

INMOS OCCAM AND TRANSPUTER

Advanced Training Course

6 to 10 July 1987

The aims of this course can be listed in order of priority:

Programming more than one transputer

Debugging multiprocessor programs

Software design and test methodologies

Hardware design and test methodologies

Maximising performance

To attend this course *everybody* must have experience in using occam and the TDS. This prime requisite will be assumed as no time is allocated for teaching basic occam programming.

Registration is between 11am and 12am monday. Lunch is provided 12.30 to 1.30 monday to friday. All morning sessions are from 9.30 to 12.30 with coffee provided at about 10.30. All afternoon sessions are from 1.30, with coffee/tea at about 3.30 to officially 4.30 with practicals extended where necessary. The course finishes after lunch on the Friday.

The attendees will work in groups of two having access to an IBM PC development system and transputers.

The agenda for the course is as follows:

1 Monday morning

System installation.
LJP+ST Deciding practical pairings.
LJP+ST testing B004s and networks

2 Monday Afternoon

PGM+LJP Course Introduction
LJP+PGM Variant protocols
LJP+PGM User defined libraries
LJP+PGM EXEs and I/O libraries
PGM+LJP Transputer hardware overview.
PGM+LJP Transputer booting and initialisation.
PGM+LJP Hardware testing.
PGM+LJP Practical -- Peek and Poke.

3 Tuesday Morning

PGM+LJP Software design
LJP+PGM Harness description.
LJP+PGM Practical -- Run time error location.
PGM+LJP Practical -- Memory testing/board faults

4 Tuesday Afternoon

PGM+LJP Hardware design.
PGM+LJP Software testing.
PGM+LJP Practical -- Multi-processor memory testing
LJP+PGM Practical -- Network analysis.

5 Wednesday Morning

LJP+PGM The transputer instruction set.
LJP+PGM The preamble
PGM+LJP Practical -- GUY code usage

6 Wednesday Afternoon

PGM+LJP Program performance.
LJP+PGM Practical -- timing block moves
off-chip, on-chip variations

7 Thursday Morning

LJP+JP TDS EXE interface.
LJP+JP Practical -- EXE list.file.
LJP+JP TDS file formats

8 Thursday Afternoon

JP+LJP The TDS server
JP+LJP Making stand alone applications
LJP+JP Practical -- stand alone EXE list.file
LJP+JP Practical -- stand alone memory tester

9 Friday Morning

LJP+GS The T800 floating point transputer.
GS+LJP The C004 link switch.
GS+LJP Modules.
GS+LJP Future product summary.



ADVANCED TRANSPUTER & OCCAM TRAINING COURSE

6-10 JULY 1987

NAME	COMPANY
BARTKOWIAK John	Ferranti Defence Systems, UK
BARTLETT Richard	Ferranti Computer Systems, UK
BINKS J	Transmitton Ltd, UK
BUSTANY Francois	Digilog, France
COSNUAU Alain	Onera, France
DE JOYBERT Xavier	Cimsa Sintra, France
HORVATH Juri	INMOS Germany
HUIKAMP W	Physics & Electronics Lab-TNO, Netherlands
JAKOBSEN Jens	Jutland Telephone Company, Denmark
KJAERGAARD Jens	R.C. Computer, Denmark
MORIGAMI Toshiyuki	INMOS Japan
NIININEN Mauri	Instrumentarium Corp, Finland
POWELL Mark,	Marconi Radar, UK
RABHAN Thierry	INMOS France
REDMAN Andrew,	Marconi Radar, UK
TURNER John	Marconi Instruments Ltd , UK
UNDERBJERG Carsten	R C Computer, Denmark
<u>WADSWORTH C P Dr</u>	Rutherford Appleton Laboratory, UK
WERNER Hett	Ingenieurschule Biel, Switzerland
YLA-OUTINEN Esa	Instrumentarium Corp, Finland

CHANNEL PROTOCOLS

The programmer must specify the *type* of message on a channel.

The format is .. `CHAN OF protocol name :`

a *protocol* can be of the following types:

- **ANY**, totally unchecked
- **Simple**, data is of a specified type
- **Named**, using a **PROTOCOL** specification

a named **PROTOCOL** can be

- **Sequential**, a sequence of simple protocols
- **Variant**, an option of more than one simple or sequential protocol.

SIMPLE PROTOCOLS

`CHAN OF REAL32 comm:`

`PAR`

`REAL32 x:`

`comm ? x`

`comm ! 3.14156 (REAL32)`

`CHAN OF [3][2]INT arrays:`

`PAR`

`[3][2]INT x:`

`arrays ? x`

`arrays ! [[1,2],[3,5],[7,9]]`

`CHAN OF INT::[]BYTE pipe:`

`PAR`

`[80]BYTE string:`

`INT length:`

`pipe ? length::string`

`pipe ! 11::"hello world"`

NAMED SEQUENTIAL PROTOCOLS

PROTOCOL *name* IS { ; *simple.protocol* } :

name can now be a protocol in a CHAN specification.

```
PROTOCOL record IS  BOOL ; INT::[]BYTE ; REAL32 :  
CHAN OF record in, out:  
PAR
```

```
    ... other processes  
    INT length:  
    [256]BYTE string:  
    REAL32 number:  
    BOOL flag:  
    SEQ  
        in ? flag; length::string ; number  
        IF  
            flag  
                out ! TRUE; length::string ; number  
        TRUE  
            out ! FALSE; 4::"quit" ; 0.0 (REAL32)
```

NAMED SEQUENTIAL PROTOCOLS 2

PROTOCOL two.strings IS BYTE::[]BYTE; BYTE::[]BYTE :

[2]CHAN OF two.strings c:

PAR

... process inputting from channel c

VAL str IS "I wandered lonely as a cloud":

SEQ

```
    c[1] ! BYTE 5::"hello"; BYTE 6::"friend"  
    c[0] ! BYTE 4::"poet"; BYTE (SIZE str)::str
```

Note N::string will use N::[string FROM 0 FOR N]

and not necessarily all of string.

VARIANT PROTOCOLS

A single channel can carry messages of different formats by specifying it to have a variant protocol.

PROTOCOL *name*

UU CASE

tag.name ; *protocol*

tag.name ; *protocol*

... as many tags as required

:

For outputting processes ...

channel ! *tag.name* ; *protocol*

For inputting processes ...

channel ? CASE

tag.name ; *protocol*

process

tag.name ; *protocol*

process -- and so on for all the tags

VARIANT PROTOCOL EXAMPLE

PROTOCOL *message*

CASE

two.ints ; INT ; INT

string ; BYTE::[]INT

:

CHAN OF *message comm*:

PAR

... *process* outputting on *comm*

INT *x*, *y*:

BYTE *size*:

[256]INT *vector*:

comm ? CASE

two.ints ; *x*; *y*

UU ... *process* dealing with the two INTs

string ; *size*::*vector*

UU ... *process* dealing with the *string*

VARIANT PROTOCOL INPUT EXAMPLE

```
PROTOCOL packet
CASE
  data ; REAL32
  terminate
:
CHAN OF packet connect:
PAR
  ... other processes
  BOOL running:
  REAL32 number:
  SEQ
    running := TRUE
    WHILE running
      connect ? CASE
        data ; number
        ... use the number
        terminate
        running := FALSE
```

VARIANT PROTOCOL OUTPUT EXAMPLE

```
PROTOCOL packet
CASE
  data ; REAL32
  terminate
:
CHAN OF packet connect:
PAR
  ... process inputting from connect
  REAL32 n:
  SEQ
    n := 2000 (REAL32)
    WHILE n < 1000000 (REAL32)
      SEQ
        connect ! data; n
        n := n + 4.5 (REAL32)
      connect ! terminate
```

TAG ONLY PROTOCOLS

```
PROTOCOL command
CASE
    execute
    move
    terminate
:
CHAN OF command orders:
PAR
    ... process issuing commands
    BOOL running:
    SEQ
        running := TRUE
        WHILE running
            orders ? CASE
                execute
                ... run batch task
            move
                ... pass on data item
            terminate
            running := FALSE
```

ALT INPUT SELECTION EXAMPLE

A CASE input may be used as an ALT guard.

```
INT x:
ALT
    channel.from.controller ? CASE
        message
        ... route through to correct output
        terminate
        ... terminate this process
    ALT i = 0 FOR number.of.applications
        channel.from.application[i] ? CASE
            data.item ; x
            ... deal with data value x
            error.message ; x
            ... print error.message
        peripheral ? x
        ... pass on to relevant receiver
```

LIBRARIES IN OCCAM

The format for a library is :

```
{{{  LIBRARY.TOP
...F myfiles1.tsr
...F myfiles2.tsr
{{{F myfiles3.tsr
{{{  LIB  interface
...   library ID
...   SC keyboard.tsr 13/2/87
...   SC screen.tsr   19/2/87
{{{
}}}
...F myfiles4.tsr
}}}
```

The contents may be any number of SCS.

Future implementations will allow VALs and PROTOCOLS.

USING LIBRARIES IN OCCAM

Libraries are created using the *make.foldset* utility.
The *make.foldset* on an empty fold adds an attribute **LIB**.

Compiled SCS (and VARS) are put inside the foldset.
Repeating *make.foldset* on the **LIB** fold validates it.

The **LIB** fold should then be put into a named file.
The filename used becomes the reference to this library fold.

Any occam may now reference this library
without copying the library in question each time eg

```
PROC test (CHAN OF ANY keyboard, screen)
  #USE "myfiles1.tsr"
  #USE "myfiles2.tsr"
  SEQ
    ... body of test
:
```

ACCESSING THE HOST THROUGH THE TDS

An EXE compilation fold has fixed parameters that allow the user to access terminal and filing facilities using the TDS.

```
{{{  EXE  a user program
{{{F source file
PROC example (CHAN OF ANY keyboard, screen,
              [4]CHAN OF ANY fromUserFiler, toUserFiler)
UN ...  source text for example
:
}}}
```

These parameters are fixed in order as above, however, if no files are required a subset can be used.

```
PROC terminal.example (CHAN OF ANY clavier, ecran)
...  source text
:
```

SIMPLE SCREEN OUTPUT

```
{{{  I/O PROC declarations
VAL tt.out.string IS BYTE 8:
VAL tt.out.byte   IS BYTE 18:

PROC write.char (CHAN OF ANY out, VAL BYTE ch)
  out ! tt.out.byte; ch
:

PROC write.full.string (CHAN OF ANY out,
                       VAL []BYTE str)
  out ! tt.out.string; SIZE str; str
:

PROC newline (CHAN OF ANY out)
  write.full.string (out, "*n*c")
:
}}}
```

WRITING INFORMATION TO THE SCREEN

```
{{{ some simple screen output
SEQ
  write.full.string (screen, "Hello World")
  newline (screen)
  newline (screen)
  write.full.string (screen, "Press any key")

  INT trash:
  keyboard ? trash  -- await key so text can be read
}}}
```

folded into:

```
... some simple screen output
```

STRUCTURE REQUIRED FOR AN EXE

```
{{{ EXE my code for "banner"
{{F banner.tsr
PROC banner (CHAN OF ANY keyboard, screen)
  ... I/O PROC declarations
  ... some simple screen output
:
}}}
```

For the current implementation the source must be filed.

The compiler is invoked on the fold :

```
... EXE my code for "banner"
```

The *get.code* and *run.process* keys can then be used.

CALLING THE USER I/O LIBRARY

All the I/O declarations above are in a system library.
The library file is `c:\tdsiolib\userio.tsr`

To use the library replace the previous example with:

```
{{{ EXE my code for "banner"
{{F banner.tsr
PROC banner (CHAN OF ANY keyboard, screen)
  #USE "c:\tdsiolib\userio.tsr"
  ... some simple screen output
:
}}}
```

Another useful procedure to remember is :

```
write.int (CHAN OF ANY screen, VAL INT number, field)
```

USEFUL EXTRAS NOT IN THE LIBRARY

```
VAL tt.goto      IS BYTE 5:
VAL tt.clear.eos IS BYTE 10:
```

```
PROC move.cursor (CHAN OF ANY out, VAL INT x, y)
  out ! tt.goto; x; y
:
```

```
PROC clear.screen (CHAN OF ANY out)
  SEQ
    move.cursor (out, 0, 0)
    out ! tt.clear.eos
:
```

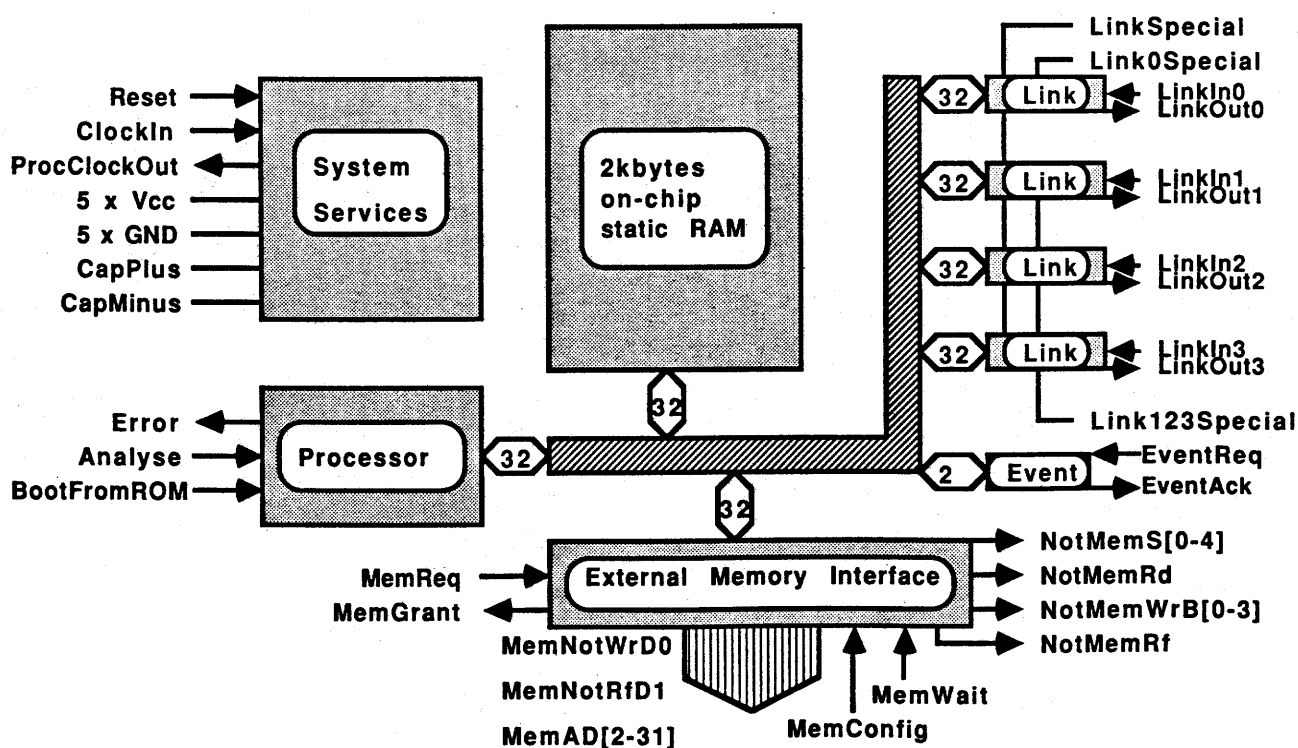
RUNNING YOUR EXE

First use *get.code* F5 to load your EXE process.

Then use *run.process* F6 which will do the following:

- 1 The TDS clears the screen
- 2 The TDS moves the cursor to the top left corner
- 3 The user process runs
term.p tags can be sent to the TDS with accompanying protocol
The EXE may read keyboard integer values if required.
The user filer is available via the other EXE parameters.
- 4 On EXE termination the TDS terminates term.p
- 5 The TDS returns to the editing environment at the position when *runprocess* was called.

IMS T414 Block Diagram



RESET AND ANALYSE

A transputer can be reset in two different ways.

