CompSys Notes

January 22, 2019

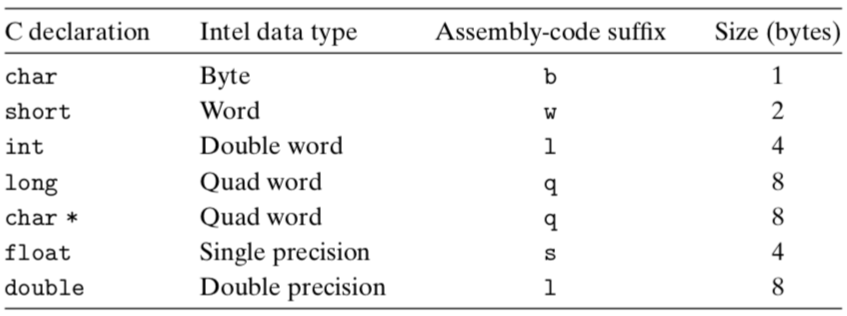
Contents

Symbols to copy paste:
{ } [ ] \ $ ∞

Machine Architecture

Assembly

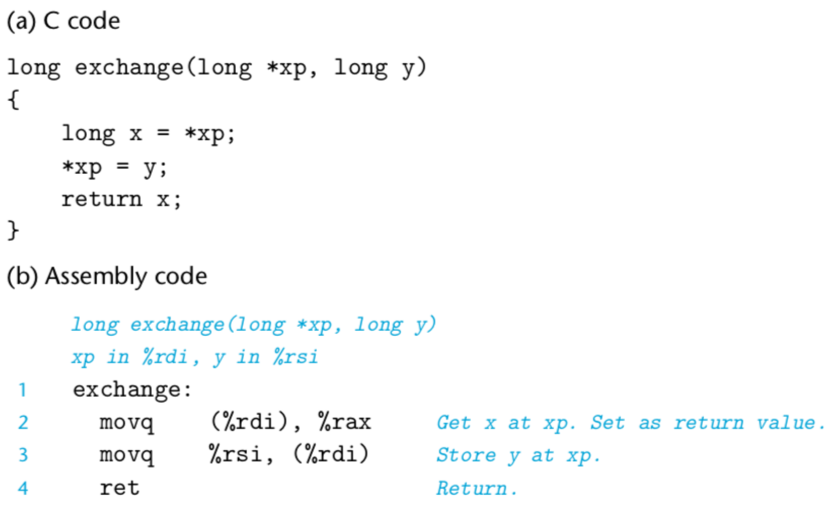
Data Types



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

Figure 3.1 **Sizes of C data types in x86-64.** With a 64-bit machine, pointers are 8 bytes long.

