F.M. STAJANO

ENGINEERING TRIPOS PART IIA

Module 3F5

COMPUTER AND NETWORK SYSTEMS

Course: Michaelmas Term 2002
Exam: 6 May 2003

Crib

Version 3

1 (a) **[40%]**

A circuit switch is normally used in telecommunications networks, where minimising delay across the network are important. This is the case when considering either video or voice services within a network. A circuit switch is a connection oriented switch which means that the connection is guaranteed for the duration of the call procedure. In the case of a voice service, this means that the connection is maintained throughout the duration of the call. This originally meant that there was an individual 'circuit' used for each possible connection, however in the modern SDH system this means that there is a guaranteed 64kbits/sec channel held open for the duration of the call. One of the key features of a circuit switch is that it is always controlled by the external management or control system of the network and does not depend on the data that is passing through it. If computer data is passed through a circuit switch, then there is a large amount of wasted bandwidth.

A packet switch is a switch which is set by the data passing through it and it can be either connection oriented (as is the case when virtual channels are allocated) or connectionless as is the case when datagrams are used (as in the internet). A packet switch usually controls access on and off a high bitrate medium via statistical multiplexing using either predefined virtual or logical channel, or by using the routing information contained within each packet. A key point in packet switches is that the switch state depends on the data passing through it, hence if there is a bottleneck, then data must be buffered and therefore it is very difficult to control packet delays. For computer data this is not a problem, however for voice and video service, this is a serious limitation.

(b) (i) **[10%]**

The delay due to the transmission line will be $25000/1x10^8 = 0.25$msec

A single packet will take $1024x8/1x10^6 = 8.2$msec
Delay per packet = 8.2 + 0.25 + 2 = 10.45msec
Acknowledgement will be sent every 8 x 10.45 = 84msec

(ii) **[10%]**

Each packet now takes 18.2msec therefore the acknowledgement will now be sent every 148msec. This is a 57% increase in the acknowledgement delay which is purely due to the packet switch buffering

(iii) **[10%]**

If an error occurs, then this will be detected by the receiver and then a retransmission request is sent by the receiver and the receiver waits until the error free packet arrives. This means that the packets after the error will be delayed by at least 2x18.45msec + the time taken for the retransmission request (at least 8.25msec).

(c) **[30%]**

The wide area link would be Ok for voice as long as no errors occurred, as the acknowledgement would only take 8.25msec to transmit and will be a short packet. This will create a packet delay of less than 26msec which is tolerable for voice transmission. However, if only a single error occurs, then this delay is exceeded and could lead to clicks and disturbances on the phone line. Even given an error probability of 10^{-5}, this would not be acceptable delay service for telephony, given the error rate of transmission over a noisy medium like coaxial cable.

On top of this, the use of TCP/IP would cause problems as well, as TCP uses a large number of short 40 byte packets to set up and monitor transmissions, hence the tolerable delay would be exceeded by the acknowledgement window.

There are several ways this can be improved:

- Reduce the need for error detection and correction by using better a medium such as a single mode optical fibre. This also makes the windowing redundant.
- Don't use TCP, use a custom VoIP protocol stack
- Use faster packet switches