Pulsing **reset** with **analyse** low

The transputer initialises

then boots

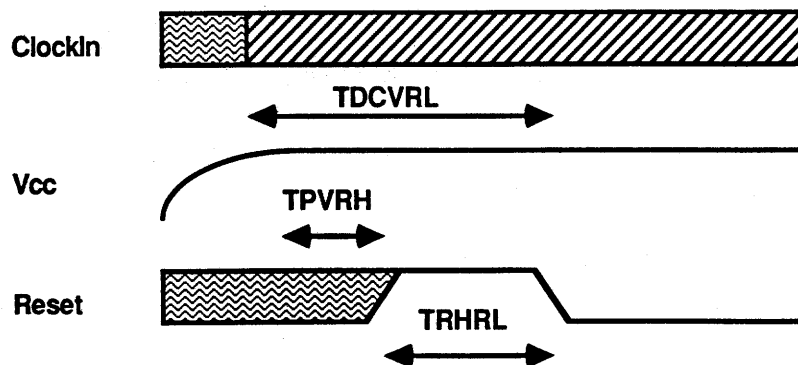
Pulsing **reset** holding **analyse** high

The transputer boots without initialising

state information is preserved

INITIALISATION OF TRANSPUTERS

(ANALYSE LOW THROUGHOUT)



TDCVRL Clock valid before reset, 10ms
 TPVRH Power valid before reset, 10ms
 TRHRL Reset pulse width, 8 clock periods

The transputer boots on the falling edge of reset.

INTERNAL CONFIGURATION CYCLE

64 cycles from internal scan of 144 clock cycles.
Searching for External Memory Interface configuration.

Scans from **MemNotWrD0** to **MemAD31**.
They all start high and are taken low in sequence.

Scanning for a simple connection to **MemConfig**.
A connection will select an internal configuration.

Connecting **MemConfig** to **Vcc** is the same as **MemAD31**.
Only 13 preset configurations are valid.

EXTERNAL CONFIGURATION CYCLE

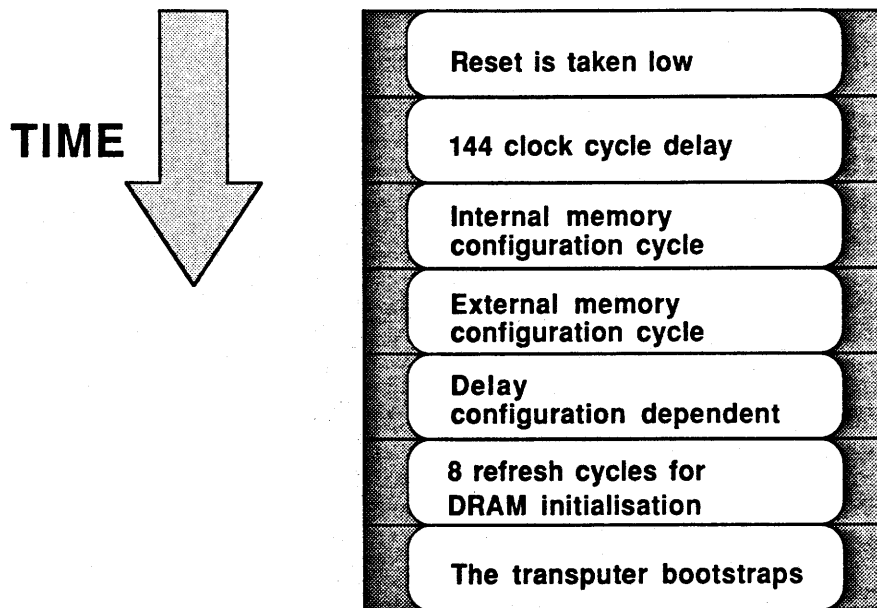
Performed every time, even if internal config chosen.
For external choice, **MemConfig** low until **MemNotWrD0** low.

36 successive read cycles from fixed ROM locations.
Default slow preset configuration of **MemAD31**.

Connect **MemConfig** via inverter to an address pin.
Single bit data is read on **MemConfig**.
One ROM may contain 32 configurations.

The External Configuration Coding is in the T4 datasheet (pp 31).

ACTION AFTER RESET TAKEN LOW



LJPMAC/init/02

BOOTING A TRANSPUTER

The action on booting depends on **BootFromROM** pin.

BootFromRom held high

The transputer jumps to address **#7FFFFFFE**.
The high positive addresses are for ROM.
Execution starts, always a backwards jump.

BootFromRom held low

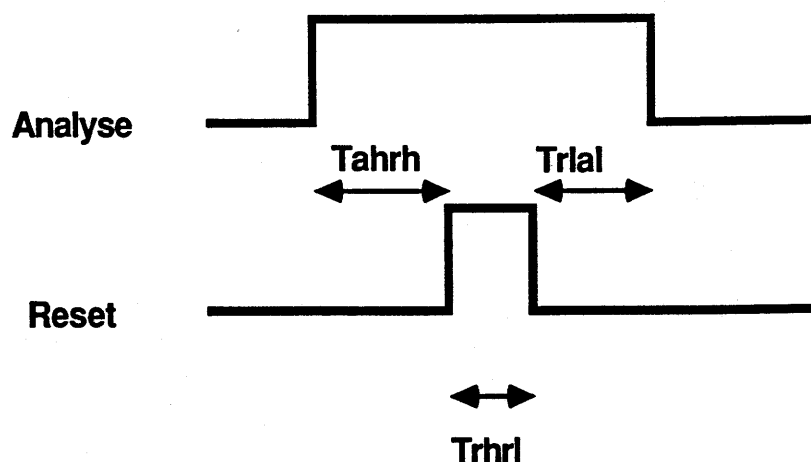
The transputer will boot from any link.
A small code packet (< 256 bytes) is loaded and run.
This *preamble* should load a loader program.

BOOTING FROM A LINK

The transputer polls all the links awaiting a message.
Any unused links should be tied low.
3 actions are possible decided by the first byte.

- Byte is 2 through 255, booting.
Followed by that number of bytes of **boot message**.
This is loaded at **MEMSTART** and executed.
- Byte is 0, writing data.
This is a **Poke**, 0 A A A A D D D D to transputer.
LS byte first, transputer resumes polling.
- Byte is 1, reading data.
This is a **Peek**, 1 A A A A to, D D D D from.
LS byte first, transputer resumes polling.

ANALYSING A TRANSPUTER



Tahrh 3 milliseconds, software and data dependent
Trhl 8 periods of input clock
Trhl 1 nanosecond

RESET AND ANALYSE 1

Analyse high to reset high

A graceful ceasing of activity. *Potentret*

- Transputer continues to next reschedule point.
- Links continue,
but outputs no longer access memory.
- Memory interface continues to refresh.

Time is finite if

- 1/ All transputers analysed.
- 2/ No long high priority processes.
- 3/ Link adaptors become idle.

RESET AND ANALYSE 2

Reset and analyse taken low.

All communication and processing activity will cease.
Transputer reboots according to **BootFromROM**, BUT

- No EMI reset or configuration cycles.
- Refresh continues.
- **Areg** holds the **Ip**tr when halted
- **Breg** holds the workspace descriptor (address and priority bit)
- **Creg** holds the boot link ID if used.
- Previous state available using *testerr testthalttestpranal*

HARDWARE TESTING

or debugging prototype transputer boards.

Single transputer boards

- 1) The transputer
- 2) The memory
- 3) The peripherals
- 4) The links

Multiple transputer boards

- 1) Extensions from single processor boards.
- 2) Communications/ power supplies/ dynamic loads.
- 3) Interactions on shared peripherals

System problems

- 1) Power
- 2) Control
- 3) Communication

SINGLE TRANSPUTER BOARDS

The transputer

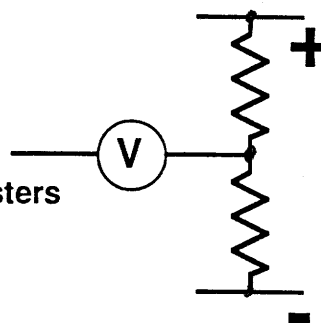
a complete "computer on a chip"

it requires

- ☐ 5 MHz clock.
- ☐ Power.
- ☐ Held low BootFromRom, Analyse and HoldToGrounds.
- ☐ A link with speed set and connected, others idle.
- ☐ Reset taken from high to low.

TESTING THE TRANSPUTER SOCKET

- O Power up without transputer
- O Check DC characteristics of socket
This can be done with a DVM and two resistors
- O LinkIns should show a pull down resistor
Pins to Vcc, Gnd or open circuit except
ClockIn and TTL driven pins
- O Check ClockIn with a scope
- O Check boot link by wiring back on itself



LJP/VAX/tester.00

SOCKETING THE TRANSPUTER

After socket is DC tested, insert transputer with power off.

Load test program to prove transputer and connections.

Add further software to test memory, peripherals and links.

Testing the memory

- 1) Use PEEK and POKE from TDS
Slow testing with 18 bytes/word comms for a word test.
Can report on hanging,
eg "Failure to test location #xxxxxxxx"
- 2) Download test program into on-chip RAM.
Faster testing once processor chip is proven

MEMORY CONFIGURATION

Check that the chosen memory configuration is implemented.

Use TDS to do continuous PEEK and POKE.

Check with a scope for gross memory timing errors.

Once gross functionality proven check memory.

- report errors to test address lines/chips
- messages should identify specific lines/chips
- report errors selectively
a 256kx1 chip fail causes a lot of output

PERIPHERALS

Testing will possibly exceed read/write capability

Simplest case

Write code to access device repeatedly and use a scope.

More complex or non-repetitive systems

Possibly use a logic analyser to capture test effects

Bidirectional peripherals

eg UARTs

Connect input to output and repetitively test.

MULTIPLE TRANSPUTER BOARDS

A simple extension from a single transputer board.

Each processor can be tested individually by connecting the TDS to each processor in turn.

Get every tested transputer to test the next by

- 1 **daisy chaining** methods
- 2 **worm** methods

LINKS

- The boot link is already proven.
- Test others by connecting links together, eg B004 test disk.
- For beta release occam 2, use sacrificial process
For future occam see Tech note 1,
"Extraordinary use of transputer links"
- Test for varying data structures, eg
BYTES, INT16s, INT32s,
slice communication in one direction,
bi-direction slicing to test for crosstalk.

MULTIPLE TRANSPUTER COMMUNICATIONS

Prove the transputers *then* exercise all links.

Exercising all interconnecting links will test:

- 1 **The links**
interconnections, buffering, crosstalk
- 2 **The power supply**
An idle transputer draws the least current.
8 simultaneous block moves draws the most current.
Dynamically exercise the PSU by switching between extremes.

INTERACTIONS VIA PERIPHERALS

The transputer concept excludes the sharing of resource
except for

- 1 **High integrity systems**
Control of a peripheral may be important enough to allow *any* transputer to interface to it.
- 2 **Peripherals interacting**
Operations on a peripheral of one transputer affect a peripheral connected to a different transputer.

Test each system in isolation with the other idle.

Empirical methods can be used as a safeguard.

SYSTEM TESTING

Proven boards *do not* constitute a proven system.

Factors affecting the system as a whole are:

- 1) **Power supply**
particularly dynamic characteristics
- 2) **Board control signals**
reset and analyse
- 3) **Communications**
links and peripheral I/O
- 4) **Electromagnetic environment**

ELECTROMAGNETIC ENVIRONMENT

In assembled systems noise can be:

- ☐ Conducted from outside
- ☐ Magnetically induced
- ☐ Received through an external wire as an arial

To help reduce noise effects:

- ☐ Put all wires into a common earth sheath where possible
- ☐ Tie chassis strongly to 0v
- ☐ Bind separate chassis with earth braids
- ☐ Use optical fibres

PROGRAM DESIGN FOR CONCURRENT SYSTEMS

based on

technical note 5

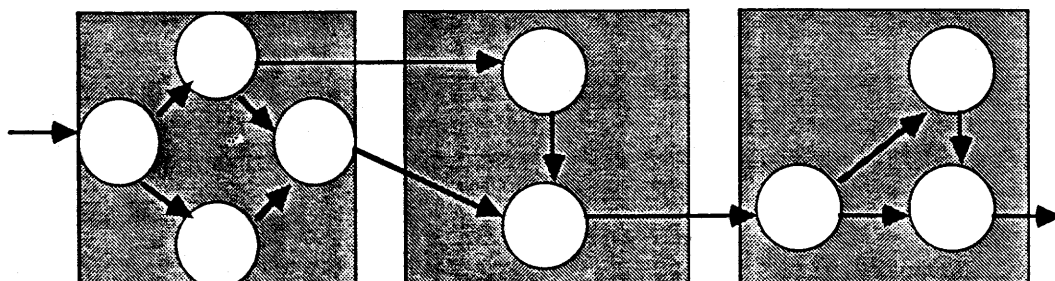
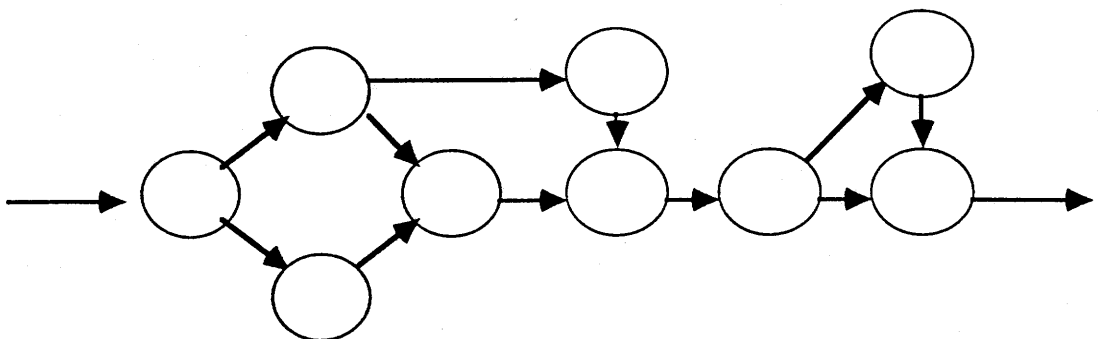
by Philip Mattos

OVERALL SYSTEM DESIGN

Balance between :

- 1 **performance**
Critical time constraints in real time systems
Bettering competitive systems on the market
- 2 **programming simplicity**
Modular high level design provides an upgrade path
Flexibility in hardware solutions
- 3 **design constraints**
Cost may prohibit large numbers of components
System size, mass or power requirements

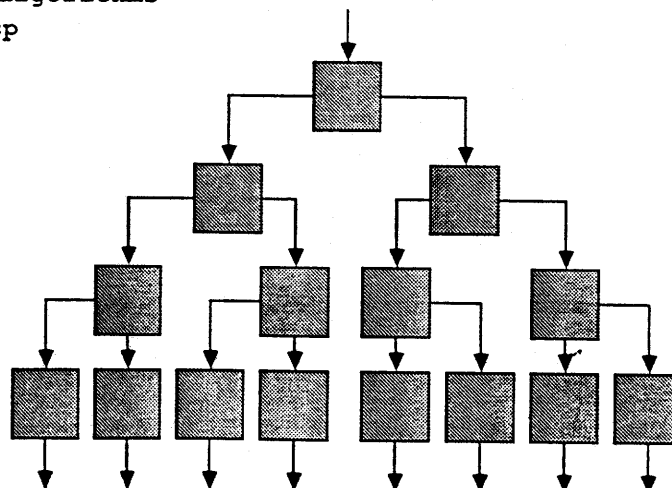
PROCESSES AND PROCESSORS



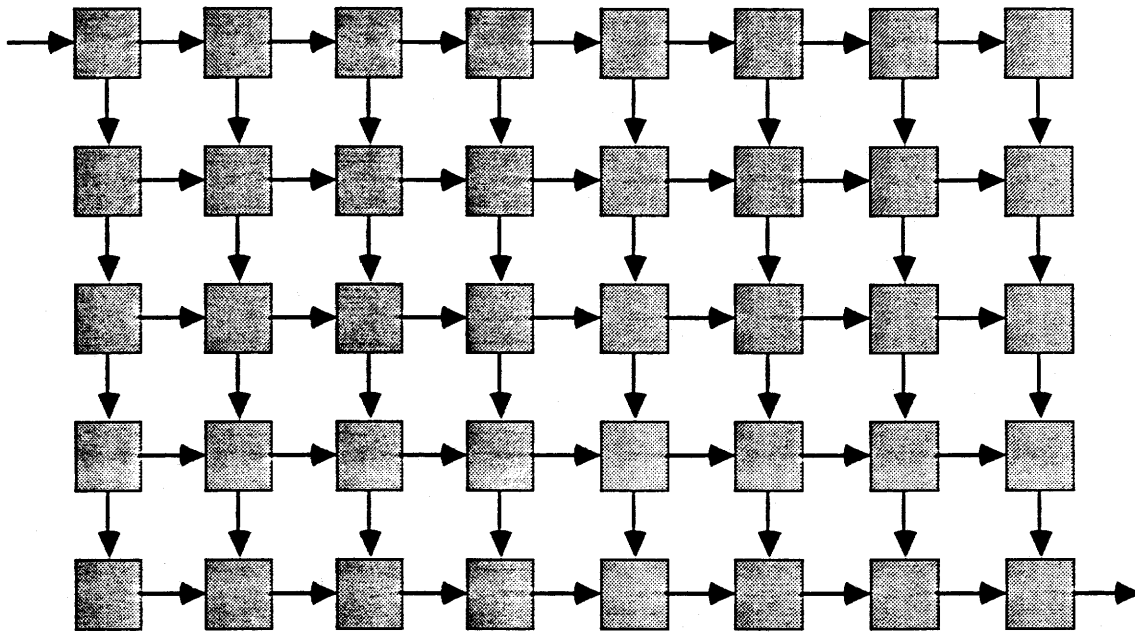
Functional	--	application derived connectivity
Pipeline	--	simple work distribution
Spaceline	--	expand by nets for more throughput
Trees	--	a form of spaceline suiting certain tasks
Arrays	--	spreads searches etc in 2 dimensions
Complex arrays	--	hypercubes etc, see appnote 13