Pointer example



(a) C code

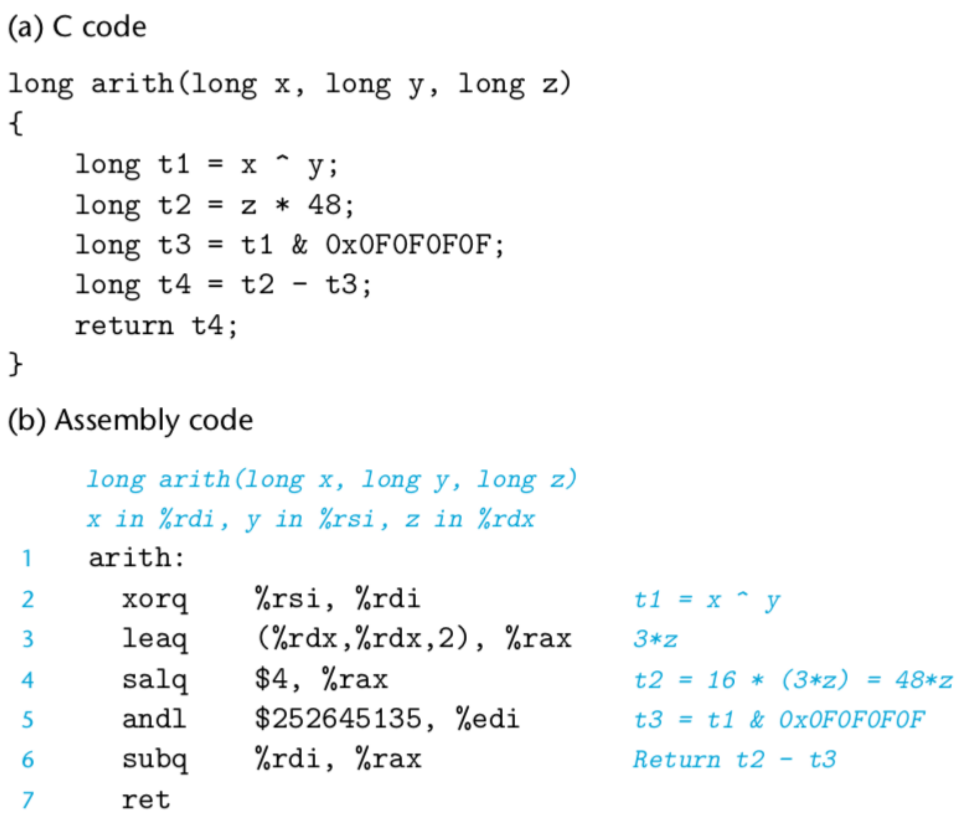
```c
long exchange(long *xp, long y)
{
    long x = *xp;
    *xp = y;
    return x;
}
```

(b) Assembly code

```
    long exchange(long *xp, long y)
    xp in %rdi, y in %rsi
exchange:
  movq    (%rdi), %rax      Get x at xp. Set as return value.
  movq    %rsi, (%rdi)      Store y at xp.
  ret                       Return.
```

Figure 3.7 **C and assembly code for exchange routine.** Registers `%rdi` and `%rsi` hold parameters `xp` and `y`, respectively.

