Announcements

- reading
  - for Thursday 5.5
- Homework #1 (due 9/30/97 in class)
  - ch1, p.4 simple expression and explanation is fine
  - ch 2, p14: just use dvision (assume mean is extact)
- Programming Project #1 will be returned on Th.

Sending More Than one Signal At Once

- Called multiplexing
  – original goal of Bell was to MUX multiple telegraph signals
- Time Division Multiplexing
  – everyone gets whole bandwidth
  – but only when its their turn

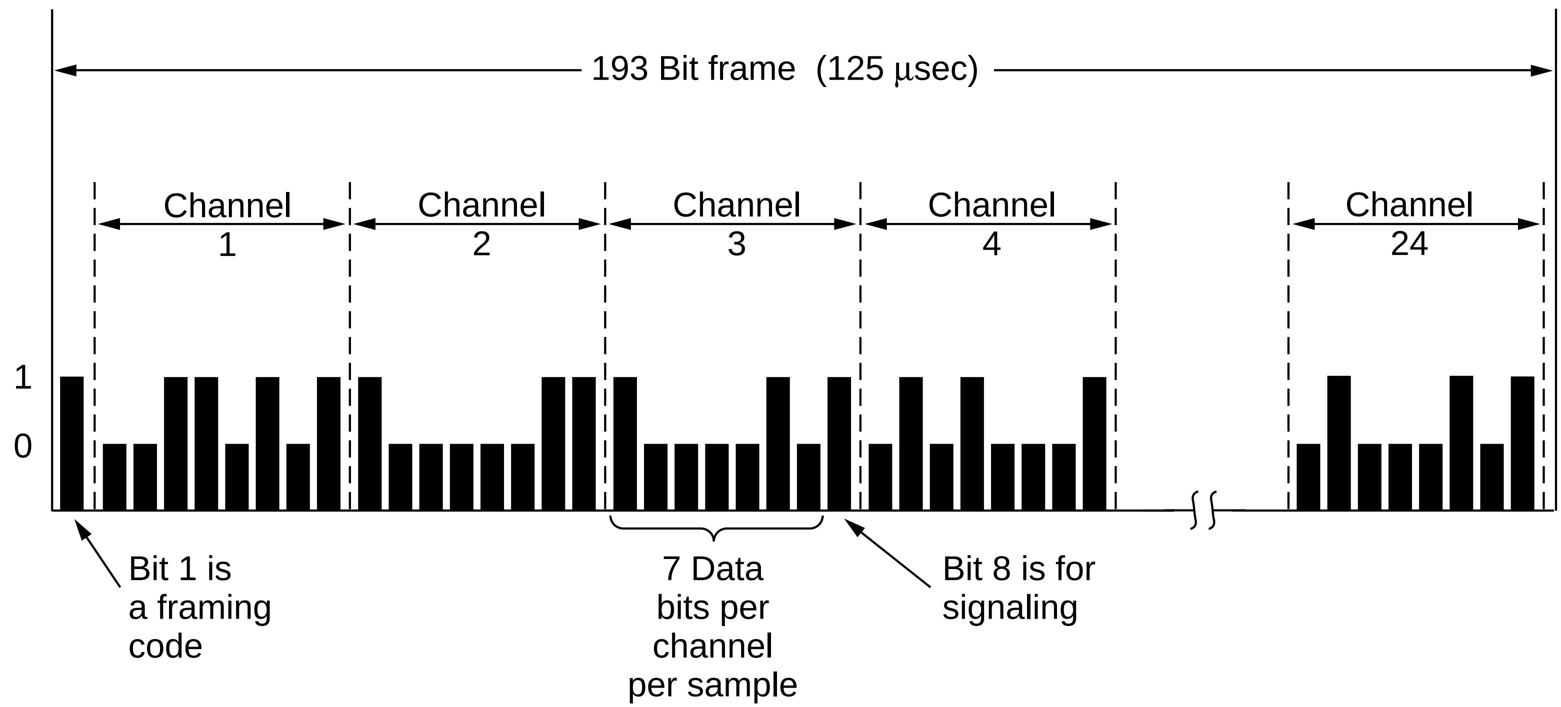
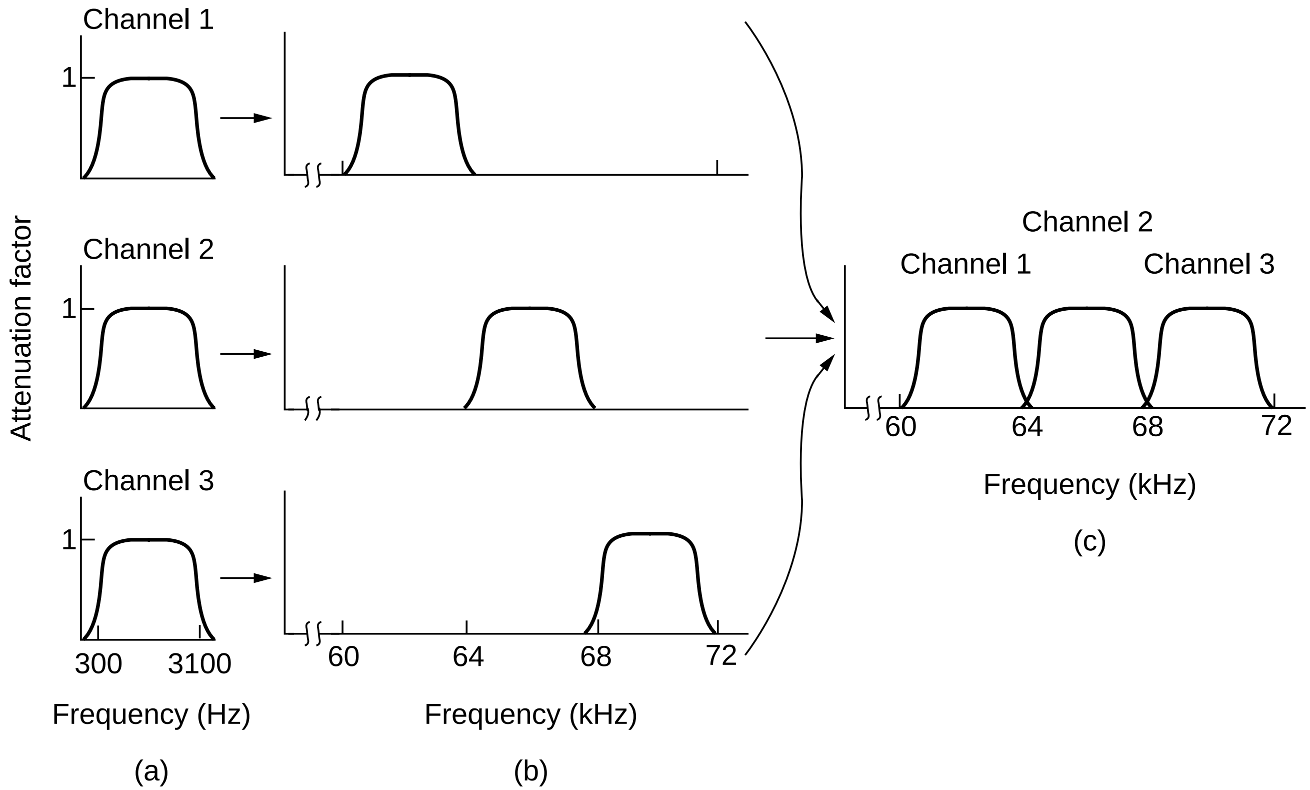