TREE STRUCTURES USING THREE LINKS

- ```
-- Decision theory
-- Game playing algorithms
-- Prolog and lisp
```

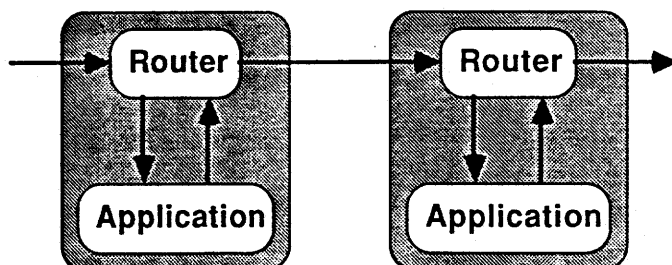
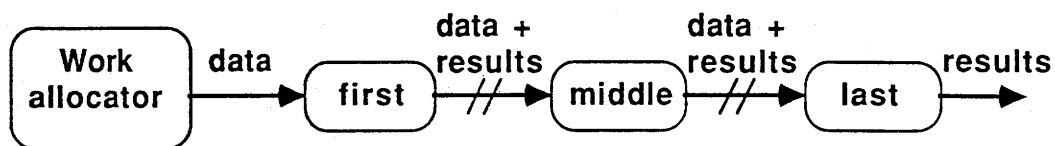


## CONNECTING TRANSPUTERS IN ARRAYS



LJP/MAC prgdsgr1/06

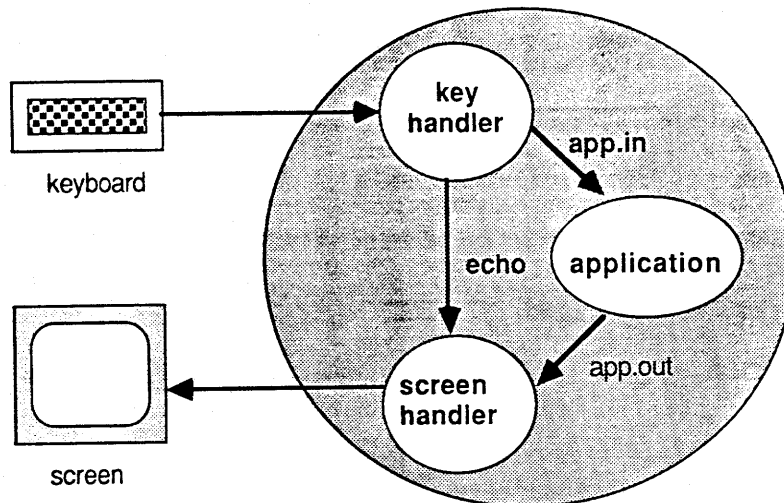
## DISTRIBUTED COMPUTATION



LJP/MAC prgdsgr1/02



## FUNCTIONAL DIVISION



CHAN OF INT app.in, app.out, echo:  
 PAR  
   keyboard.handler (keyboard, app.in, echo)  
   screen.handler (screen, app.out, echo)  
   application (app.in, app.out)

LJP/MAC/prgdsgn1/00

## USE OF FOLDS TO DEFINE STRUCTURE

```

PROC my.proc (parameters)

 ... declarations, constants, variables and PROCs
 SEQ
 ... initialisation
 ... WHILE condition
 SEQ
 ... loop initialise
 ... input data
 ... act upon it
 ... tidy up this pass
 ... tidy up process
 :
```

## EXAMPLE OF SIMPLE KEYBOARD HANDLER

```
PROC keyboard.handler (CHAN OF INT in, out, to.screen)
 INT ch:
 VAL stopch IS INT '@':
 BOOL running:
 SEQ
 running := TRUE
 WHILE running
 SEQ
 in ? ch
 PAR
 out ! ch
 to.screen ! ch
 IF
 ch = stopch
 running := FALSE
 TRUE
 SKIP
 :
```

## USING ALT FOR SELECTING CHANNEL INPUT

```
WHILE condition
 ALT
 ... input from echo
 SEQ
 ... go to echo cursor position
 ... output text
 ... update cursor position
 ... input from application
 SEQ
 ... go to application cursor position
 ... output text
 ... update cursor position
```

## CONFIGURING FOR THREE TRANSPUTERS

```

... SC keyboard.handler
... SC screen.handler
... SC application
CHAN OF INT keyboard, screen, echo, app.in, app.out:
PLACED PAR ANY
LJ PROCESSOR 0 T4
 PLACE keyboard AT Link0in:
 PLACE echo u AT link1out:
 PLACE app.out AT Link2out:
 keyboard.handler (keyboard, app.in, echo)
PROCESSOR 1 T4
 PLACE screen AT Link0out:
 PLACE echo AT Link1out:
 PLACE app.out AT Link2in:
 screen.handler (screen, app.out, echo)
PROCESSOR 2 T4
 PLACE app.in AT Link0in:
 PLACE app.out AT Link1out:
 application (app.in, app.out)

```

## COMBINING SC PROCS FOR CONFIGURATION

```

{{{ SC keys.and.screen
{{{F keyscr.tsr
PROC keys.and.screen (CHAN key.in,
 scr.out, key.out, scr.in)

... SC keyboard.handler
... SC screen.handler

CHAN OF INT echo:

PAR
 keyboard.handler (key.in, key.out, echo)
 screen.handler (scr.out, scr.in, echo)
:
}}
}}
```

## CONFIGURATION FOR TWO TRANSPUTERS

```
... SC keys.and.screen
... SC application
```

CHAN OF INT keyboard, screen, app.in, app.out:

PLACED PAR

PROCESSOR 0 T4

```
PLACE keyboard AT Link0in:
PLACE screen AT Link0out:
PLACE app.in AT link1out:
PLACE app.out AT Link2in:
keys.and.screen (keyboard, screen, app.in, app.out)
```

PROCESSOR 1 T4

```
PLACE app.in AT Link0in:
PLACE app.out AT Link1out:
application (app.in, app.out)
```

## FACTORS AFFECTING CONFIGURATION OPTIONS

- 1 The **number** of transputers available
- 2 The **connectivity**  
Each transputer has only four links
- 3 The processor **loading** cannot be ideal  
The system will run as fast as the most loaded node
- 4 The local **memory** on each node  
Internal and external RAM speed differences  
Particular processes may need large workspaces

# **HARDWARE DESIGN USING THE TRANSPUTER**

## **Considerations**

Boards Format

- Imposed
- free choice
- considerations

Memory

Buffering

- to connectors
- to memory

Power supply

Communications

Design for rapid development

Design for production

Design for reliability

## **BOARD FORMAT**

-Assuming the system design stipulates the amount of memory on each transputer

- ... but the hardware designer must be consulted
- ... often the system could make better use of certain memories
- ... so iterate

-Imposed Format

Avoid making memory interface across board edge  
Investigate can requirements be met on one board  
Different memory chip formats x1, 4 x8 - SILs, ZIPs  
can several nodes share the board

-Free Format

same considerations  
match board choice to node size

-Layers and Tracking

## MEMORY

-Assume that by iteration, a memory size has been agreed between hardware designer and system designers.

### Choice of Format

Assuming 32 chips are necessary, evaluate \*1, \*4, \*8

Assuming 16 chips are necessary, evaluate \*4, \*8

Similarly as size reduces, \*8 becomes preferable

Availability of common I/O or separate I/O versions

Availability of OE pin

Minimising number of banks simplifies address decode but increases power requirements

Refresh requirements

## Physical Layout

Minimise track lengths

Is buffering

1. required [capacitance > 50 pf inc. track]
2. advisable [<50 pf, skew tolerances]
3. not needed

Damping/matching resistors

Through-tracking

Clever tricks - cf B004 driver positioning

Track separation - cross talk

## Memory Timing Design

The TDS provides a program to investigate possible configurations.  
See Technical Note 9

Investigate suitability of internal configurations.

If not suitable - tailor an external one.

Is ROM/EPROM required : if so, external configuration is almost free,  
otherwise PAL required.

Consider other transputer speeds : often a slower  
transputer will run as fast in big memory systems

e.g. 3 cycles of 67 ns = 200 ns

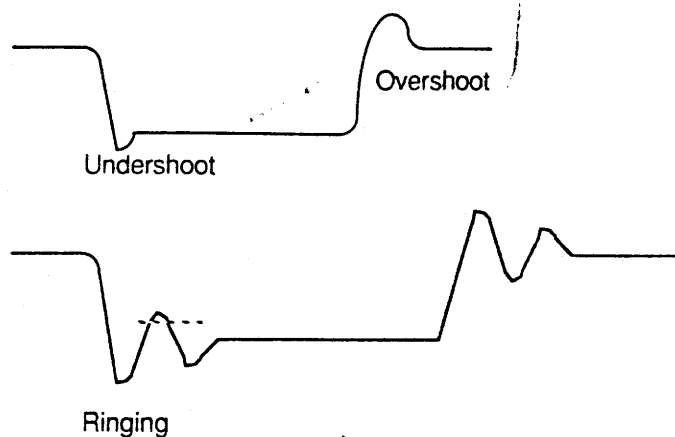
So a -20/4 cycle or -15/3 cycle will give similar performance in EXTERNAL  
memory.

## Memory

### Signal Integrity

1. Undershoot, Overshoot, Ringing
2. Damping
3. Edge Speeds
4. Timing Skew

## Undershoot & Ringing



Permitted levels are  $<0.5v$  outside rails  
Misoperation may occur if ringing causes level to cross threshold.

## Memory

### Damping

- reduces overshoot/undershoot/ringing
- the more rapid the edge transition, the worse the problem
- the longer the track the worse the problem

If reflections return after the edge has passed halfway point, problems

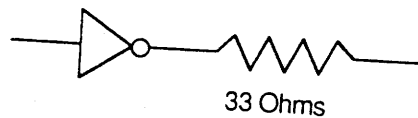
Fairchild FAST., Texas AS, more recently FACT,  
have edge speeds down to 1ns.

Edges should be slowed to around 5ns, and repeat  
reflections prevented by using a series damping resistor.

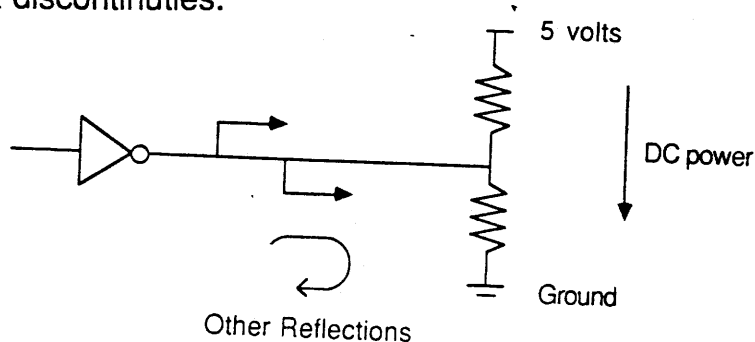
Also serves to match line.

Values 22 ohms to 82 ohms by experiment.

## Damping



Parallel Termination possible, but uses DC current, and may not prevent reflections from intermediate track discontinuities.



## Memory

### Timing Skew

Memory interface program (production model) allows for transputer induced skew, but not TTL, or damping resistors, load capacitors.

Beta release does not allow for even transputer skew  
- watch particularly read-data set-up.

Dynamic RAMs are very sensitive to addresses not set up in time

- can cause loss of complete 2 rows of data

Ensure time is allowed for transition time

## Buffering

### Off Board

In general, all signals crossing board boundary should be buffered, but there are exceptions.

1. Drive capability
2. Static
3. Reflections etc from "unknown" beyond connector
4. Reduce the cost of a spanner in the works  
- a buffer is cheaper than a transputer.

## Buffering

Offboard continued

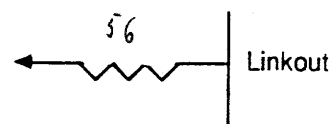
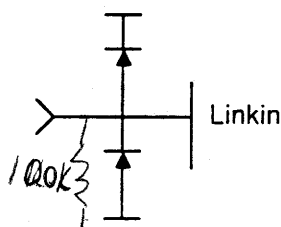
### Exceptions

Transputer Links

known to be point to point

skew very important

must be static protected/matched



Buffering - on board to memory.

Why

- drive capability
- to mask turn on/off incompatibility
- to speed up edge times with capacitive loads

Remember - in most systems, the dominant load is capacitive, not DC.

## Problems

- 1 Introduces delay
- 2 Introduces skew (Variable delay)
- 3 Takes board space and power

## Power Supply

- 1 DC : the transputer takes less than a watt, so presents no problem.
- 2 Remember it has several Vcc, GND pins.  
All must be connected and to the same  
Vcc, GND i.e. no transit current
- 3 AC
  1. The transputer is very fast
  2. It has a 32 bit bus, swinging through 5v CMOS levels.
  3. It has phase locked loops that need "protecting".

## Recommendations for Power

Use a dedicated power and ground plane  
(i.e. 4 layer board) to achieve a low inductance power path.

Ensure board is well decoupled :  
at least 50  $\mu$ f on a double Eurocard.  
Ensure HF covered.

3000  $\mu$ f

Ensure noise sources are well decoupled  
- 10 nF/100nF near each chip.

Ensure power feed cables from PSU to  
board are adequate i.e. Low Voltage drop  
(perhaps remote voltage sense)

Low Inductance (perhaps chassis decoupling capacitors)  
Ensure switching PSU's do not suffer from  
resonance on dynamic loads.

## Communications Transputer links

Four considerations

Drive

Skew

Delay

## Drive

2 mA/50pf

If over 50pf - i.e. other than short local connection, buffering required.

Point-to-point, so 2mA not a problem.

## Skew

The link spec allows a skew of

|         |      |
|---------|------|
| 5 Mband | 20nS |
| 10      | 10n  |
| 20      | 3nS  |

If buffering used, it must meet this.

## Links

### Delay

If links are not buffered, the delay from end to end must not exceed 0.4 bit-times, in order that reflections have returned before the next bit is sent.

### Speed

Considerations 1-3 lead to a choice of link speed.

Remember

- 1 Link0 can run at special speed, Links 123 can, but links 123 must run the same speed.
- 2 A single transputer can operate only on 2 link speeds i.e. 5+10 or 10+20, not both.

## Design for Rapid Development

A transputer is a fully operational computer provided clock, Reset, Power, Analyse etc are correctly wired, and a test program can be loaded.

To allow easy testing of services  
- ensure test points are available,  
including ground points.