Arithmetic example



(a) C code

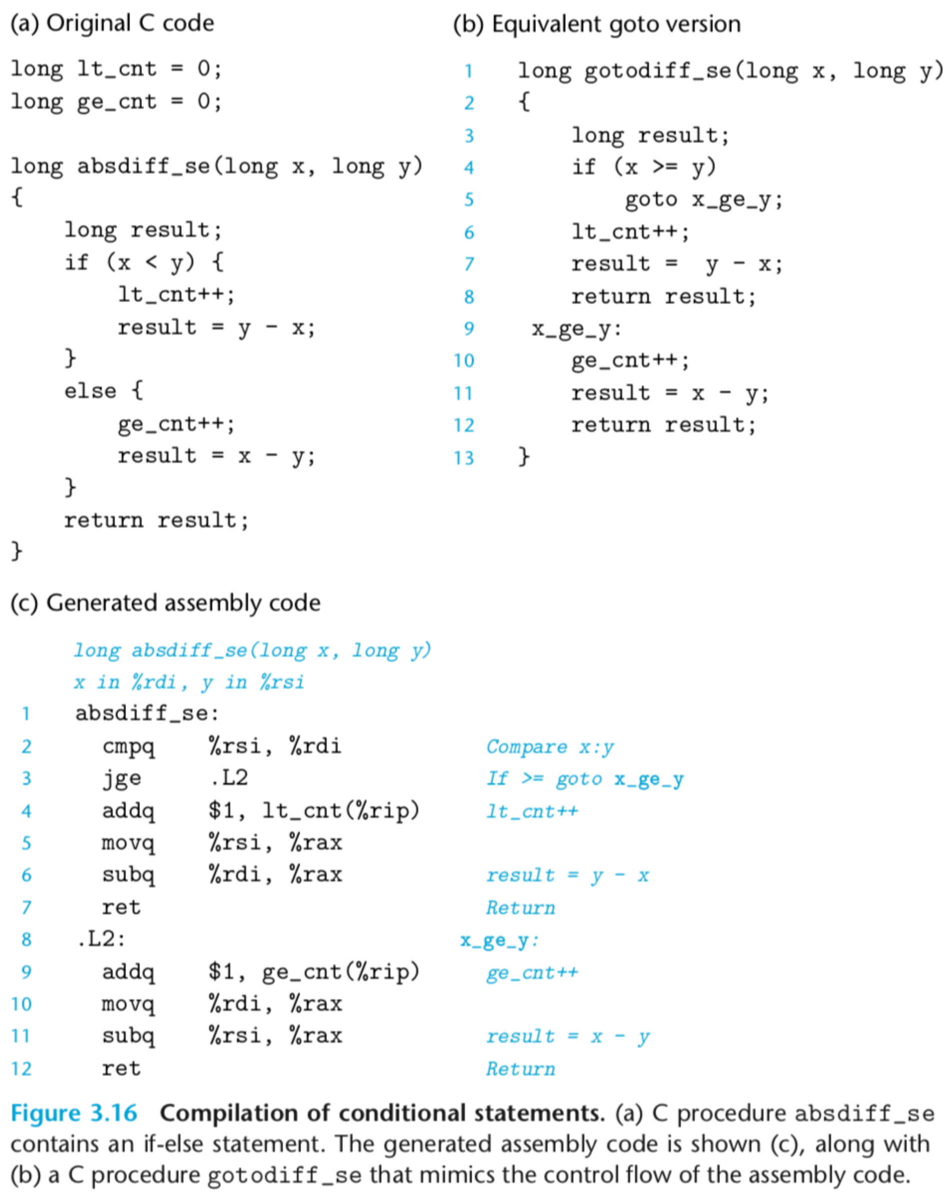
```c
long arith(long x, long y, long z)
{
    long t1 = x ^ y;
    long t2 = z * 48;
    long t3 = t1 & 0x0F0F0F0F;
    long t4 = t2 - t3;
    return t4;
}
```

(b) Assembly code

```
  long arith(long x, long y, long z)
  x in %rdi , y in %rsi , z in %rdx
arith:
  xorq    %rsi, %rdi              t1 = x ^ y
  leaq    (%rdx,%rdx,2), %rax     3*z
  salq    $4, %rax                t2 = 16 * (3*z) = 48*z
  andl    $252645135, %edi        t3 = t1 & 0x0F0F0F0F
  subq    %rdi, %rax              Return t2 - t3
  ret
```

Figure 3.11 C and assembly code for arithmetic function.

if-else example



(a) Original C code

```
long lt_cnt = 0;
long ge_cnt = 0;

long absdiff_se(long x, long y)
{
    long result;
    if (x < y) {
        lt_cnt++;
        result = y - x;
    }
    else {
        ge_cnt++;
        result = x - y;
    }
    return result;
}
```