END OF PAPER

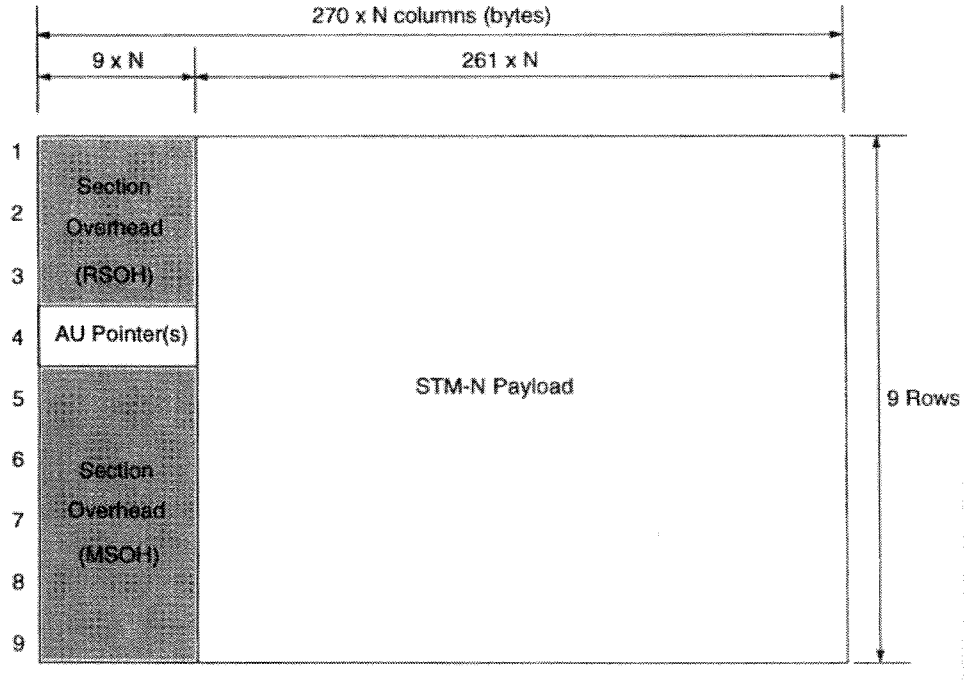
2 (a) **[30%]**

The aim of SDH was to correct the defects of PDH.

- Synchronisation. All elements of the system are synchronised to the same master clock. All tributaries are at a common rate before Mux. Hence only one level of bit stuffing is required.
- Pointers. Information about the muxed signals is transmitted at fixed intervals, which indicate the positions of units within the mux process. Hence any unit within the mux process can be identified and drop and insert processes can be done dynamically from an external viewpoint.
- Control and management. This is done outside the transmission process allowing complete control of the mux process. Time slots are put aside for a variety of tasks in ensuring the synchronicity of voice and data services.

(b) **[30%]**

STM-1 signal is a repeated series of 125usec frames of 270x9 (2430) bytes. (Same as 1x64kbit/sec byte to avoid delays).



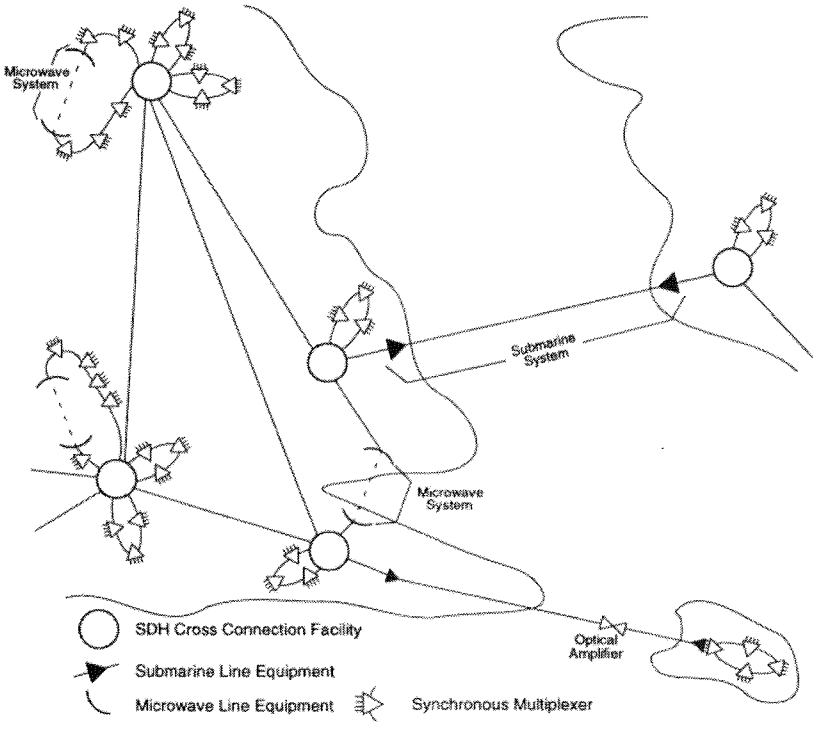
A 2D frame format is used as it effectively interleaves the pointer information with the data payload. This means that synchronicity can be easily implemented by adding special markers in the overhead sections. It also helps to minimise possible effects of delays.