## Design for Rapids Development cont.

Loading requires:-

- 1 a free link
- 2 BOOT from ROM = 0
- 3 No other link active

Therefore ensure:

- 1 If all links are to be attached to link adaptors, or other transputers on the board, that at least one link has a jumper that can be removed for TDS connection.
- 2 If the system is designed to boot from ROM, a switch or jumper is provided in the BOOT from ROM line.
- 3 All other links are idle - achieved by resetting all transputers and providing a pull **down** resistor on each off-board or un-used link.

## Rapid Development

Once a program can be loaded, the memory interface and links can be tested, and hence anything connected to them.

For details on memory testing etc see "Hardware Testing"

## Design for Production

As with any design - how often does the prototype work, but not the first production boards - if anything, the high speed characteristics of transputer boards mean if a hand wired version works, so will the PCB.

Remember - volume producers need space around the edge of a board, and possibly holes, to handle it.

- Auto insertion tools need space between chips and prefer a single orientation.
- The yield of a board full of transputers e.g. 42 x 84 plated holes will be terrible
- subdivide the problem.

The transputer allows very dense boards - but do not overdo it.

## Design for Reliability

The transputer is fast. It encourages the use of fast memory, often with common I/O.

There is a temptation to try and cut corners on tristate times.

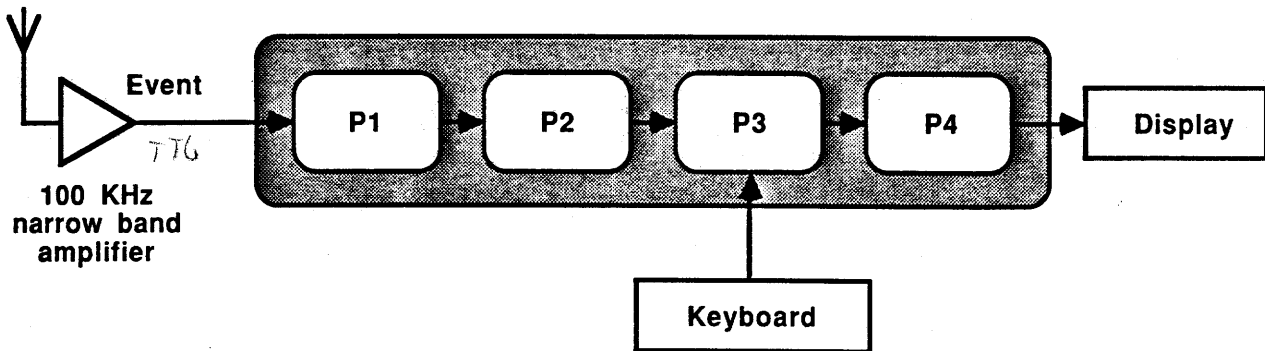
Ensure contention current is limited to very few mA/ns, or memories, buffers, transputers will suffer from metal migration [and extra noise is created].

The transputer encourages efficient use of board space.

It is specified 0-70 degrees Centigrade in still air.

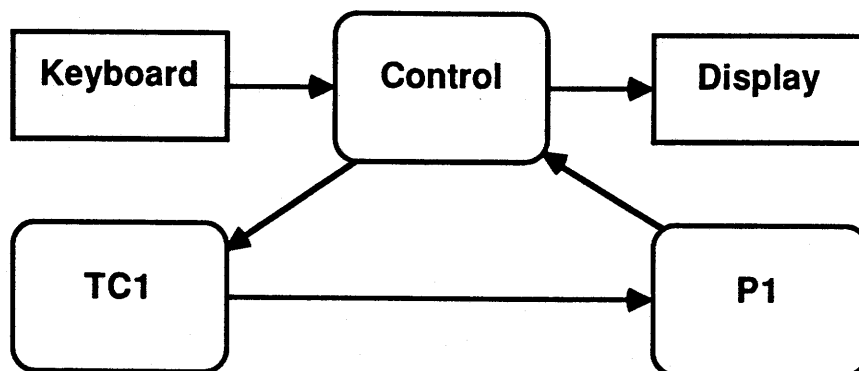
- 1 Most memories are specified at 400 ff/min <sup>air</sup> ~~over~~ flow.
- 2 Ensure that with that packing density, the heat from memories plus TTL does not take air temperature over 70 degrees centigrade - ideally 40 degrees centigrade (because I don't like wearing asbestos gloves).

## NAVIGATION SYSTEM STRUCTURE



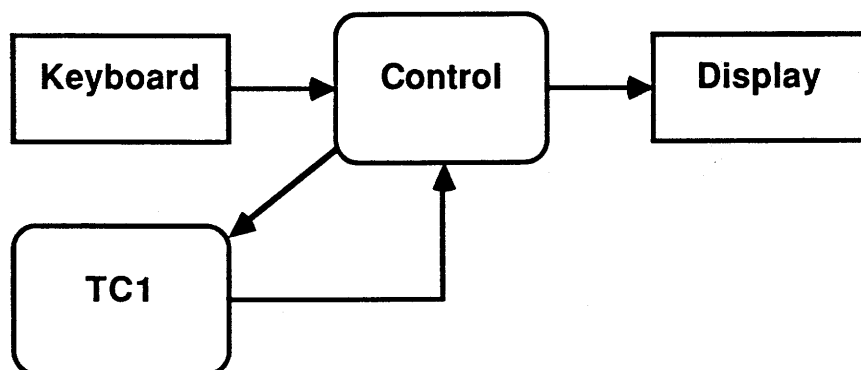
LJP/MAC/navfoils/00

## SOFTWARE TEST DATA GENERATOR



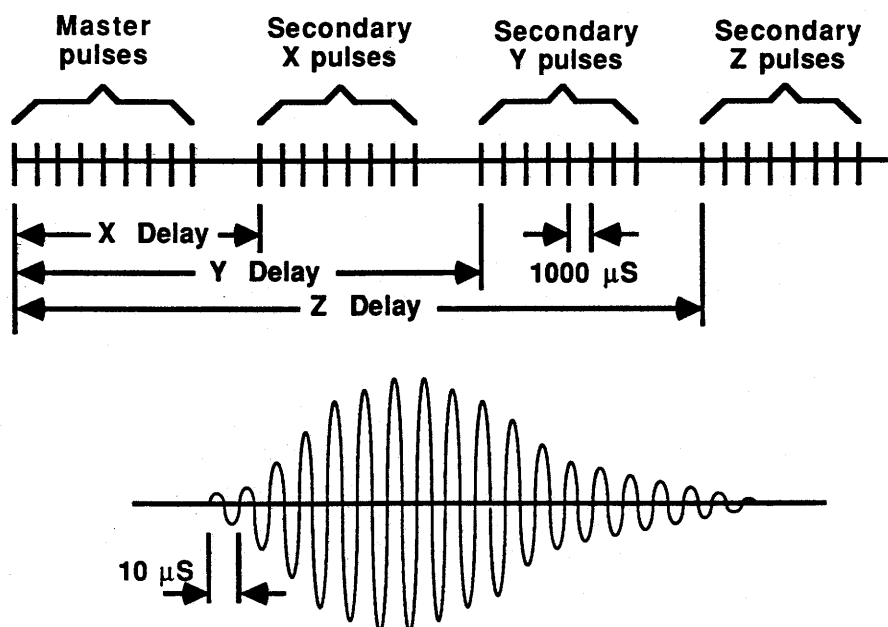
LJP/MAC/navfoils/01

## TESTING THE DATA GENERATOR



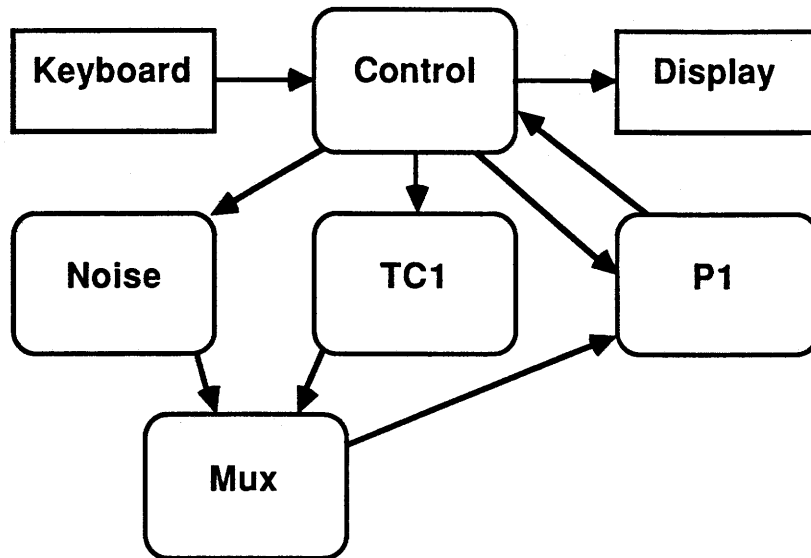
LJP/MAC/navfoils/02

## OUTPUT FROM DATA GENERATOR



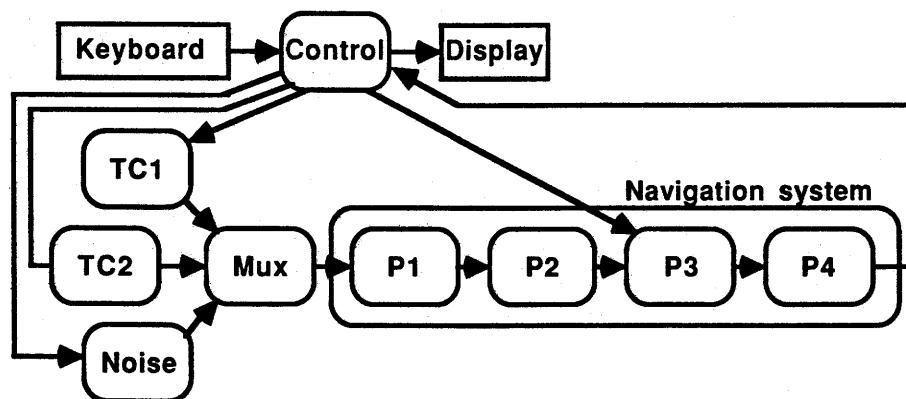
LJP/MAC/navfoils/03

## INCLUDING TEST DATA FOR NOISE



LJP/MAC/navfoils/04

## TESTING THE COMPLETE SYSTEM



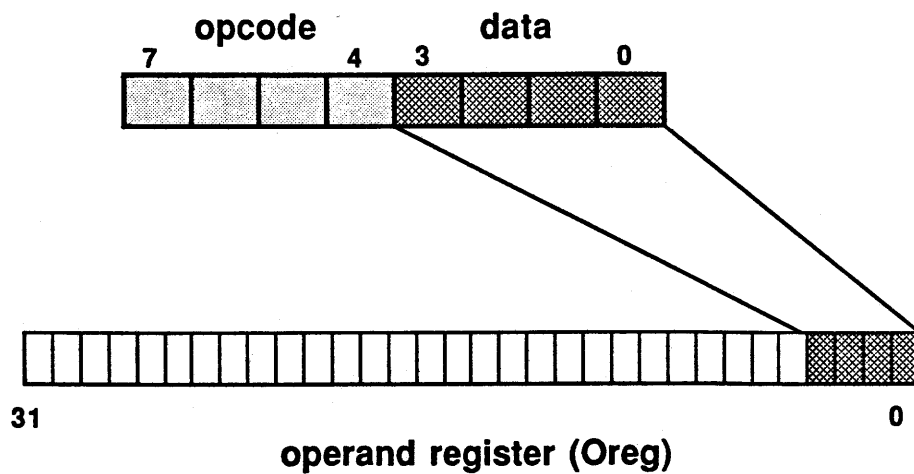
LJP/MAC/navfoils/05

# THE TRANSPUTER INSTRUCTION SET

## (T414 specific)

LJP/MAC/r4set

## BYTE SIZED INSTRUCTIONS

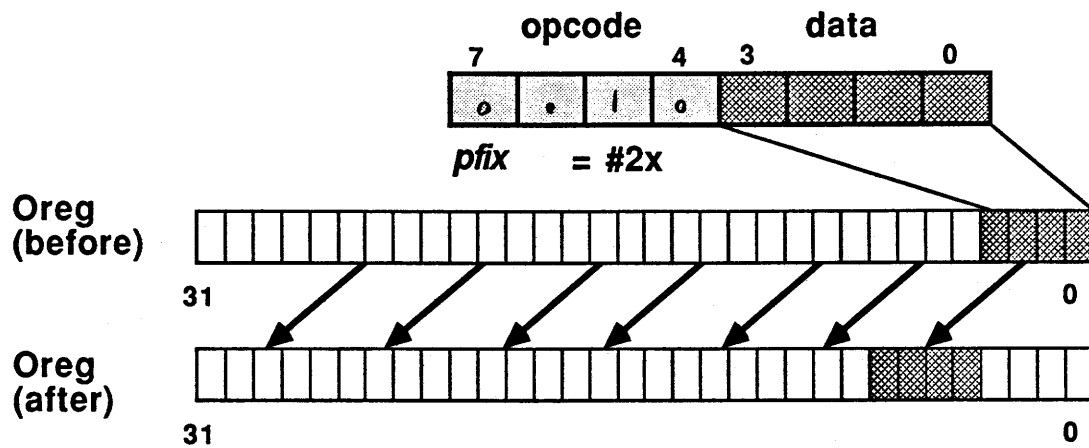


All operations except pfix and nfix end by setting Oreg to zero.

LJP/MAC/r4set

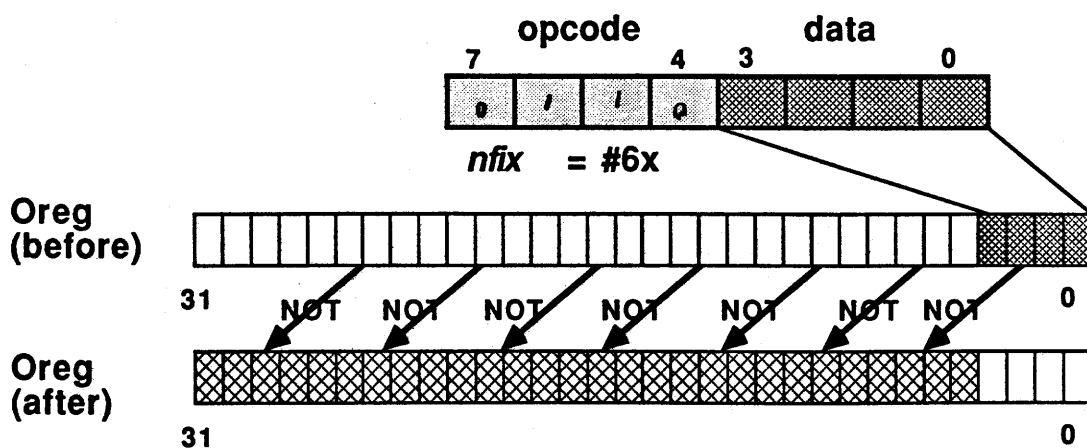
## PREFIX INSTRUCTIONS

- prefix**  
(1 cycle)
- Loads 4 bit data into Oreg
  - Shifts Oreg left 4 bits
  - Does not set Oreg to zero