figure copyright , 1996, Andrew S. Tanenbaum

Frequency Division Multiplexing

- Frequency Division
  - everyone gets to talk at once
  - but only in their own frequency

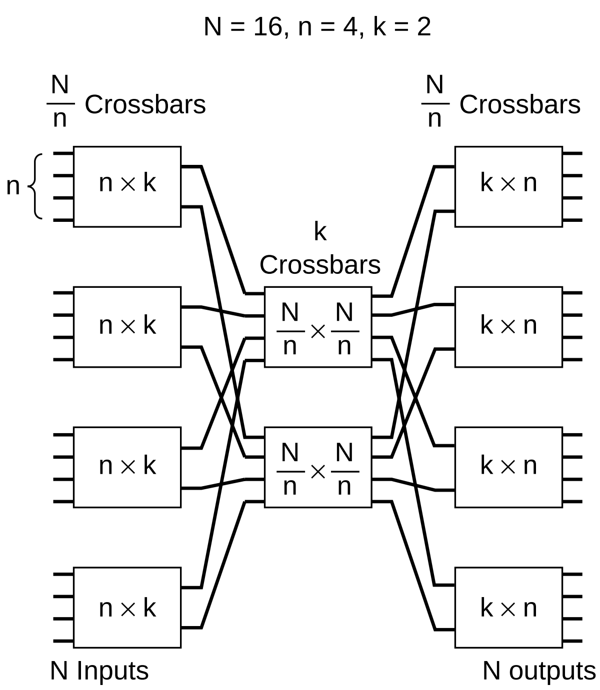


ATM Switching

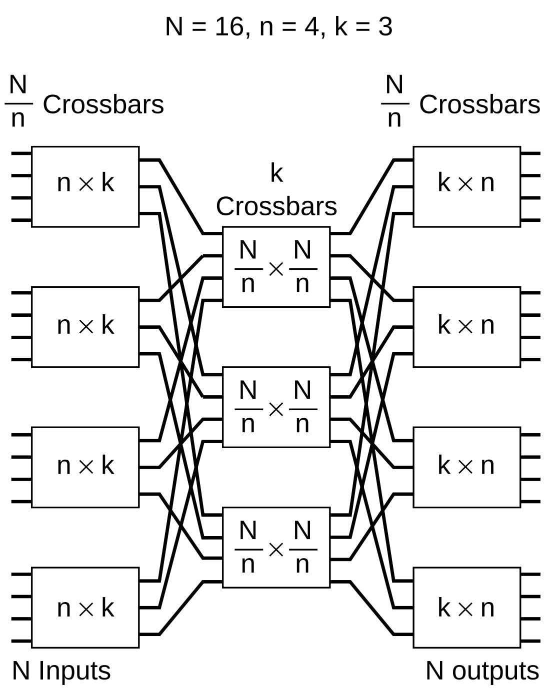
- Requirements
  - be able to switch 360,000 cells/sec per input link
  - switch cells with as low a discard rate as possible
  - never reorder the cells on a virtual circuit
- Issues
  - multiple cells destined for the same output at once
    - need to buffer one of them
    - must ensure fairness is maintained
  - head-of-line blocking
    - possible that a blocked output is holding up cells that could be delivered

Switching Fabric (space division)

- Cross bars are great, but require $O(n^2)$ wires
- Can use a collection of smaller cross bar switches
  - penalty: a request to connect may **block**



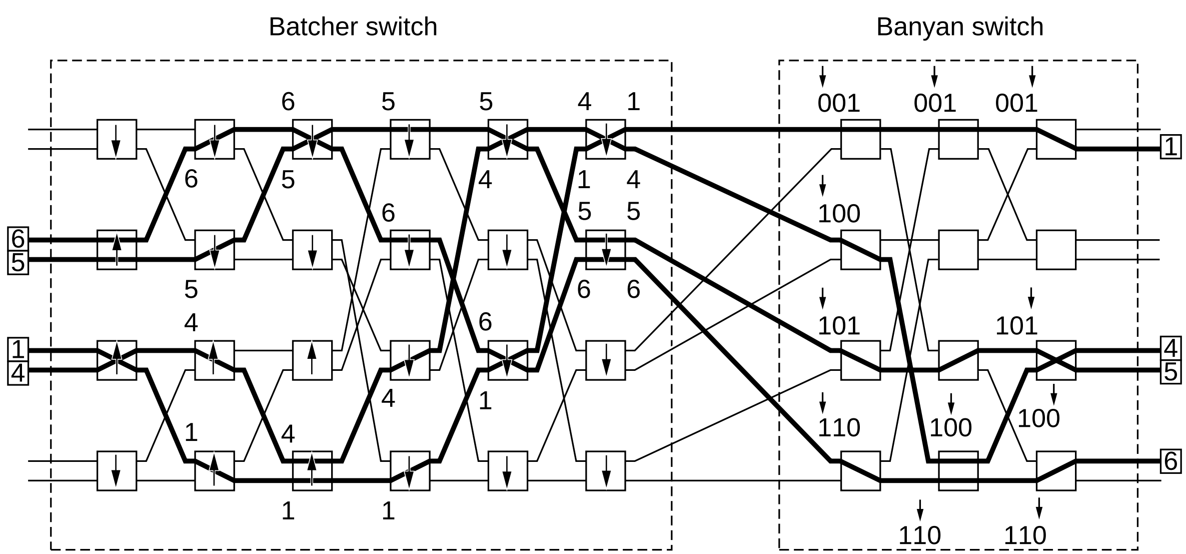
(a)



