CCs Src1 Src2
`testq Src2, Src1`	Sets CCs Src1 & Src2
`jmp label`	jump
`je label`	jump equal
`jne label`	jump not equal
`js label`	jump negative
`jns label`	jump non-negative
`jg label`	jump greater (signed >)
`jge label`	jump greater or equal (signed ≥)
`jl label`	jump less (signed <)
`jle label`	jump less or equal (signed ≤)
`ja label`	jump above (unsigned >)
`jb label`	jump below (unsigned <)
`pushq Src`	%rsp = %rsp 8, Mem[%rsp] = Src
`popq Dest`	Dest = Mem[%rsp], %rsp = %rsp + 8
`call label`	push address of next instruction, `jmp label`
`ret`	%rip = Mem[%rsp], %rsp = %rsp + 8

Arithmetic operations

`leaq Src, Dest`	Dest = address of Src
`incq Dest`	Dest = Dest + 1
`decq Dest`	Dest = Dest − 1
`addq Src, Dest`	Dest = Dest + Src
`subq Src, Dest`	Dest = Dest − Src
`imulq Src, Dest`	Dest = Dest ∗ Src
`xorq Src, Dest`	Dest = Dest ˆ Src
`orq Src, Dest`	Dest = Dest | Src
`andq Src, Dest`	Dest = Dest & Src
`negq Dest`	Dest = − Dest
`notq Dest`	Dest = ∼ Dest
`salq k, Dest`	Dest = Dest ≪ k
`sarq k, Dest`	Dest = Dest ≫ k (arithmetic)
`shrq k, Dest`	Dest = Dest ≫ k (logical)

Addressing modes

- **Immediate**
  $Val Val
  val: constant integer value
  `movq $7, %rax`
- **Normal**
  (R) Mem[Reg[R]]
  R: register R specifies memory address
  `movq (%rcx), %rax`
- **Displacement**
  D(R) Mem[Reg[R]+D]
  R: register specifies start of memory region
  D: constant displacement D specifies offset
  `movq 8(%rdi), %rdx`
- **Indexed**
  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]
  D: constant displacement 1, 2, or 4 bytes
  Rb: base register: any of 8 integer registers
  Ri: index register: any, except %esp
  S: scale: 1, 2, 4, or 8
  `movq 0x100(%rcx,%rax,4), %rdx`

Instruction suffixes

b	byte
w	word (2 bytes)
l	long (4 bytes)
q	quad (8 bytes)

Condition codes

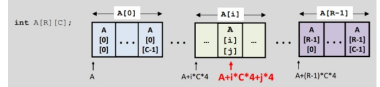
CF	Carry Flag
ZF	Zero Flag
SF	Sign Flag
OF	Overflow Flag

Integer registers

%rax	Return value
%rbx	Callee saved
%rcx	4th argument
%rdx	3rd argument
%rsi	2nd argument
%rdi	1st argument
%rbp	Callee saved
%rsp	Stack pointer
%r8	5th argument
%r9	6th argument
%r10	Scratch register
%r11	Scratch register
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

Nested Array Element Access

- Array Elements:
  - `A[i][j]` is element of type *T*, which requires *K* bytes
  - Address `= A + i * (C*K) + j*K`
    `= A + (i*C + j) * K`



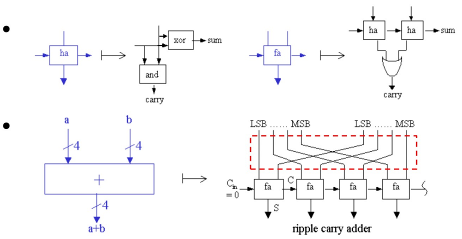
Nested Array Element Access Code

```
int get_pgh_digit(int index, int digit ){
  return pgh[index][digit];
}
```

```
leaq    (%rdi, %rdi, 4), %rax     # 5 * index
addq    %rax, %rsi                # digit + 5 * index
movl    pgh(, %rsi, 4), %eax      # *(pgh + 4*(digit + 5*index))
```

- Array Elements:
  - `pgh[index][digit]` is `int` and `sizeof(int)=4`
  - Address: `pgh + 5*4*index + 4*digit`
- Assembly Code:
  - Computes address as: `pgh + ((index+4*index) + digit)*4`
  - `movl` performs memory reference