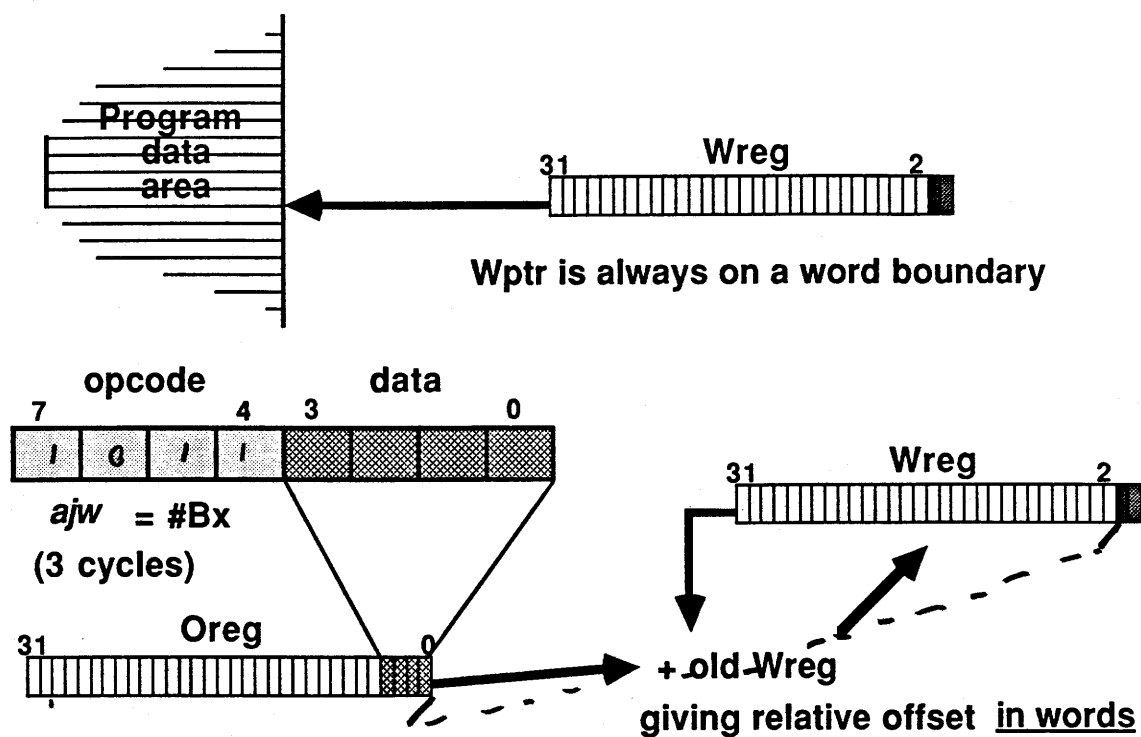


## NEGATIVE PREFIXING

- nfix**  
(1 cycle)
- Loads 4 bit data into Oreg
  - Inverts every bit in Oreg
  - Shifts Oreg left 4 bits
  - Does not set Oreg to zero



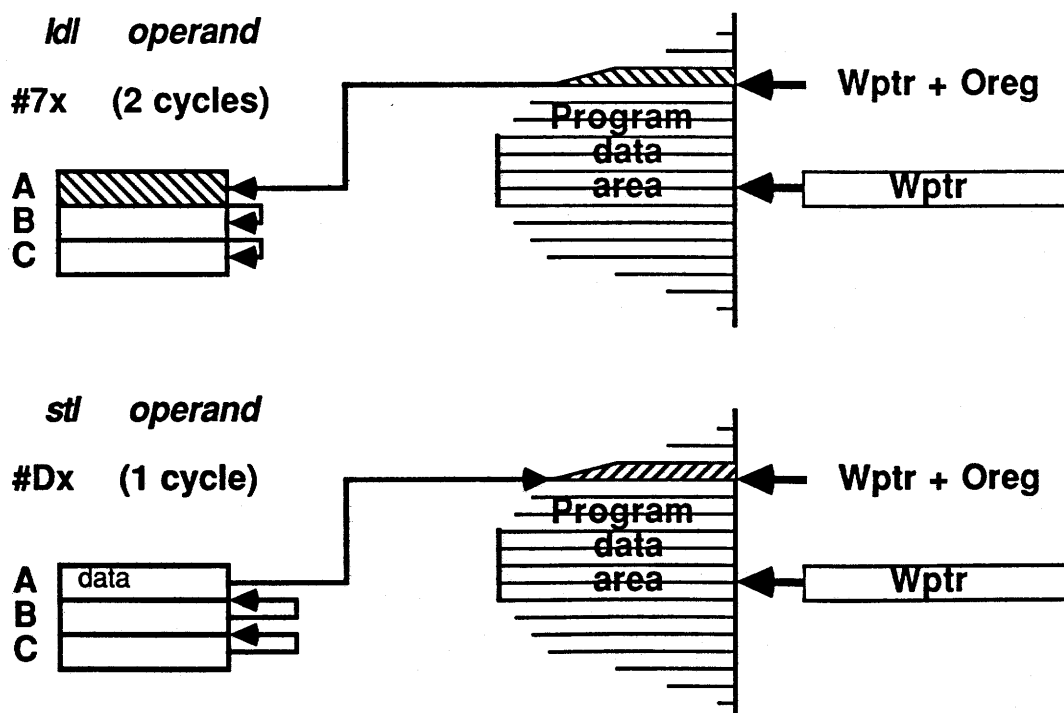
# THE WORKSPACE POINTER



LJP/MAG4oct

## LOCAL VARIABLES

Loads and stores are offsets from Wreg



LJP/MAG4oct

## EXPRESSION EVALUATION STACK

Instructions have no register field giving code compaction

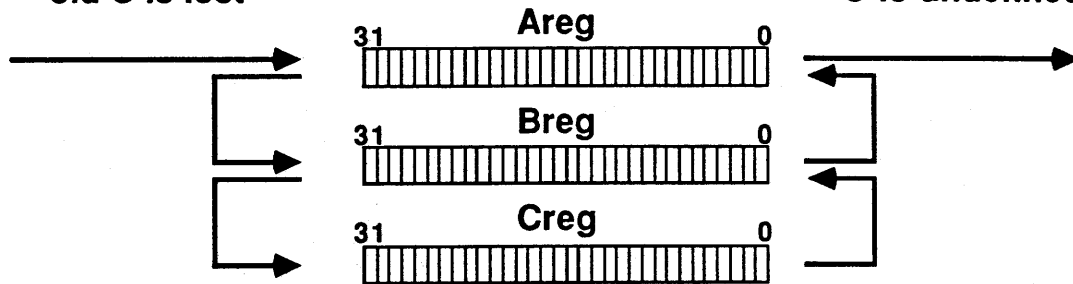
Loads, stores and arithmetic refer to Areg

### Loading into A

- pushes old A into B
- pushes old B into C
- old C is lost

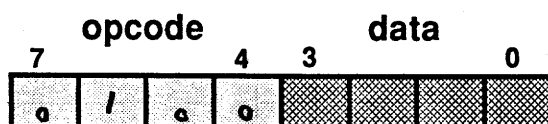
### Storing from A

- pops B into A
- pops C into B
- C is undefined

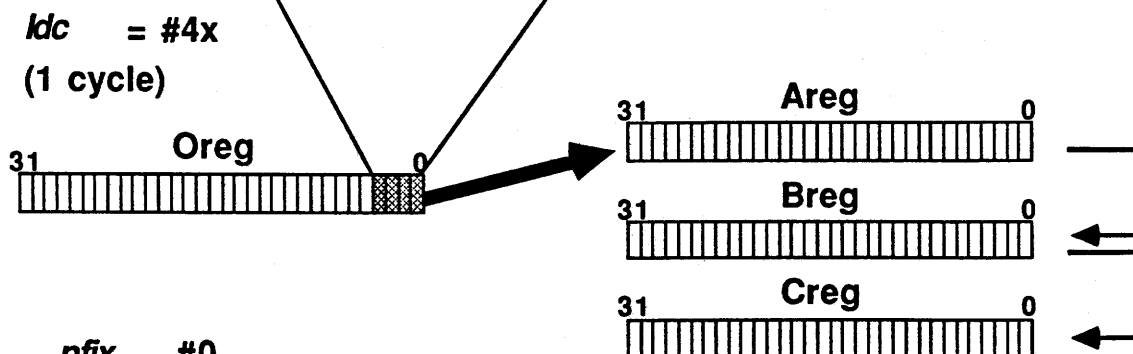


LIP/MAC/Mset

## LOADING CONSTANTS INTO AREG



Typical constants 0-15 loaded in a single byte



`nfix #0`

`ldc #E`

sets Areg = -2

Negative constants -1, -256 take two bytes

LIP/MAC/Mset

# EXPRESSION EVALUATION



VAL top IS 15, bottom IS 7:

INT a, b, c, d, e, f, g:

SEQ

$a := (b + (c \setminus 3)) * (d - ((f + (top - bottom)) / g))$

*ldl d ; ldl f ; adc 8*

|   |                      |
|---|----------------------|
| A | $f + (top - bottom)$ |
| B | $d$                  |
| C |                      |

*; ldl g ; div ; sub*

|   |                                  |
|---|----------------------------------|
| A | $d - ((f + (top - bottom)) / g)$ |
| B |                                  |
| C |                                  |

*; ldl c ; ldc 3 ; rem*

|   |                                  |
|---|----------------------------------|
| A | $c \setminus 3$                  |
| B | $d - ((f + (top - bottom)) / g)$ |
| C |                                  |

*; ldl b ; add*

|   |                                  |
|---|----------------------------------|
| A | $b + (c \setminus 3)$            |
| B | $d - ((f + (top - bottom)) / g)$ |
| C |                                  |

*; mul ; stl a*

LIPMAC84cel02

## THE OPERATE INSTRUCTION



Only 16 instructions can be represented by 1 byte

To extend the instruction set we use *opr*

*opr* uses the contents of Oreg as the opcode

|    |             |   |               |           |   |            |          |
|----|-------------|---|---------------|-----------|---|------------|----------|
| eg |             |   | <i>prefix</i> | <i>#3</i> | = | <i>#23</i> | 1 cycle  |
|    | <i>gajw</i> | = | <i>opr</i>    | <i>#C</i> | = | <i>#FC</i> | 2 cycles |

|                         |          |
|-------------------------|----------|
| Global adjust workspace |          |
| swaps Areg and Wreg     | 3 cycles |

LIPMAC4cel

## Arithmetic operators

The sequence *ldl X; ldl Y; op* evaluates *X op Y*

### Checked operators

| code  | mnemonic   | commutative | total cycles |
|-------|------------|-------------|--------------|
| #F5   | <i>add</i> | yes         | 1            |
| #FC   | <i>sub</i> |             | 1            |
| #25F3 | <i>mul</i> | yes         | 39           |
| #22FC | <i>div</i> |             | 40           |
| #21FF | <i>rem</i> |             | 38           |

### Unchecked operators (modulo arithmetic)

| code  | mnemonic    | commutative | total cycles              |
|-------|-------------|-------------|---------------------------|
| #F4   | <i>diff</i> |             | 1                         |
| #25F2 | <i>sum</i>  | yes         | 2                         |
| #F8   | <i>prod</i> | yes         | significant.bits.of.Y + 4 |

## COMPARISON AND CONDITIONALS

***eqc***  
#Cx (2 cycles)

Test equal to constant  
Compare Areg with constant in Oreg  
TRUE or FALSE left in Areg

***diff***  
#F4 (1 cycle)

Compute the difference between Breg and Areg  
The numerical result is left in Areg  
Do not check for overflow

***gt***  
#F9 (2 cycles)

Compare the values in Areg and Breg  
TRUE left if Breg greater than Areg  
Otherwise FALSE

***j***  
#0x (3 cycles)

Relative jump to lptr + Oreg

***cj***  
#Ax (2 or 4 cycles  
if jump taken)

Conditional relative jump to lptr + Oreg  
Jump taken if Areg is FALSE  
Areg popped if jump NOT taken

# OPTIMISING CONDITIONAL JUMPS

**#0x j operand** jump in 3 cycles (may deschedule)  
**#Ax cj operand** conditional jump (jump on FALSE)  
Areg = 0, jump not taken (2 cycles)  
Areg unaffected  
Areg <> 0, jump taken (2 cycles)  
Areg popped

## Conditional tests

|         |                                 |
|---------|---------------------------------|
| TRUE    | ldc 1                           |
| FALSE   | ldc 0                           |
| NOT X   | X ; eqc 0                       |
| X OR Y  | NOT (NOT X ; cj L ; NOT Y ; L:) |
| X AND Y | X ; cj L ; Y ; L:               |

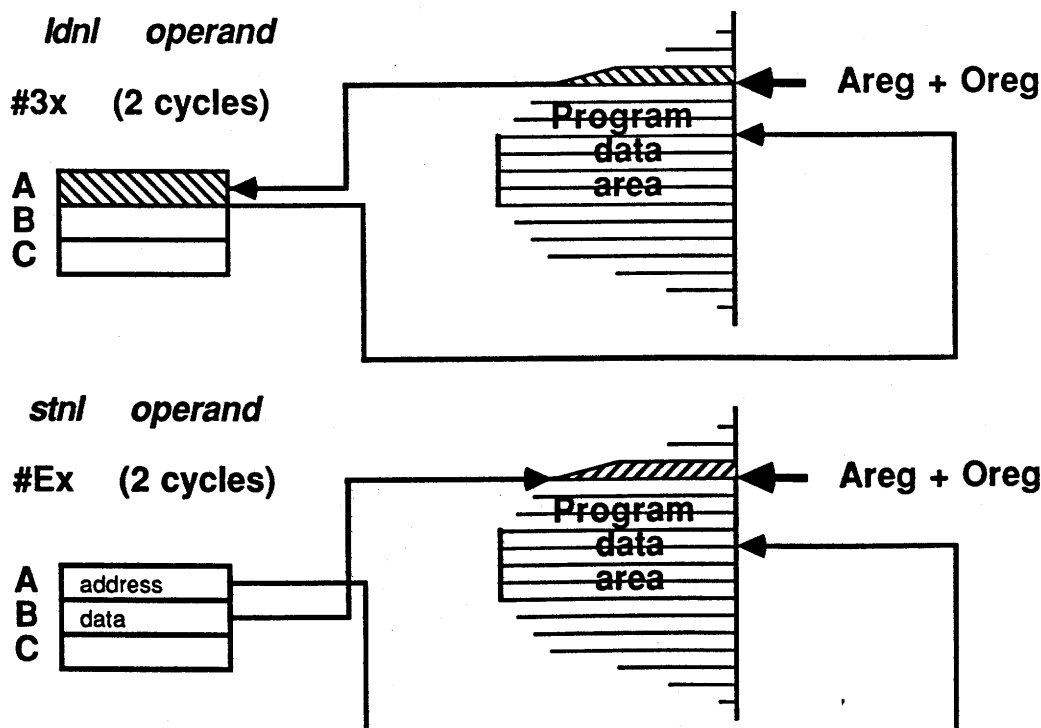
## Conditional jumps

|                  |                              |
|------------------|------------------------------|
| (X OR Y) ; cj L  | NOT X ; cj M ; Y ; cj L ; M: |
| (X AND Y) ; cj L | X ; cj L ; Y ; cj L          |