(b)

Batcher-banyan Switching

- Banyan
  - can do a “good” or “poor” job of switching due to collisions
  - if the inputs are sorted, we get performance
- Batcher
  - sorts traffic base on full address of destination
  - compares two colliding packets and uses final destination to select output port
  - requires $O(n\log^2 n)$ nodes (2x2 switching elements)



Introduction to Pthreads

- Often want multiple “threads of control”
  - separate logical acctivity (processing different requests)
  - can exploit multiple processors if they are available
- Threads
  - multiple execution streams that share an address space
    - premptive: each thread gets a timeslice
    - non-premptive: threads only switch on a block or a yield
  - similar to processes
- Need to share information
  - different threads are working on the same problem
  - goal: let them share all of their global and heap variables
  - problem: coordinating access

Producer-consumer: shared memory

- Consider the following code for a producer

```
repeat
     ….
     produce an item into nextp
     …
     while counter == n;
     buffer[in] = nextp;
     in = (in+) % n;
     counter++;
until false;
```

- Now consider the consumer

```
repeat
     while counter == 0;
     nextc = buffer[out];
     out = (out + 1) % n;
     counter--;
     consume the item in nextc
until false;
```

- Does it work? Answer: NO!

Problems with the Producer-Consumer Shared Memory Solution

- Consider the three address code for the counter

Counter Increment	Counter Decrement
reg_1 = counter	reg_2 = counter
reg_1 = reg_1 + 1	reg_2 = reg_2 - 1
counter = reg_1	counter = reg_2

- Now consider an ordering of these instructions

T_0	producer	reg_1 = counter	{ reg_1 = 5 }
T_1	producer	reg_1 = reg_1 + 1	{ reg_1 = 6 }
T_2	consumer	reg_2 = counter	{ reg_2 = 5 }
T_3	consumer	reg_2 = reg_2 - 1	{ reg_2 = 4 }
T_4	producer	counter = reg_1	{ counter = 6 }
T_5	consumer	counter = reg_2	{ counter = 4 } ← This should be 5!

Defintion of terms

- *Race Condition*
  - Where the order of execution of instructions influences the result produced
  - Important cases for race detection are shared objects
    - counters: in the last example
    - queues: in your project
- *Mutual exclusion*
  - only one process at a time can be updating shared objects
- *Critical section*
  - region of code that updates or **uses** shared data
    - to provide a consistent view of objects need to make sure an update is not in progress when reading the data
  - need to provide mutual exclusion for a critical section

Critical Section Problem

- processes must
  - request permission to enter the region
  - notify when leaving the region
- protocol needs to
  - provide mutual exclusion
    - only one process at a time in the critical section
  - ensure progress
    - no process outside a critical section may block another process
  - guarantee bounded waiting time
    - limited number of times other processes can enter the critical section while another process is waiting
  - not depend on number or speed of CPUs
    - or other hardware resources

Using Locks for the Critical Section

- Lock:
  - if no thread has the lock mark it locked and return
  - if another thread has the lock, wait
- Unlock:
  - release the lock
  - if other threads waiting, notify one **or** all of them
- Called mutexs in pthreads
  - pthread_mutex is the data type
  - pthread_mutex_init used to initialize it
  - pthread_mutex_lock locks it
  - pthread_mutex_unlock releases it
- Lock Grainularity
  - want to lock enough to protect accesses
  - don't want to lock too much to slow down the program