(b) Equivalent goto version

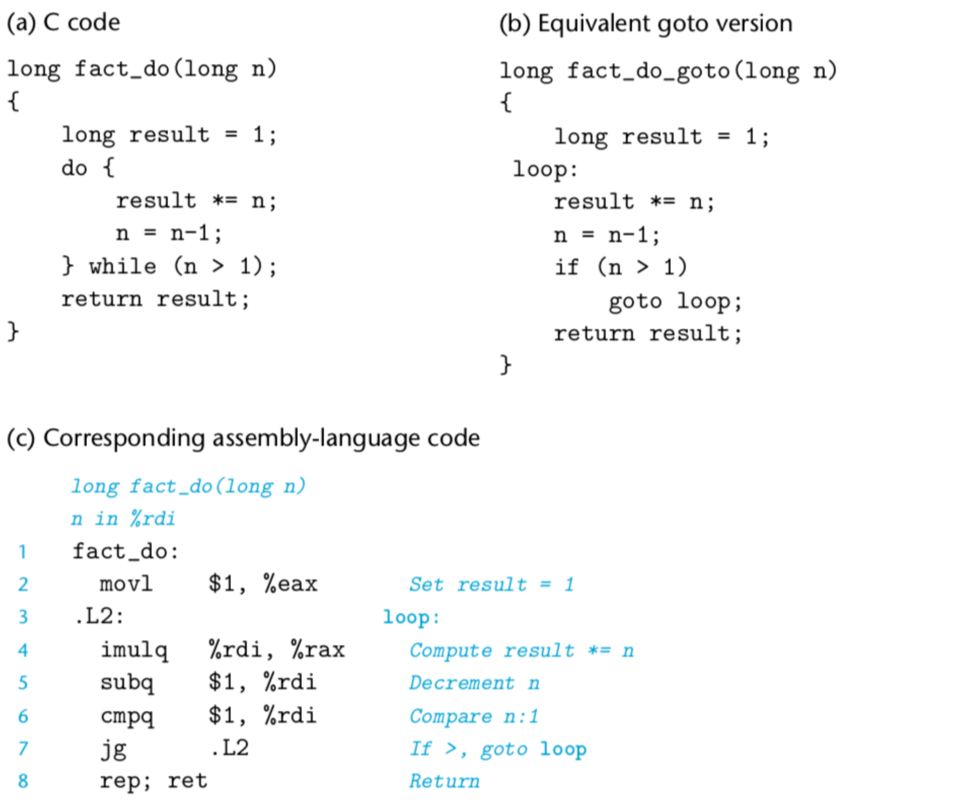
```
long gotodiff_se(long x, long y)
{
    long result;
    if (x >= y)
        goto x_ge_y;
    lt_cnt++;
    result =  y - x;
    return result;
 x_ge_y:
    ge_cnt++;
    result = x - y;
    return result;
}
```

(c) Generated assembly code

```
  long absdiff_se(long x, long y)
  x in %rdi, y in %rsi
absdiff_se:
  cmpq    %rsi, %rdi          Compare x:y
  jge     .L2                 If >= goto x_ge_y
  addq    $1, lt_cnt(%rip)    lt_cnt++
  movq    %rsi, %rax
  subq    %rdi, %rax          result = y - x
  ret                         Return
.L2:                        x_ge_y:
  addq    $1, ge_cnt(%rip)    ge_cnt++
  movq    %rdi, %rax
  subq    %rsi, %rax          result = x - y
  ret                         Return
```

Figure 3.16 **Compilation of conditional statements.** (a) C procedure absdiff_se contains an if-else statement. The generated assembly code is shown (c), along with (b) a C procedure gotodiff_se that mimics the control flow of the assembly code.

do-while example



(a) C code

```
long fact_do(long n)
{
    long result = 1;
    do {
        result *= n;
        n = n-1;
    } while (n > 1);
    return result;
}
```

(b) Equivalent goto version

```
long fact_do_goto(long n)
{
    long result = 1;
 loop:
    result *= n;
    n = n-1;
    if (n > 1)
        goto loop;
    return result;
}
```

(c) Corresponding assembly-language code

```
     long fact_do(long n)
     n in %rdi
     fact_do:
       movl    $1, %eax        Set result = 1
     .L2:                    loop:
       imulq   %rdi, %rax      Compute result *= n
       subq    $1, %rdi        Decrement n
       cmpq    $1, %rdi        Compare n:1
       jg      .L2             If >, goto loop
       rep; ret                Return
```

Figure 3.19 Code for `do-while` **version of factorial program.** A conditional jump causes the program to loop.

while example

(a) C code

```
long fact_while(long n)
{
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

(b) Equivalent goto version