LJP/MAC/t4set03/02

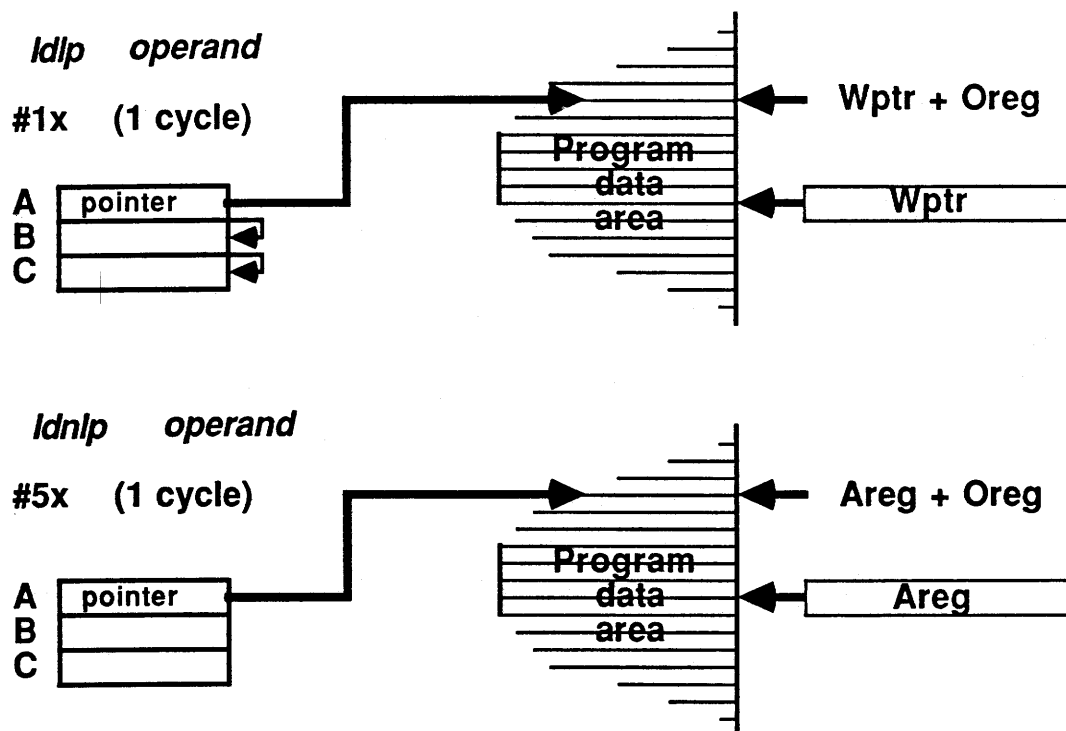
# NON - LOCAL VARIABLES

Loads and stores are offsets from Areg



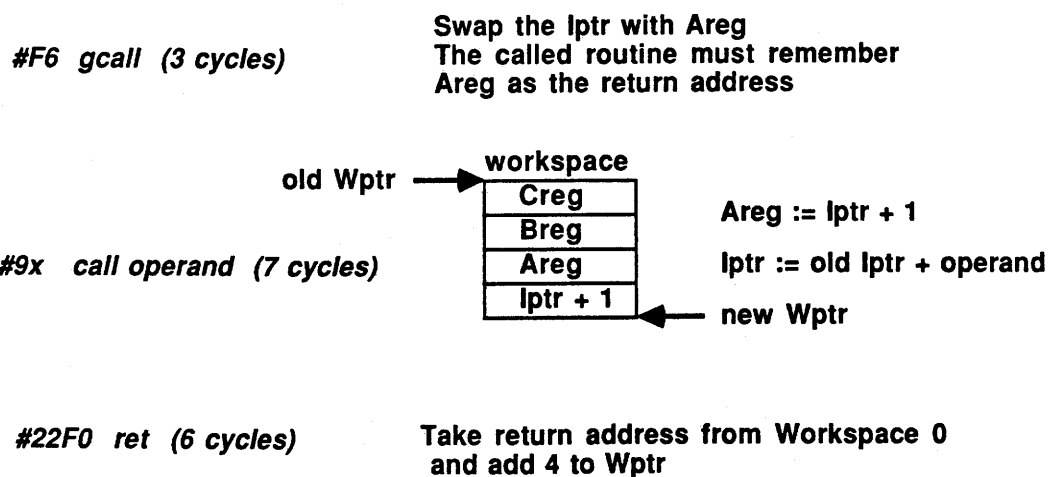
LJP/MAC/t4set

# POINTERS

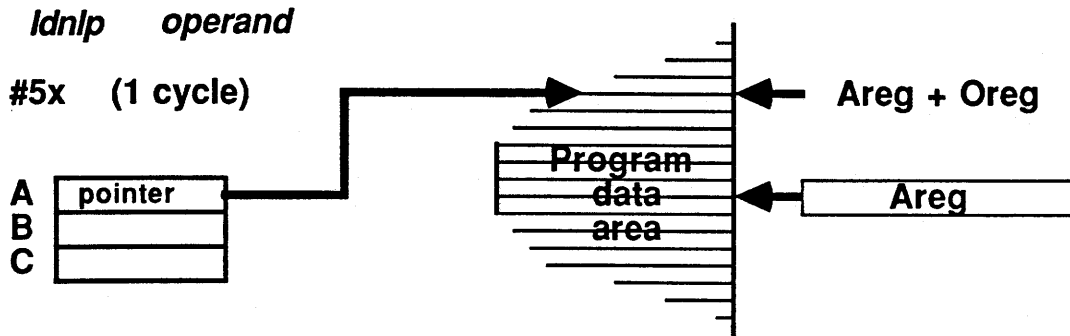
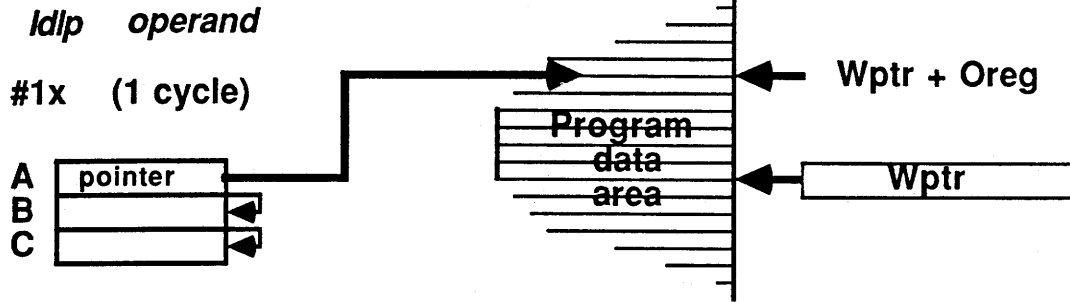


LJP/MAC/4set03

# PROCEDURE CALLS



# POINTERS

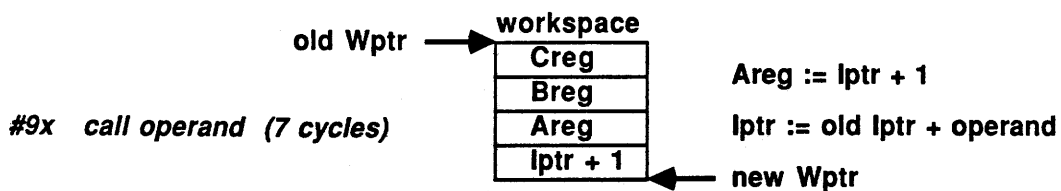


LJP/MAC/4set02

# PROCEDURE CALLS

#F6 gcall (3 cycles)

Swap the lptr with Areg  
The called routine must remember  
Areg as the return address



#22F0 ret (6 cycles)

Take return address from Workspace 0  
and add 4 to Wptr

# TREATMENT OF PARALLEL PROCESSES

#FD startp

Schedule process with workspace Areg and relative  
lptr in Breg on current priority process queue

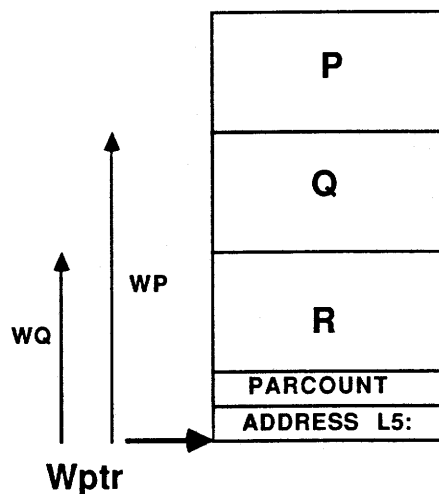
#F3 endp

Areg points to Wptr on diagram

If PARCOUNT = 1 continue from L5:

IF PARCOUNT <> 1 run next process on queue

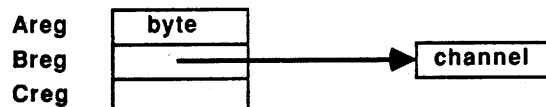
PAR  
P  
Q  
R



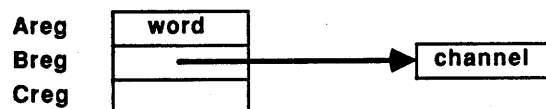
```
ldc 3; stl 1;
ldc (L5-L6); ldlp;
L6: stl 0;
ldc (L1-L2); ldlp WP;
L2: startp;
ldc (L3-L4); ldlp WQ;
L4: startp;
R; ldlp 0; endp;
L1: P; ldlp -WP; endp;
L3: Q; ldlp -WQ; endp;
L5:
```

# COMMUNICATION

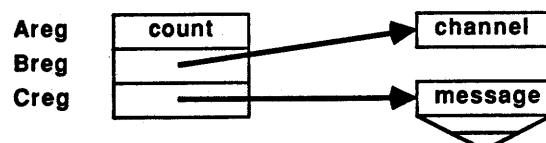
#FE outbyte



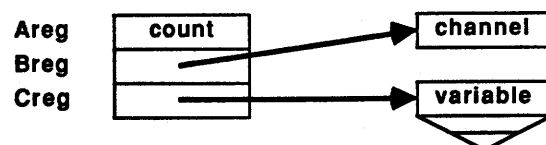
#FF outword



#FB out



~~#F7 in~~  
#F7 in



These instructions use links if Breg contains link address

## Indexing and array operations

### Subscripting

For array subscripts Areg := the address of Breg[Areg]

| code | mnemonic    | total cycles |                |
|------|-------------|--------------|----------------|
| #F2  | <i>bsub</i> | 1            | byte subscript |
| #FA  | <i>wsub</i> | 2            | word subscript |

### Loading and storing bytes quantities.

| code  | mnemonic  | total cycles |            |
|-------|-----------|--------------|------------|
| #23FB | <i>sb</i> | 6            | store byte |
| #F1   | <i>lb</i> | 5            | load byte  |

## Bit manipulation operators

Bitwise operators that work on all 32 bits.

| code  | mnemonic   | total cycles |                     |
|-------|------------|--------------|---------------------|
| #24F6 | <i>and</i> | 2            |                     |
| #24FB | <i>or</i>  | 2            |                     |
| #23F3 | <i>xor</i> | 2            | exclusive or        |
| #23F2 | <i>not</i> | 2            | (affects Areg only) |

Bitwise shifts of the form *X shift n*

| code  | mnemonic   | total cycles |             |
|-------|------------|--------------|-------------|
| #24F1 | <i>shl</i> | $n + 3$      | shift left  |
| #24F0 | <i>shr</i> | $n + 3$      | shift right |

All vacated bit positions are filled with zeros.

## Other useful opcodes

| code | mnemonic | total cycles |
|------|----------|--------------|
|------|----------|--------------|

|       |                |        |                             |
|-------|----------------|--------|-----------------------------|
| #24FA | <i>move</i>    | 2w + 9 | move message                |
| #21F2 | <i>resetch</i> | 4      | reset channel               |
| #21FB | <i>ldpi</i>    | 2      | load pointer to instruction |
| #21FE | <i>ldpri</i>   | 2      | load current priority       |

### error flag

|       |                |                |                      |
|-------|----------------|----------------|----------------------|
| #22F9 | <i>testerr</i> | 3 (4 on error) | test error and clear |
| #21F0 | <i>seterr</i>  | 2              | set error            |
| #25F5 | <i>stoperr</i> | 3              | stop on error        |

### process queues

|       |              |   |                      |
|-------|--------------|---|----------------------|
| #23FD | <i>savel</i> | 5 | save fptr1 and bptr1 |
| #23FE | <i>saveh</i> | 5 | save fptr0 and bptr0 |
| #21F8 | <i>sthf</i>  | 2 | store fptr0          |
| #25F0 | <i>sthb</i>  | 2 | store bptr0          |
| #21FC | <i>stlf</i>  | 2 | store fptr1          |
| #21F7 | <i>stlb</i>  | 2 | store bptr1          |

## THE PREAMBLE

A byte array of transputer code.

Its functions are:

- 1 To be the first packet of code a transputer receives of less than 256 bytes in length.  
This has to initialise the transputer for occam.
- 2 To load a larger loader program, written in occam.
- 3 To complete the loading operation when the loader terminates so that the loader can be overwritten, if necessary.

used in PROC insert.preamble in EXE make.loader in \tds2soft  
A simpler version is available for the TDS server to boot the B004.

## THE LOADER REQUIREMENTS

The loader was shaped by the initial design goals.

- 1 Any code to be sent to the network (SCs) should only be transmitted by the host once, even if it is to be loaded at different addresses on different processors.
- 2 Blocks of code may be loaded in any order to any location on any processor.
- 3 The loader should not permanently occupy space in the target processors memory.
- 4 The loading strategy should be reasonably efficient for the number of transputers likely to be used with the TDS, say 500.
- 5 The loading scheme must be available yesterday.
- 6 All currently available transputers must be supported.

## THE PREAMBLE WORKSPACE LAYOUT

|                    |        |                         |
|--------------------|--------|-------------------------|
| VAL MemStart       | IS 12: | -- start of boot part 2 |
| VAL entry.address  | IS 11: | -- used for entry point |
| VAL entry.point    | IS 10: | -- loader parameter 5   |
| VAL buffer.start   | IS 9:  | -- loader parameter 4   |
| VAL memory         | IS 8:  | -- loader parameter 3   |
| VAL bootlink.param | IS 7:  | -- loader parameter 2   |
| VAL NotProcess     | IS 6:  | -- loader parameter 1   |

```
VAL preamble.words IS return.address:
 -- workspace common to loader and preamble
VAL return.address IS 5: -- <--- loader wptr
VAL next.wptr IS 4:
VAL bootlink IS 3:
VAL next.address IS 2, count IS 2:
VAL load.length IS 1, base IS 1:
VAL load.start IS 0: -- <--- preamble wptr
```

## THE PREAMBLE CODE 1

*initial.adjustment = (loader.workspace + 4) - preamble.words*

-- set up workspace save registers

L0:

*ajw initial.adjustment* -- grab space to work in  
*call 0* -- save A,B,C registers

-- save MemStart

*ldc L0-L5*  
*ldpi* -- get start address

L5:

*stl MemStart* -- save in local data

-- save NotProcess

*mint* -- the most negative integer  
*stl NotProcess* -- cannot be a process

## THE PREAMBLE CODE 2

-- Initialise process queues

*ldl NotProcess*  
*sthf* -- high priority queue

*ldl NotProcess*  
*stlf* -- low priority queue

*clrhalterr* -- clear halt on error  
-- or *sethalterr* if required

*testerr* -- clear error flag

## THE PREAMBLE CODE 3

```
-- Initialise link and Event words
-- SEQ base = 0 FOR 11
ldc 0 -- index starts from 0
stl base -- index of words, a variable
ldc 11 -- number to initialise
stl count -- count of words left
```

**L1:**

```
-- memory[base] := NotProcess
ldl NotProcess -- the value to save
ldl base -- index to word
ldl NotProcess -- offset of word array
wsub -- address of word
stnl 0 -- initialise word

ldlp base -- loop control area
ldc L2-L1 -- jump size
lend -- loop or continue
```

**L2:**

## THE PREAMBLE CODE 4

```
-- Set up some loader addresses
-- loader([8]CHAN OF ANY links, CHAN OF ANY bootlink,
-- [4]BYTE memory, VAL INT buffer, INT entry.point)
ldlp entry.address -- address of entry word
stl entry.point -- store in param 5

ldlp MemStart -- address of word offset
ldl NotProcess -- bottom of memory
diff -- offset of free word
stl buffer.start -- store in param 4

ldl NotProcess -- pointer to memory
stl memory -- param 3

ldl bootlink -- copy bootlink for loader use
stl bootlink.param -- passed as parameter 2
```

## THE PREAMBLE CODE 5

- Load bootstrap part 2
- Must be two bytes shorter than part 1 to fit the space.
- The two bytes prevent overwriting the call instruction.

```
ldlp load.length -- packet size location
ldl bootlink -- address of link
ldc 1 -- bytes to load
in -- input length byte

ldl MemStart -- start of load area
ldl bootlink -- address of link
ldl load.length -- message length
in -- input code block

ldl MemStart -- start of loaded code
gcall -- enter bootstrap part 2
```

## THE PREAMBLE CODE 6

The second part of the preamble.