Data rate $270x9x8/125x10^{-6} = 155.52$ Mbits/sec

(c) **[40%]**

An idealised synchronous transmission network contains not only SDH fibre optic rings, but also different protocol systems like submarine cables, satellite and microwave.

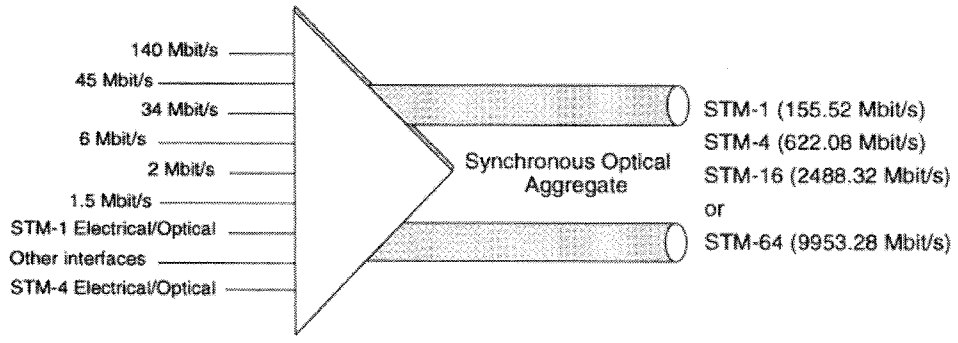
In this system, the synchronous OXC is used to connect together a series of self healing rings which also contain add/drop Mux units.



Synchronous multiplexer.
Allows optical integration at the STM-N level. Optical interfaces are often repeated for redundancy as main/standby or East/West pairs. Often in a ring topology.

END OF PAPER

Must offer digital mux to STM-1 with flexible inputs.



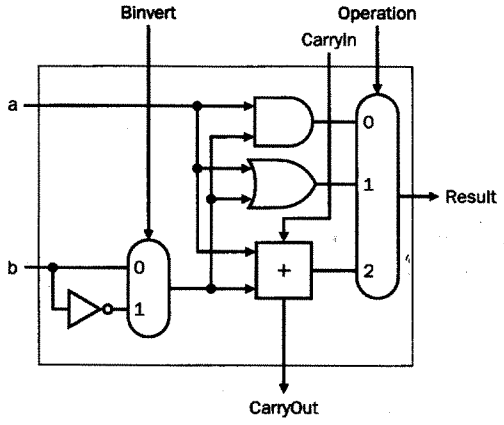
Synchronous, Plesiochronous, LANs/FDDI and MAN access, B-ISDN and ISDN, ATM, IP

The synchronous cross-connect (TDM switch)
These are used to reconfigure the network by setting up semi-permanent connections at the VC level. This is mostly a time slot re-arrangement operation, as the VC's are 'virtual' containers. It could also involve space switching between different network nodes.
Connections are set by the network manager, not by the user. (As would be the case in the old telephony example).

The SDH OXC can be done with a sync mux, but there are two examples of stand alone OXC units.

This not the case for a real SDH network as there was an existing PDH system which had to be replaced gradually. The PDH system was built on existing local office exchange systems, hence the SDH is far from ideal and has a residual tiered structure due to the older systems.

3 (a) (i) **[10%]**



(ii) **[20%]**

The SLT instruction has this format: SLT $r1, $r2, $r3. If $r2 < $r3 it sets $r1 to 1, otherwise it resets it to 0.

First I add an extra input, *less*, to the *operation* multiplexer of the 1-bit ALU. This comes from an external line. A 32-bit 1 or 0 has all its bits set to 0 except for the LSB, so I set the *less* input on the top 31 ALUs to 0.

On the LSB ALU (bit position 0) I want to set it to the sign of the result when I perform the subtraction a-b.

Without overflow, the sign of the result is the MSB, which I could get from the topmost ALU (bit 31).