```
long fact_while_jm_goto(long n)
{
    long result = 1;
    goto test;
 loop:
    result *= n;
    n = n-1;
 test:
    if (n > 1)
        goto loop;
    return result;
}
```

(c) Corresponding assembly-language code

```
  long fact_while(long n)
  n in %rdi
fact_while:
  movl    $1, %eax        Set result = 1
  jmp     .L5             Goto test
.L6:                    loop:
  imulq   %rdi, %rax      Compute result *= n
  subq    $1, %rdi        Decrement n
.L5:                    test:
  cmpq    $1, %rdi        Compare n:1
  jg      .L6             If >, goto loop
  rep; ret                Return
```

Figure 3.20 C and assembly code for `while` version of factorial using jump-to-middle translation. The C function `fact_while_jm_goto` illustrates the operation of the assembly-code version.

Operand forms

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b,r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b,r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(,r_i,s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(,r_i,s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b,r_i,s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b,r_i,s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Figure 3.3 Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

MOVQ examples

	Source	Dest	Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Pipelining

Nedenfor ses afviklingen af en stump x86prime kode på en simple pipeline:

Den viste kode er den indre løkke i en funktion som multiplicerer alle elementer i en nul-termineret tabel med et argument.

Ved indgang til løkken indeholder %r10 pointeren til tabellen og %r12 det tal alle elementer skal multipliceres med.

```
Loop:
    movq (%r10),%r11    FDXMYW
    cbe $0,%r11,Done     FDDXMYW
    multq %r12,%r11       FFDXMYW
    movq %r11,(%r10)        FDDDXMYW
    addq $8,%r10                 FFFDXMYW
    jmp Loop                        FDXMYW
Done:
```

Data hazard: Læsning/skrivning til registre efter skrivning

Control hazard: Ved branching og andre ting der ændrer PC.

Bogstaverne til højre viser hver instruktions passage gennem pipelinen. Betydningen af bogstaverne er:

- F: Fetch, instruktionhentning
- D: Decode, afkodning, læsning af registre, evt venten på operander
- X: eXecute, udførelse af ALU op, af adresseberegning eller af første del af multiplikation
- M: Memory, læsning/skrivning af data fra data-cache, midterste del af multiplikation
- Y: sidste del af multiplikation
- W: Writeback, opdatering af registre

Alle instruktioner passerer gennem de samme 6 trin. Multiplikation udføres over 3 pipeline-trin, E, M og Y. Et ubetinget hop udføres i D-trinnet, dvs den instruktion der hoppes til kan blive hentet i cyklussen efter. Et betinget hop udføres derimod først i X-trinnet.

Der er fuld forwarding af operander fra en instruktion til en afhængig instruktion. Instruktioner venter i D-trinnet indtil operander er tilgængelige.

Locality

Temporal locality

Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small time duration. Temporal locality is exploited by keeping recently used instruction and data values in cache memory and by exploiting a cache hierarchy. Or even in a register, not in memory at all.

This can for example be achieved by making a texttttsum-variable we keep referencing often.

Spatial locality

Spatial locality refers to the use of data elements within relatively close storage locations (nearby memory addresses, nearby sectors on a disk, etc.). Spatial locality is generally exploited by using larger cache blocks and by incorporating prefetching mechanisms (fetching items of anticipated use) into the cache control logic. This could for example be of the type:

```
for (int i = 0; i < 500; i++) {
    for (int j = 0; j < 500; j++) {
        arr[i][j];
    }
}
```

While the following would exhibit **bad** spatial locality:

```
for (int i = 0; i < 500; i++) {
    for (int j = 0; j < 500; j++) {
        arr[j][i];
    }
}
```

Operating System

Signals and interrups

Interrupts can be viewed as a mean of communication between the CPU and the OS kernel. Signals can be viewed as a mean of communication between the OS kernel and OS processes.
Interrupts may be initiated by the CPU (exceptions - e.g.: divide by zero, page fault), devices (hardware interrupts - e.g: input available), or by a CPU instruction (traps - e.g: syscalls, breakpoints). They are eventually managed by the CPU, which "interrupts" the current task, and invokes an OS-kernel provided ISR/interrupt handler.

Signals may be initiated by the OS kernel (e.g: SIGFPE, SIGSEGV, SIGIO), or by a process(`kill()`). They are eventually managed by the OS kernel, which delivers them to the target thread/process, invoking either a generic action (ignore, terminate, terminate and dump core) or a process-provided signal handler.
Multitasking is also implemented through exploiting a hardware interrupt. All drivers generally work by interpreting interrupts. Signals are used to communicate between processes.

Processes and threads

Both processes and threads are independent sequences of execution. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces. Child processes do not inherit their parent's threads.

Processes

Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

Threads

A thread is an entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

Demand paging

Demand paging is a method of virtual memory management. In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it and that page is not already in memory (i.e., if a page fault occurs). It follows that a process begins execution with none of its pages in physical memory, and many page faults will occur until most of a process's working set of pages are located in physical memory.
When a process tries to access a page, the following steps are generally followed:

- Attempt to access page.
- If page is valid (in memory) then continue processing instruction as normal.
- If page is invalid then a **page-fault trap** occurs.
- Check if the memory reference is a valid reference to a location on secondary memory. If not, the process is terminated (**illegal memory access**). Otherwise, we have to **page in** the required page.
- Schedule disk operation to read the desired page into main memory.
- Restart the instruction that was interrupted by the operating system trap.

The advantages of this (as opposed to loading all pages immediately) is that we only load pages that are demanded by the executing process, there will be more space in main memory so more processes can be loaded and there is less loading latency at program startup since less information is accessed from secondary storage.

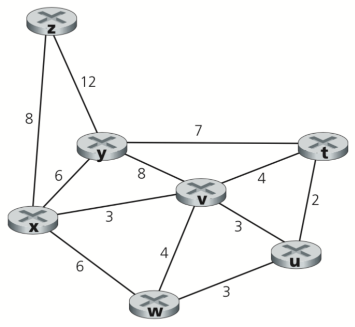
The disadvantages are that individual programs face extra latency when they access a page for the first time, memory management becomes slightly more complex and thrashing (when the virtual memory resources are overused, leading to a constant state of paging and page faults) might occur.

Computer Network

Link State (LS)