- set up loader addresses

```
ldc packet.length -- buffer size of 60
ldlp MemStart -- buffer address
bsub -- address of loader code area
stl load.start -- start of next block to load

ldlp preamble.words -- loader's workspace 0
stl next.wptr -- wptr of loaded code
```

L7:

```
ldl load.start
stl next.address -- initialise next address
```

## THE PREAMBLE CODE 7

**L3:** -- loading the network loader program

|             |                     |                                          |
|-------------|---------------------|------------------------------------------|
| <i>ldlp</i> | <i>load.length</i>  | -- set to zero by <i>lend</i> previously |
| <i>ldl</i>  | <i>bootlink</i>     | -- link address                          |
| <i>ldc</i>  | <i>1</i>            | -- one byte only                         |
| <i>in</i>   |                     | -- byte length is in 32 bit word         |
|             |                     |                                          |
| <i>ldl</i>  | <i>load.length</i>  | -- test length = 0                       |
| <i>cj</i>   | <b>L4</b>           | -- jump on zero                          |
| <i>ldl</i>  | <i>next.address</i> | -- address of next block                 |
| <i>ldl</i>  | <i>bootlink</i>     | -- link address                          |
| <i>ldl</i>  | <i>load.length</i>  | -- bytes to receive                      |
| <i>in</i>   |                     | -- read in next block                    |
|             |                     |                                          |
| <i>ldl</i>  | <i>load.length</i>  | -- bytes received                        |
| <i>ldl</i>  | <i>next.address</i> | -- old block address                     |
| <i>bsub</i> |                     | -- new block address                     |
| <i>stl</i>  | <i>next.address</i> | -- new area to load                      |
| <i>j</i>    | <b>L3-L4</b>        | -- go back for next block                |

**L4:**

## THE PREAMBLE CODE 8

-- set up return address and call the loader

|              |                       |                             |
|--------------|-----------------------|-----------------------------|
| <i>ldc</i>   | <b>L6</b>             | -- offset to return address |
| <i>ldpi</i>  |                       | -- as absolute address      |
| <i>stl</i>   | <i>return.address</i> | -- saved                    |
|              |                       |                             |
| <i>ldl</i>   | <i>next.wptr</i>      | -- load new workspace       |
| <i>gajw</i>  |                       | -- set wptr, old in Areg    |
|              |                       |                             |
| <i>ldnl</i>  | <i>load.start</i>     | -- from old workspace       |
| <i>gcall</i> |                       | -- enter loader code        |

**L6:**

|                |                                |                     |
|----------------|--------------------------------|---------------------|
| <i>ajw</i>     | - (4 + <i>preamble.words</i> ) | -- set wptr         |
|                |                                |                     |
| <i>ldc</i>     | <i>0</i>                       | -- beginning from 0 |
| <i>sttimer</i> |                                | -- start the clock  |

## THE PREAMBLE CODE 9

```
-- set up parameters for last few blocks, if any

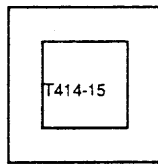
ldl bootlink.param -- new copy of link
ldl entry.address -- main body code entry offset
ldl NotProcess -- base of memory
bsub -- convert to address
ldc 0 -- reset load length byte
ajw 8 - initial.adjustment -- set wptr
-- reset workspace to start + 4 for call
-- this means that whilst the last few blocks are being
-- loaded the below workspace requirement overlaps these
-- last few instructions which are never used again.
call 0 -- write data into memory

ldl next.address -- loaded code wptr
stl next.wptr -- wptr set for entry

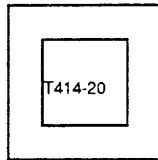
j L7-L8 -- go back for remaining blocks

L8:
```

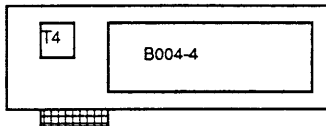
# THE TRANSPUTER ON-CHIP MEMORY



32 bit memory cycle in 66ns  
minimum external cycle of 200ns



32 bit memory cycle in 50ns  
minimum external cycle of 150ns



32 bit internal cycle in 66ns  
External memory cycle of 330ns

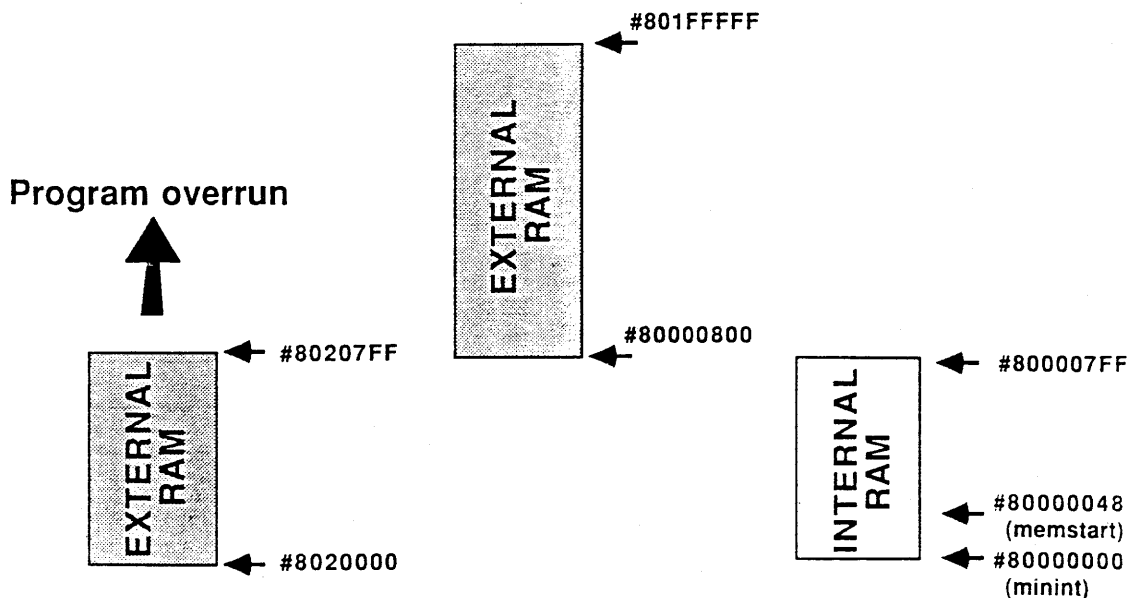
Internal RAM has 5x bandwidth of external RAM

LJP/MAC/perform2/01

# TYPICAL TRANSPUTER ADDRESS MAP

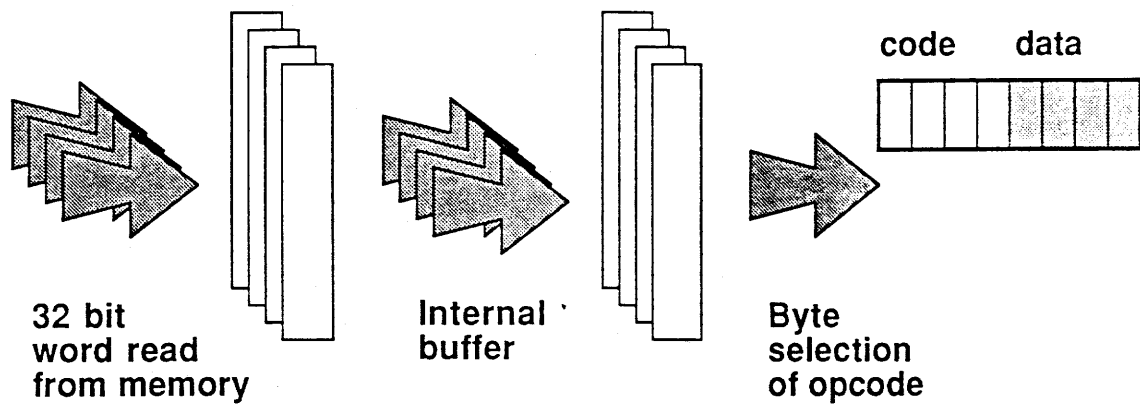


T414 with 2 Megabytes external RAM



MAC/LJP/perform/00

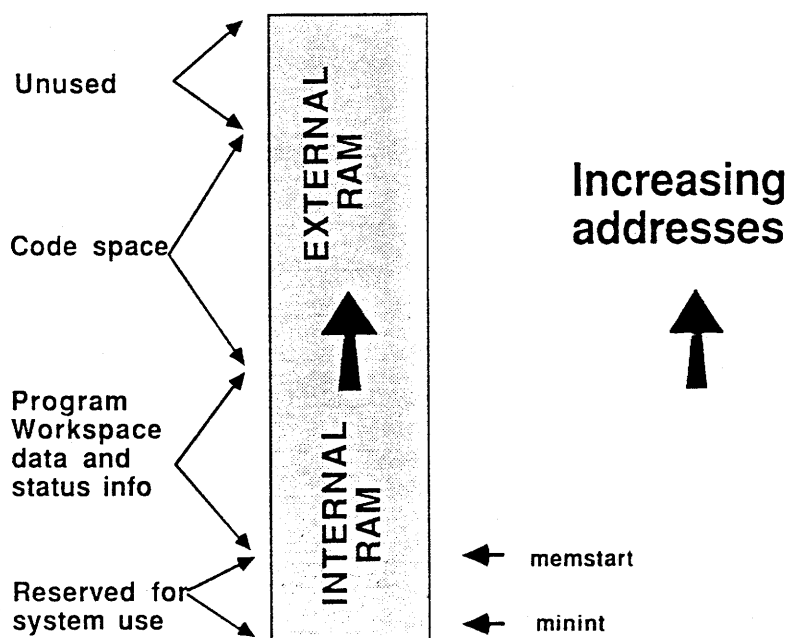
## T414 INSTRUCTION PREFETCH



There are always 4 or more instructions buffered.

LJP/MAC:perform2/00

## OCCAM COMPILED CODE LAYOUT

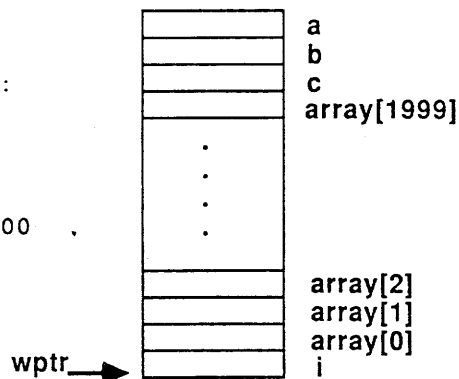


MAC/LJP/per14/01

## WORKSPACE ALLOCATION

```
INT32 a, b, c:
[2000] INT32 array:
```

```
SEQ
 a := 42
 b := 98
 SEQ i = 0 FOR 2000
 array[i] := a
```



MAC/LJ2/perform/02

## WASTING ON-CHIP MEMORY

```
INT a, b, c :
[2000] INT32 largeVector :
```

```
PROC totallyWasteOnChipRam ()
 [2000] INT32 largeData : -- all on-chip RAM taken
 SEQ i = 0 FOR 2000
 largeData [i] := i
 :
```

```
SEQ
 totallyWasteOnChipRam ()
 a := 42 -- all variables will be
 b := a * (a + 5) -- in external RAM
 SEQ i = 0 FOR 2000
 largeVector [i] := 0 -- including i
```

## USING ON-CHIP MEMORY

```
[2000] INT32 largeVector, vectorForProc :
INT a, b, c :

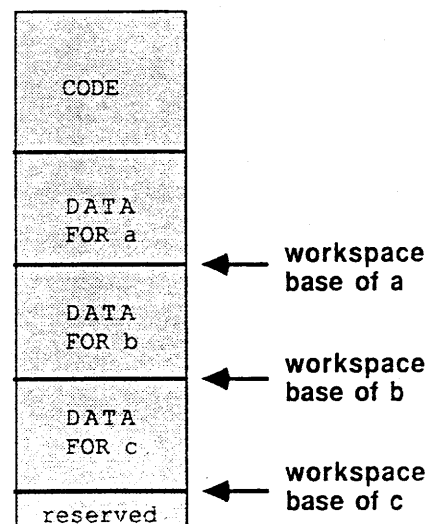
PROC totallyWasteOnChipRam ()
 [2000] INT32 largeData IS vectorForProc :
 SEQ i = 0 FOR 2000
 largeData [i] := i
 :

SEQ
 totallyWasteOnChipRam () ,
 a := 42 -- all variables will be
 b := a * (a + 5) -- in internal RAM
 SEQ i = 0 FOR 2000
 largeVector [i] := 0 -- including i
```

## WORKSPACE LAYOUT IN A PAR

```
... proc a
... proc b
... proc c

PAR
 a ()
 b ()
 c ()
```



## USING ON-CHIP MEMORY

```

PAR
 [2000] INT vector: -- vector is in local workspace
 a (vector)
 [2000] INT vector:
 b (vector)
 [2000] INT vector:
 c (vector)

[2000] INT vecA, vecB, vecC :
PAR
 a (vecA) -- local variables on-chip
 b (vecB)
 c (vecC)

```

## USING ABBREVIATIONS

```

PROC initNode (VAL INT nodePtr)
 SEQ
 tree [nodePtr + n.reflect] := nil
 tree [nodePtr + n.refract] := nil
 tree [nodePtr + n.next] := nil
 tree [nodePtr + n.object] := nil
 :

PROC initNode (VAL INT nodePtr)
 node IS [tree FROM nodePtr FOR nodeSize] :
 SEQ
 node [n.reflect] := nil
 node [n.refract] := nil
 node [n.next] := nil
 node [n.object] := nil
 :

```

## USING ABBREVIATIONS

```
tree [nodePtr + n.refract] := nil
```

|       |                  |    |                                         |
|-------|------------------|----|-----------------------------------------|
| ldc   | <i>nil</i>       | -- | get data to save                        |
| ldl   | <i>nodePtr</i>   | -- | load node offset                        |
| ldl   | <i>static</i>    | -- | get static chain                        |
| ldnlp | <i>tree</i>      | -- | get pointer to <i>tree</i> base         |
| wsub  |                  | -- | combine base and offset for destination |
| stnl  | <i>n.refract</i> | -- | save the data                           |

```
node [n.refract] := nil
```

|      |                  |    |                      |
|------|------------------|----|----------------------|
| ldc  | <i>nil</i>       | -- | get data to save     |
| ldl  | <i>node</i>      | -- | load pointer to node |
| stnl | <i>n.refract</i> | -- | and store at offset  |

The overhead is fixed so benefits increase with multiple accesses.

## USING ABBREVIATIONS FOR BYTE ACCESSES

```
[big] INT32 vector :
INT32 word :
[4] BYTE bWord RETYPES word :
BYTE b0 IS bWord [0] :
BYTE b1 IS bWord [1] :
BYTE b2 IS bWord [2] :
BYTE b3 IS bWord [3] :
SEQ i = 0 FOR big
 SEQ
 word := vector [i]
 ... use b0,b1,b2 and b3 repeatedly
```

Abbreviations add a run time overhead for initialising.

If used more than one they add performance.

They can add readability, eg with multi-dimensional arrays.

## USING ABBREVIATIONS IN OPENED LOOPS

Loops are about 20 times slower than addition.  
Including more work per loop improves speed but adds code.

```
SEQ i = 0 FOR 20000
 a[i] := b[i] + c[i]
```

May be sped up by opening the loop

```
SEQ i = 0 FOR loops
 VAL base IS i << 4:
 aSlice IS [a FROM base FOR 16] :
 bSlice IS [b FROM base FOR 16] : --ldl bSlice
 cSlice IS [c FROM base FOR 16] : --ldnl 15
 SEQ --ldl cSlice
 aSlice [0] := bSlice [0] + cSlice[0] --ldnl 15
 aSlice [1] := bSlice [1] + cSlice[1] --add
 ... etc --ldl aSlice
 aSlice[15] := bSlice[15] + cSlice[15] --stnl 15
```

## PLACING CRITICAL VECTORS ON-CHIP

```
PROC clearScreen (VAL BYTE pattern)
 [256][1024] BYTE screen RETYPES screenRAM [scrNum] :
 [1024] BYTE fastVec : -- this is in on-chip memory
 SEQ
 initBYTEvec (fastVec, pattern, 1024)
 SEQ y = 0 FOR 256
 screen [y] := fastVec
 :
```

will be quicker than

```
PROC clearScreen (VAL BYTE pattern)
 [512*512] BYTE screen RETYPES screenRAM [scrNum] :
 initBYTEvec (screen, pattern, 512*512)
 :
```

As fastVec is accessed many times it should be on-chip.

## BEWARE THE PLACE STATEMENT

The **PLACE** statement is ideal for handling peripherals eg

```
BYTE inputStatusRegister, inStatus :
PLACE inputStatusRegister AT #20000001 :
inStatus := inputStatusRegister
```

It is not the best way to use internal RAM.

```
INT a: -- ldc 3
PLACE a AT 30 : -- mint the absolute
address
a := 3 -- stnl 30 offset from MININT
 -- 6 cycles total

INT a: -- ldc 3 in on-chip RAM
a := 3 -- stl a offset from wptr
 -- 2 cycles total
```

Note that **PLACE** can overlay used memory.

## UTILISING THE BLOCK MOVE INSTRUCTION

Block moves are directly supported by occam  
and by the transputer instruction **move**.