Add / subtract



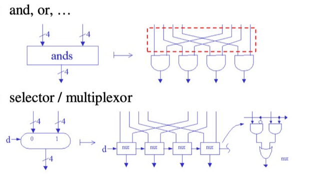
- subtractor: a-b
  - $(\sum 2^i \times b_i) + (\sum 2^i \times \bar{b}_i) = -1$ e.g. $0101_2 \oplus 1010_2 = 1111_2 = -1_{10}$
  - $(\sum 2^i \times a_i) - (\sum 2^i \times b_i) = (\sum 2^i \times a_i) + (\sum 2^i \times \bar{b}_i) + 1$

  i.e. invert b bitwise and set C_{in} = 1 for adder

wl 2023 4.7

Bit-wise logical / selection operations

- and, or, …
- selector / multiplexor



wl 2023 4.6

Choosing instructions for conditionals

compare or test		cmp b,a	test b,a
`je`	"Equal"	a == b	a&b == 0
`jne`	"Not equal"	a != b	a&b != 0
`js`	"Sign" (negative)		a&b < 0
`jns`	(non-negative)		a&b >= 0
`jg`	"Greater"	a > b	a&b > 0
`jge`	"Greater or equal"	a >= b	a&b >= 0
`jl`	"Less"	a < b	a&b < 0
`jle`	"Less or equal"	a <= b	a&b <= 0
`ja`	"Above" (unsigned >)	a > b	
`jb`	"Below" (unsigned <)	a < b	

jump or set

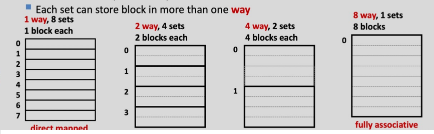
Examples:

```
        cmp 5, (%rax)
je:     (%rax) == 5
jne:    (%rax) != 5
jg:     (%rax) >  5
jl:     (%rax) <  5
```

```
        test %rdi, %rdi
je:     %rdi == 0
jne:    %rdi != 0
jg:     %rdi >  0
jl:     %rdi <  0
```

```
        test %rax, 0x1
je:     %rax_LSB == 0
jne:    %rax_LSB == 1
```

- **So we *combine* two ideas:**
  - Each address maps to exactly one set
  - Each set can store block in more than one way



x86-64 Integer Registers

63 … 31 … 15 … 7 … 0					63 … 31 … 15 … 7 … 0				
%rax	%eax	%ax	%al	Return value (Caller saved)	%r8	%r8d	%r8w	%r8b	5th Argument (Caller saved)
%rbx	%ebx	%bx	%bl	Callee saved	%r9	%r9d	%r9w	%r9b	6th Argument (Caller saved)
%rcx	%ecx	%cx	%cl	4th Argument (Caller saved)	%r10	%r10d	%r10w	%r10b	Caller saved
%rdx	%edx	%dx	%dl	3rd Argument (Caller saved)	%r11	%r11d	%r11w	%r11b	Caller saved
%rsi	%esi	%si	%sil	2nd Argument (Caller saved)	%r12	%r12d	%r12w	%r12b	Callee saved
%rdi	%edi	%di	%dil	1st Argument (Caller saved)	%r13	%r13d	%r13w	%r13b	Callee saved
%rsp	%esp	%sp	%spl	Stack pointer	%r14	%r14d	%r14w	%r14b	Callee saved
%rbp	%ebp	%bp	%bpl	Callee saved	%r15	%r15d	%r15w	%r15b	Callee saved

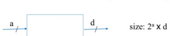
(Spring '21) Introduction to Computer Architecture

What to do on a write-miss?

- **Write-allocate** (load into cache, update line in cache)
  - Good if more writes to the location follow
  - More complex to implement
  - May evict an existing value
  - Common with write-back caches
- **No-write-allocate** (writes immediately to memory)
  - Simpler to implement
  - Slower code (bad if value consistently re-read)
  - Seen with write-through caches

C – size of cache
B – block size
N – associativity

ROM: Read Only Memory



- **Cache Set (S)**
  - A cache with M = 2^m unique addresses is organised as a group of S = 2^s *cache sets*. Each cache set consists of E *cache lines*.
- **Block Size (B)**
  - Each cache lines consists of B = 2^b *blocks* of data.
  - Given in bytes and always a power of 2 (e.g., 64 B)
  - Blocks consist of adjacent bytes, for spatial locality!
- **Offset field**
  - Low-order $\log_2(\mathbf{B})$ = **b** bits of address tell you which byte within a block
    - (address) $mod\ 2^n = n$ lowest bits of address
  - (address) modulo (# of bytes in a block)

m-bit address:	Block identifier (t bits)	Set index (s bits)	Block Offset (b bits)

Average Memory Access Time (AMAT): average time to access memory considering both hits and misses

AMAT = Hit time + Miss rate × Miss penalty

99% hit rate is twice as good as 97% hit rate!