The algorithm where each router knows all distances and then each and everyone uses Djikstra's algorithm to compute distances.

Example computation for node x below:



Step	N'	$D(z), p(z)$	$D(y), p(y)$	$D(v), p(v)$	$D(w), p(w)$	$D(t), p(t)$	$D(u), p(u)$
0	x	8, x	6, x	3, x	6, x	∞	∞
1	xv	8, x	6, x		6, x	7, v	6, v
2	xvy	8, x			6, x	7, v	6, v
3	xvyw	8, x				7, v	6, v
4	xvywu	8, x				7, v	
5	xvywut	8, x					
6	xvywutz						

The forwarding table then becomes:

Dest node	Edge
z	(x, z)
y	(x, y)
v	(x, v)
w	(x, w)
t	(x, v)
u	(x, v)

Advantages:

- *Fast convergence times.* Link-state protocols use triggered updates and LSA (link-state advertisement) floods to immediately report changes in the network topology to all routers in the network.
- *Difficult for routing loops to occur.* Each router has a complete and synchronized picture of the network.
- Routers use the *latest information* to make the best routing decisions.

Disadvantages:

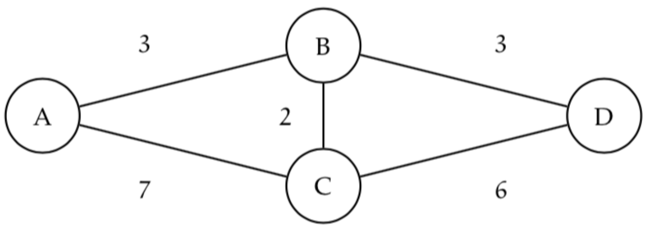
- Requires more memory and processor power than distance vector protocols.
- Requires strict hierarchical network design, so that a network can be broken into smaller areas to reduce the size of the topology tables.
- They flood the network with LSAs during the initial discovery process. This process can significantly decrease the capability of the network to transport data. It can noticeably degrade the network performance.

Distance Vector (DV)

The algorithm where each router only knows its own distances and then get sent the other distances from the rest of the routers.

Example computation:

Consider the network topology outlined in the graph below



Network Routing, 3.4.1: Apply the distance vector algorithm and compute the distance vectors on each node. Assume that the algorithm operates in a synchronous fashion (i.e., nodes simultaneously receive distance vectors from their neighbors, compute their new distance vectors, and inform their neighbors if their distance vector has changed.) (*Note: Remember to show the steps of the algorithm.*)

All the nodes compute their distance vectors based on costs of links to connected neighbors (as shown below) and send it to their neighbors.

	A	B	C	D
A	0	3	7	∞
B	3	0	2	3
C	7	2	0	6
D	∞	3	6	0

The distance vectors are recomputed on all nodes but only change on nodes A, C and D (as shown below) and the changed distance vectors are propagated to connected neighbors.

	A	B	C	D
A	0	3	5	6
B	3	0	2	3
C	5	2	0	5
D	6	3	5	0

Finally distance vectors are again recomputed (as shown below) which remain unchanged and hence the algorithm terminates.

	A	B	C	D
A	0	3	5	6
B	3	0	2	3
C	5	2	0	5
D	6	3	5	0

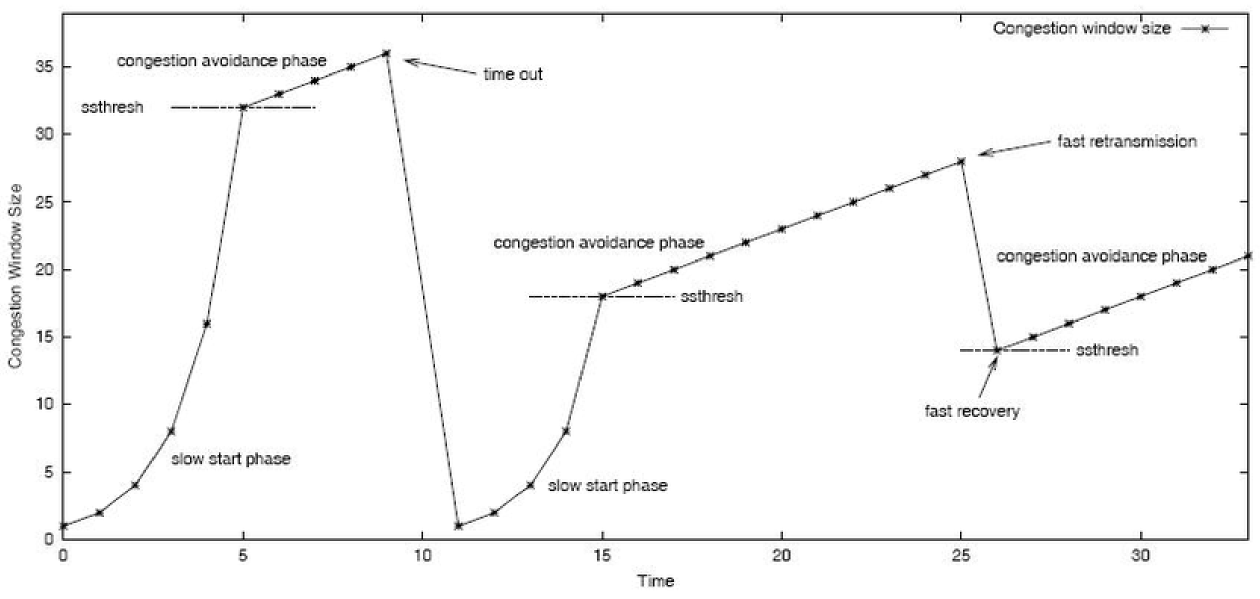
Advantages:

1. Does not need perfect and complete knowledge of the network before the algorithm can run.
2. Has a low messaging complexity since distance vectors are exchanged between connected neighbors only
3. Not a centralized protocol and each node can compute the distance vectors simultaneously and in an iterative fashion.

Disadvantages:

1. Has low speed of convergence (potentially never), can have routing loops and count-to-infinity problems.
2. Not very robust since a faulty router can potentially affect the distance vector computations on all routers.