However we said that that ALU must return 0 when performing SLT, so I need to export the result of the adder of that ALU through another line. This will be fed back to the *less* input of the LSB ALU.

(iii) **[20%]**

Let's call them all additions, with the operands being two's complement signed numbers, since that's what the ALU does anyway.

If the two operands are of the same sign: if their sum is of the opposite sign, we have overflow, otherwise not.

If the two operands are of opposite signs, then their sum can never cause an overflow.

No other cases exist.

So, looking at the operands (a, b) and sum (s) of the 1-bit full adder in the MSB ALU, we have overflow iff abs' OR a'b's.

END OF PAPER

This circuitry is included just in the MSB ALU, which therefore gets an extra output to signal overflow. No other ALU is affected.

(iv) **[20%]**

As before, we want to be able to derive the boolean result a<b by computing a-b. We used to say: a<b if and only if a-b is negative. This is true if no overflow.
If there is overflow, the sign bit of a-b is incorrect—it's the opposite of what it should be. So we say: in case of overflow, a<b iff a-b is positive.
Calling n the sign bit of the MSB ALU, and v the overflow flag, we have a<b = nv' OR n'v. This is equivalent to n XOR v.

(b) **[30%]**

For each bit position, carry look-ahead generates two signals, generate and propagate, in constant time (independent of the position). From these, the carry to the next bit can also be generated in constant time.

"Generate" is true if and only if this bit position generates a carry for the next position, regardless of what the previous positions do. "Propagate" is true if and only if this bit position outputs a carry-out assuming it receives one on carry-in.

gi = ai bi
pi = ai + bi

The carry-out is true if we generate a carry, or if we propagate it and we receive one from below.

ci+1 = gi + pi ci

To avoid cascading as in ripple-carry, we expand this equation so that it's computed in constant time for all bit positions.

c1 = g0 + p0 c0
c2 = g1 + p1 g0 + p1 p0 c0
c3 = g2 + p2 g1 + p2 p1 g0 + p2 p1 p0 c0
c4 = g3 + p3 g2 + p3 p2 g1 + p3 p2 p1 g0 + p3 p2 p1 p0 c0 [10%]

4 (a) **[20%]**

main symbol	**t**	**execution time (seconds)**
	s	**speedup (dimensionless)**
	u	**usage (dimensionless)**
"what" subscript	**p**	**whole program**
	f	**feature being improved**
	r	**rest of program**
"when" subscript	**o**	**old**
	n	**new**

$$s_p = \frac{t_{po}}{t_{pn}};\ s_f = \frac{t_{fo}}{t_{fn}};\ u = \frac{t_{fo}}{t_{po}}.$$

From the definitions above, some straightforward algebra coupled with some basic observations (such as tro = trn, since r is the part that wasn't speeded up) yields $s_p = \frac{1}{1-u+\frac{u}{s_f}}$. The derivation is easy, the only problem being the abundance of variables.

As for the extreme cases:
If sf = 1 (no speedup for the feature): sp = 1 (no speedup for the program). OK.
If sf = ∞ (feature is made infinitely fast): sp = 1/(1-u). From this I can derive that tpn = trn, ie that tfn = 0 (feature takes no time). OK.

(b) **[10%]**

sp = 3; sf = 5; plugging this in the formula derived in (a) yields an equation in u, from which u = 5/6. (This figures: 25 minutes for memory, 5 for rest; when memory down to 5 minutes, 5+5=10.) Speedup of 4 times means total time is 7.5 minutes; minus 5 for rest, this leaves 2.5 minutes for memory, which therefore must be speeded up by 10 times from its original 25 minutes. [10%]

END OF PAPER

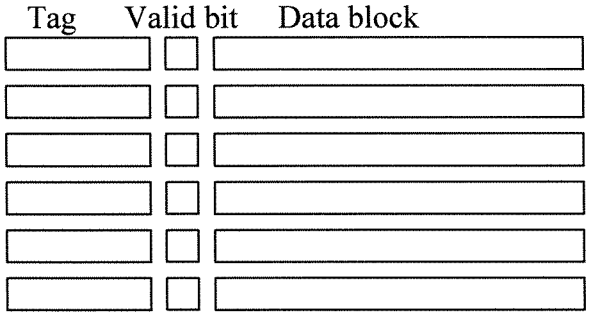
(c) **[30%]**

A cache is a piece of expensive but fast memory in which we store a subset of main memory. When we need data, if we find it in the cache rather than in memory (cache hit), we save time. If it's not in the cache (cache miss) we must get it from main memory so there is no benefit.