- Assume HT of 1 clock cycle and MP of 100 clock cycles
- 97%: AMAT = 1 + (1 − 0.97) × 100 = 1 + 3 = 4 clock cycles
- 99%: AMAT = 1 + (1 − 0.99) × 100 = 1 + 1 = 2 clock cycles

- **Cache Size (C):** amount of *data* in bytes that the cache can store
  - Cache can only hold so much data (subset of next level)
  - The cache size is determine by the number of sets per the number of lines (N) in each set and the block size, thus C = S x N x B.
  - The number of blocks can be computed given the cache size and the block size (C/B)
    - Example: C = 32 KiB = 512 blocks if using 64Bytes blocks size
  - Cache organisation can be defined by the tuple (S, N, B, m)

M-bit address:	Tag (T)	Index (I)	Offset (O)
	User for tag comparison	$I = log_2(C/B/N)$	$O = log_2(B)$

- **Compulsory** (cold) miss:
  - Occurs on first access to a block
- **Conflict** miss:
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot.
    - E.g., referencing blocks 0, 8, 0, 8, … could miss every time
  - Direct-mapped caches have more conflict misses than N-way set-associative
- **Capacity** miss:
  - Occurs when the set of active cache blocks (the **working set**) is larger than the cache
  - Note: Fully-associative only has Compulsory and Capacity misses

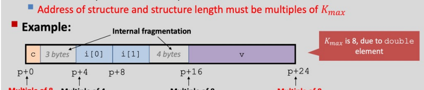
Bit Pattern	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sign & Magnitude	+0	+1	+2	+3	+4	+5	+6	+7	−0	−1	−2	−3	−4	−5	−6	−7
1s Complement	+0	+1	+2	+3	+4	+5	+6	+7	-7	-6	-5	-4	-3	-2	-1	-0
2s Complement	+0	+1	+2	+3	+4	+5	+6	+7	-8	-7	-6	-5	-4	-3	-2	-1
Excess-8	-8	-7	-6	-5	-4	-3	-2	1	0	1	2	3	4	5	6	7
BCD	0	1	2	3	4	5	6	7	8	9	-	-	-	-	-	-

What to do on a write hit?

- **Multiple copies of data exist. What is the problem with that?**
- **Write-through**
  - Write immediately to memory and all caches in between
  - Memory is always consistent with the cache copy
  - Slow: what if the same value (or line!) is written several times
- **Write-back**
  - Defer write to memory until line is evicted (replaced)
  - Need a dirty bit
    - Indicates line is different from memory
  - Higher performance (but more complex)

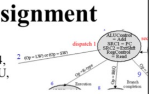
Satisfying Alignment with Structures

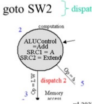
- **Within structure:**
  - Must satisfy each element's alignment requirement
- **Overall structure placement:**
  - Each structure has alignment requirement K_{max}
    - K_{max} = Largest alignment of any element
    - Counts array elements individually as elements
  - Address of structure and structure length must be multiples of K_{max}
- **Example:**



Microprogram: field assignment

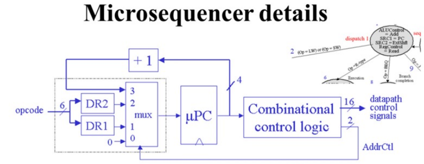
- 0 Fetch: ALUControl = Add, SRC1 = PC, SRC2 = 4, Memory = ReadPC, PCWriteControl = ALU, Sequencing = Seq
- 1 ALUControl = Add, SRC1 = PC, SRC2 = ExtShift, ReadControl = Read,
  Sequencing = if (opcode=lw) or (opcode=sw) then goto Mem1 (2)
  else if (opcode=R-type) then goto Rformat1 (6)
  else if (opcode=BEQ) then goto BEQ1 (8)
  else if (opcode=J) then goto JUMP1 (9)
  } dispatch 1
- 2 Mem1: ALUControl = Add, SRC1 = A, SRC2 = Extend, Sequencing = if (opcode=lw) then goto LW2 else goto SW2 } dispatch 2
- 3 LW2: Memory = ReadALU, Sequencing = Seq
- 4 RegControl = WriteMDR, Sequencing = Fetch
- 5 SW2: Memory = WriteALU, Sequencing = Fetch
- 6 Rformat1: ...





wl 2023 8.11

Microsequencer details



- Dispatch ROM 1 (DR1):

	opcode in	state out	
R-type	000000	0110	state 6
jump	000010	1001	state 9
beq	000100	1000	state 8
lw	100011	0010	state 2
sw	101011	0010	state 2

if R-type then state 6
elseif jump then state 9

wl 2023 8.15

C declaration	Intel data type	Assembly code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long int`	Quad word	`q`	8
`char*`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8
`long double`	Extended precision	`t`	10/12