TCP Congestion



In TCP, the congestion window is one of the factors that determines the number of bytes that can be outstanding at any time. The congestion window is maintained by the sender.

Slow start:
Although the strategy is referred to as slow start, its congestion window growth is quite aggressive, more aggressive than the congestion avoidance phase. The value for the congestion window size will be increased by one with each acknowledgement (ACK) received, effectively doubling the window size each round-trip time. The transmission rate will be increased by the slow-start algorithm until either:

- A loss is detected
- The receiver's advertised window (rwnd) ois the limiting factor
- *sshtresh* is reached

Congestion avoidance:
This is when we see linear growth. The window increases by 1 segment for each RTT (round-trip delay time). This keeps on going until we observe any loss'es/timeouts.

Timeout:
Happens when packets aren't received for a set amount of time. During timeout:

- Congestion window is reset to 1 MSS.
- *ssthresh* is set to half the congestion window size before the timeout.
- *slow start* is initiated.

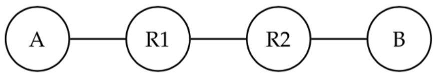
Fast retransmit and fast recovery:
Fast retransmit reduces the time a sender waits before retransmitting a lost segment. This happens when then sender receives *three duplicate ACK's.*
Then it can be reasonably confident that the segment with the next higher sequence number was dropped. A sender with fast retransmit will then retransmit this packet immediately without waiting for its timeout.

Difference between TCP Tahoe and TCP Reno:
TCP Reno is what is mainly used today. In Reno, if three duplicate ACKs are received, Reno wil perform a fast retransmit and skip the slow start phase and instead just halve the congestion window as described above.

In TCP Tahoe, we set *sshtresh* to half of the current congestion window (no matter if timeout or three duplicate ACK's) and then reset to slow start state.

IP fragments

The MTU (maximum transmission unit) on end host A and B is 1000 bytes while on the routers R1 and R2 are 500 and 250 bytes respectively. If an IP datagram of 1000 bytes is transmitted from end host A to B, explain how and where fragmentation and re-assembly of the datagram takes place in this example:



The fragmentation takes place on the routers R1 and R2 and the re-assembly takes place on end host B. Every datagram consists of 20 bytes of header which must be accounted for. Since IP datagrams consist of 20 bytes of header, the data portion of the IP datagram transmitted by A is 980 bytes.

First time (R1)

The datagram consisting of 980 bytes of data is split into three fragments in the following way:

Header	D_1	D_2	D_3
Length	500	500	40
Flag	1	1	0
Identifier (16 bit)	x	x	x
Offset (13 bit)	0	480	960

Second time (R2)

On router R2, we split the fragments D_1 and D_2 into six new again. This gives us:

Header	D_{11}	D_{12}	D_{13}	D_{21}	D_{22}	D_{23}	D_3
Length	250	250	40	250	250	40	40
Flag	1	1	1	1	1	1	0
Identifier (16 bit)	x	x	x	x	x	x	x
Offset (13 bit)	0	230	460	480	710	940	960

The 16-bit identifier x used in this is not specified, but it will be the same in *all* fragments.