Caching only works if the data access pattern exhibits temporal or spatial locality. To exploit temporal locality, we cache recently used data in the hope it will be reused again. To exploit spatial locality, on cache miss we fetch entire blocks as opposed to individual cells, in the hope that nearby locations will also be referenced.

In steady state the cache is full, so fetching a new block requires flushing an existing one. If the flushed block is referenced again later, it has to be re-fetched. So, ideally, we want a replacement strategy that will not flush blocks that will be referenced again. This guesswork is hard.

Writing to the cache is fast, but we must ensure that any changed data is written back to main memory before being flushed. There are two main strategies: write-through (every time you write to the cache you also write to memory), simpler and slower; and write-back (just write to the cache, only updating main memory when you must flush a "dirty" block), faster but more complex to implement.



A cache entry contains at least a block of data, a valid bit (reset to 0 on startup, set to 1 once the entry has been filled with valid data) and a tag. Since many memory blocks map to the same cache entry, the tag contains the part of the block address that is not implied by the cache index of the entry. Without the tag it would be impossible to know whether the valid block held in that entry is the one corresponding to the memory address you are trying to access, or just one of the many others whose address aliases to the same cache index position.

(d) **[20%]**

Given a block of memory to be cached, where should we put it?

- Direct-mapped: address of block determines a unique cache index.

- Set-associative: cache is partitioned into sets of blocks. Each set holds the same number of blocks (typically 2 or 4). Address of block determines a unique set, but block can go anywhere in the set.
- Fully associative: block can go anywhere in the cache.

Advantages and disadvantages:

- Direct-mapped: easiest to implement and fastest to access, but inefficient because it may replace blocks that are going to be referenced again soon.
- Fully associative: great flexibility in the replacement strategy and therefore optimum use of cache space, but high implementation cost: if any block can go anywhere, the whole cache must be searched at every access. Doing this in hardware, and in parallel, is very expensive except for tiny caches. Better suited for virtual memory.
- Set-associative: a compromise. Mitigates the direct-mapped problem of replacing a block too soon, but at the same time the implementation is not as expensive as that of a fully associative cache.

(e) **[20%]**

Given a memory address, all the elements of the set to which the address maps must be available, so that their tags can be compared in parallel. So the width of the cache in bits is $2\times(t+v+d) = 2\times(t+1+32) = 2t + 66$, where t, v, d are the widths of the tag, valid and data fields respectively.

To provide 66 bits with 4-bit wide chips I need at least 17 of them. But I can't fit 34 chips on the board, so the depth of the cache will be the depth of one chip, ie 64 k entries.

So what's the tag size? The cache index is log2(64k) = 16 bits wide; in main memory the address of a (word-sized) block uses 30 bits, so the tag is 30-16=14 bits. So the width of the cache is 2x14+66=94 bits. Since 94/4=23.5, I need 24 chips.

Each cache set spans the whole array of 24 chips, taking 4 bits from each. For each of the two cache entries in the set, 8 chips store the 32 bits of data and another 4 chips store the 14 bits of tag—with 2 bits to spare, one of which is used for the valid bit while the last one is wasted.

There are 64 k sets. Each set provides 2 words (8 bytes) of data storage, so the net capacity of the cache is 8 B x 64 k = 512 kB. (The gross capacity of the 24 chips, by comparison, is 24 x 4 x 64 k / 8 = 768 kB.)

The 32 bit address breaks down into the following fields, from least to most significant: 2 bits for byte offset in word, 0 bits for word offset in block, 16 bits for cache index, 14 bits for tag.

END OF PAPER