For example

```
[65536] BYTE v1, v2 :
[v1 FROM 0 FOR 65536] := [v2 FROM 0 FOR 65536]
```

which is similar to writing `v1 := v2`

This compiles down to only four instructions,

```
ldl v1 -- assuming the vectors are
ldl v2 -- abbreviated locally
ldc 65536 -- obviously prefixed
move -- transfer the data
```

## ILLEGAL USE OF THE T414 BLOCK MOVE

A fast vector initialiser can be written using **move**

Non-architectural

-- but works on T414 for word aligned vectors

```
PROC illegalInitBYTEvec ([] BYTE vec,
 VAL BYTE pattern,
 VAL INT n)

 SEQ
 vec [0] := pattern -- clear first word
 vec [1] := pattern -- since block moves read
 vec [2] := pattern -- and write words at
 vec [3] := pattern -- full memory bandwidth.

 [vec FROM 4 FOR n - 4] := [vec FROM 0 FOR n - 4]
:

** Note ** the use of overlapping segments is device dependent.
```

## PROPER USE OF THE T414 BLOCK MOVE

```
PROC initBYTEvec ([] BYTE vec,
 VAL BYTE pattern,
 VAL INT bytes)

 INT dest, transfer :
 SEQ
 transfer := 1
 dest := transfer
 vec [0] := pattern
 WHILE dest < bytes
 SEQ
 [vec FROM dest FOR transfer] :=
 [vec FROM 0 FOR transfer]
 dest := dest + transfer
 transfer := transfer + transfer
 :
```

Logarithmic assignment on powers of 2 vector sizes.

## THE TRANSPUTER MULTIPLICATION TIME

The \* operator in occam takes 38 processor cycles.

This is a full 32 by 32 with error checked overflow.

The T414 has a fast multiply instruction **TIMES**

**TIMES** takes significant bits in RHS + 3 cycles.

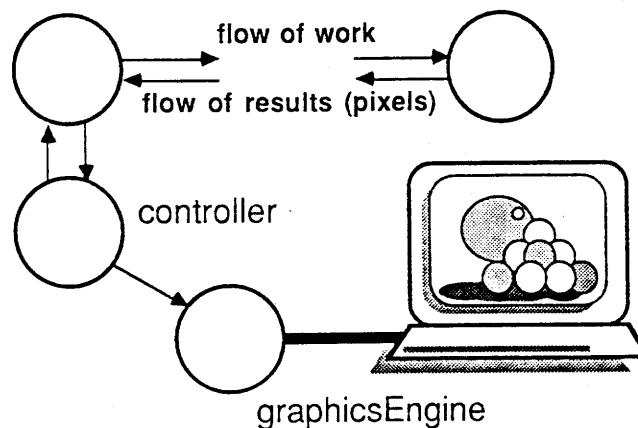
eg     a **TIMES** 4

with 3 significant bits, takes only 6 processor cycles.

This would be 300ns on a T414-20 rather than 1.9 $\mu$ s for \*

**TIMES** should be used anywhere when overflow is unimportant and in particular with **TIMERS**.

## THE RAY TRACING PROGRAM



# DECOUPLING COMMUNICATION AND COMPUTATION

Transputer links are autonomous DMA engines.  
 Processor merely initiates link transfer.  
 Larger data packets reduces processor involvement.

out ! [data FROM 0 FOR 300] is far superior to

```
SEQ i = 0 FOR 300
 out ! data [i]
```

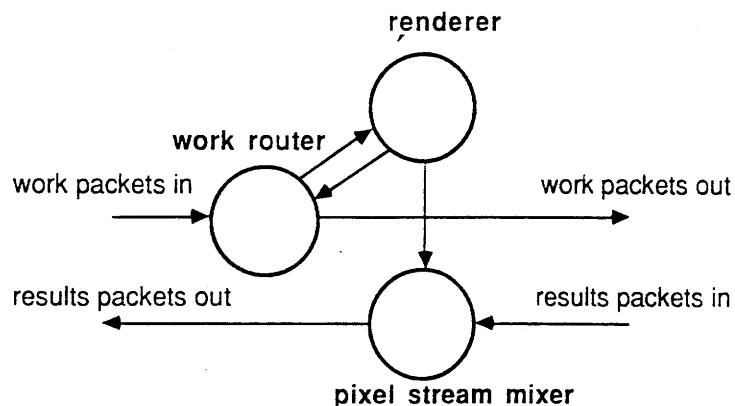
Links communication only fractionally degrades processor.  
 Utilise hardware by running links and processor in parallel.

```
PROC transform (CHAN in, out)
 [datasize] INT data :
 WHILE TRUE
 SEQ
 in ? data
 applyTransform (data)
 out ! data
 -- decoupling I/O
 PAR
 buffer (in, a)
 transform (a, b)
 buffer (b, out)
```

LJP/VAX/MultiP/00

Performance : February 26, 1987

# RAY CALCULATING ELEMENT



The router and mixer are prioritised over the renderer.

## PASSING LINK DATA BY REFERENCE

```
SEQ
 input (a) -- startup sequence
 PAR
 input (b)
 transform (a) -- output now ready
 WHILE TRUE
 SEQ
 PAR
 input (c) -- round-robin between
 transform (b) -- input output and
 output (a) -- transformer
 PAR
 input (a)
 transform (c)
 output (b)
 PAR
 input (b)
 transform (a)
 output (c)
```

## PRIORITISATION

Prioritisation is a key issue in parallel systems.

To enable links and processor to work concurrently

- 1 Run 2 parallel processes for each link.
- 2 Minimise processor involvement in routing.  
Large message packets. Routing tables.
- 3 Ensure the links have been given work to do,  
ie, **prioritise the communication**

A PRI PAR should have only two components,  
nested PARs allow many processes to run at each priority.

```
PRI PAR
 PAR
 ... high priority processes
 PAR
 ... low priority processes
```

# TYPICAL PRIORITISATION OF COMMUNICATION

```

.WHILE TRUE
 SEQ
 PRI PAR
 PAR
 input (c)
 output (a)
 transform (b)
 PRI PAR
 PAR
 input (a)
 output (b)
 transform (c)
 PRI PAR
 PAR
 input (b)
 output (c)
 transform (a)

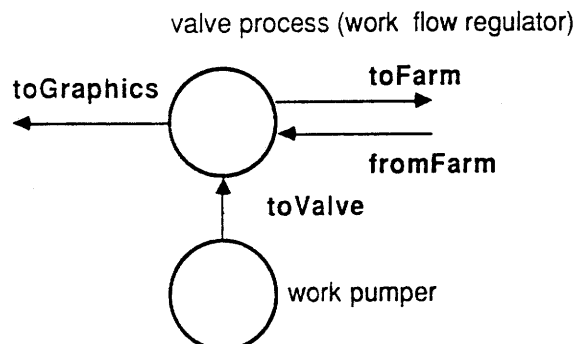
```

-- Note the counter intuitive  
 -- necessity of computation  
 -- being at the lower priority.

LJP/VAX/MultiP/03

Performance : February 26, 1987

# PROCESS FARM WORK CONTROLLER



The valve is prioritised over the work pumper.

## REGULATING WORK FLOW INTO A PROCESSOR FARM

PAR

-- pump work unconditionally

SEQ i = 0 FOR workPackets

inject ! packet

-- regulate flow of work into farm, 'idle' the valve

SEQ

idle := processors

WHILE running

PRI ALT

-- Prioritise results

from.farm ? results, -- as each farm processor

idle := idle + 1 -- can buffer work

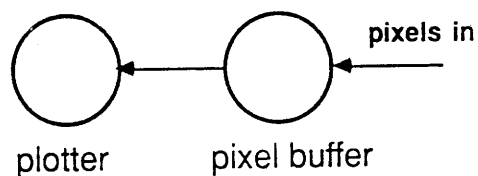
(idle > 0) & inject ? packet

SEQ

to.farm ! packet

idle := idle - 1

## THE GRAPHICS PROCESS



The buffer is prioritised over the plotter

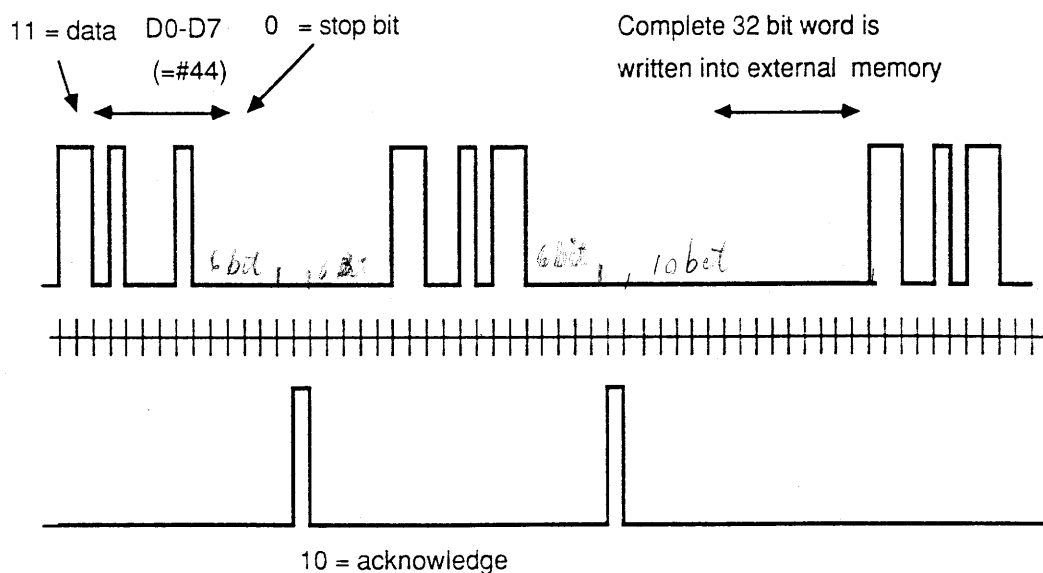
## DE-OPTIMISING THE RAY TRACE FARM

| Features             | Timings  | Relative degradation                              |
|----------------------|----------|---------------------------------------------------|
| Fully optimised      | 143.28s  | 0 %                                               |
| Extra Buffering      | 143.47s  | 0.1%                                              |
| No global vectors    | 190.74s  | 33%                                               |
| No prioritised comms | 156.56s  | 9%<br>becoming even worse<br>with more processors |
| All the above        | 199.87s  | 39%                                               |
| Prioritised compute  | 1862.39s | 1300%<br>Clearly a costly mistake                 |

LJP/VAX/MultiP/09

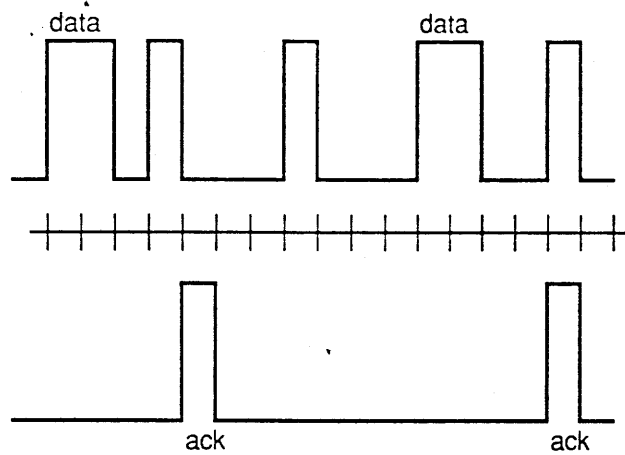
Performance : February 25, 1987

## T414 LINK PROTOCOL



Typical 23 bit times for one byte transferred  
Data rate = 10Mbits \* 8/23 = 430kbytes/sec

# T800 LINK PROTOCOL



**Acknowledges overlap data, linkin data is buffered.**

**Typical 11 bit times for one byte transferred.**

**Data rate = 10Mbits \* 8/11 = 900kbytes/sec.**

## ACCESSING FILES

Programs running in the TDS have access to host facilities via channels which are passed as parameters.

```
{{{ EXE a user program
PROC example (CHAN OF ANY keyboard, screen,
 [4]CHAN OF ANY FromUserFiler, ToUserFiler)
 ... occam source text
:
}}}
```

A pair of user channels `ToUserFiler[i]` and `FromUserFiler[i]` control data flow to and from a file as well as status and control information.

`keyboard` and `screen` provide a user terminal interface.

## FOLDS AND FILES 1

Utilities operate on the fold under the cursor.

Similarly a user program accesses data and files /split from the structure under the cursor when run.

The outermost fold can be treated in two ways:

- 1 **Like a directory** - Folds within the bundle, under the cursor, may be open, read, written, deleted etc.
- 2 **As a file** - the contents may be read or (if empty) written.

These two modes can occur sequentially but not in parallel.

## Folds and Files 2

```
{{{ A fold bundle
...F The first fold (filed)
There can also be text inside the outermost fold
which will be ignored by the filer

... The second fold (not filed)
...F graphics.tsr Another filed fold
}}}
```

To access the contents of any of these four folds we run the program with the cursor on the line `... A fold bundle`.

The folds are numbered, the outer fold is number 0. Those inside are numbered 1, 2, 3, 4 and so on, any four of which can be open simultaneously.

## Fold Attributes

The attributes of a fold define:

- 1 **The fold type** - displayable text, executable code, a compilable set of folds, etc.
- 2 **The fold contents** - program source, object code, link information etc.
- 3 **The fold indentation** - relative to the enclosing fold.

For example the attributes

```
[ft.opstext, fc.source.text, 2]
```

define a fold whose contents could be :

- displayed by the editor (`ft.opstext`)
- used as input to a compiler (`fc.source.text`)
- and is indented by 2 spaces within the enclosing fold.

## Data Streams

Once a file has been opened using a pair of filer channels, that channel pair may be used to read or write a stream of data.

After the file is closed, that pair of channels will again be available for issuing filer commands.

Up to 4 folds/files may be open at any one time.  
If closed and reopened then more than 4 folds can be used in an EXE.

### NOTE

If the outermost fold is opened for reading or writing then none of the inner folds may be opened.  
ie, we are using fold number 0 as a file and not a directory.

## The Filer Channels

The user filer channels are used for two purposes:

- 1 To send commands to the **user filer** and receive the results corresponding to these commands.

This allows the user to create or open files etc.

- 2 To communicate a **stream of data** when reading or writing a file.

This allows sequential access to folded files or files in the host operating system format.

## Command Protocol 1

Commands to the file system are all single byte values, or tags, which are followed by the parameters of the command.

Similarly, to communicate differing types of data, each item of data is preceded by a tag indicating its type.

For example: a file is opened for reading by sending `uf.open.data.read` followed by the fold number of the file.

Commands to the user filer are all prefixed by `uf`.

## Command Protocol 2

The commands sent by a process reading a data stream are identified by `fsc`. (file stream control) tags.

For example: to read the next item from the stream one sends the command `fsc.read`.

The transmitting process may then respond to this request with data or a status/error record.

## Command Protocol 3

Results are returned from the user filer and the data stream preceded by file stream data `fsd.` tags.

For example: the next record in the file is returned as a tag followed by the length of the record then a vector of bytes;

`fsd.record; length; [length]record`

Status and error values are returned preceded by tags beginning `fi..` For example: a successful command will return `fi.ok`.

## Command Protocol 4

The file stream protocol is symmetrical.

The file streamer sends the same data and commands when being read as the user process sends when writing a file. And vice-versa.

A process which reads from the file stream and a process which writes to it can be connected together by a pair of channels and communicate with each other.

## Opening a File

Files can be opened for reading or writing in one of three modes:

### **data stream**

A sequence of records is written to/read from the file with no fold information.

### **text stream**

As data mode but the file is in host operating system format (e.g. MS-DOS).

### **fold stream**

As well as data read/write commands there are commands to navigate/create the fold structure of the file.

## Opening a File For Reading

In the following examples all file access will be done in **data** mode.

~~Text mode uses the same protocol~~  
**folded** mode uses a superset of this protocol.

To open a file inside the fold bundle:

```
toFiler ! uf.open.data.read; fold.number
fromFiler ? tag; status
```

If the operation succeeds then **tag** is **fsd.result** and **status** is **fi.ok**,

otherwise **tag** is **fsd.error** and **status** is an error code.

## Creating a File For Output

Only an empty fold may be opened for output so, in order to write to a file, it is necessary to create a fold, file it and then open it:

```
toFiler ! uf.create.fold; attributes
fromFiler ? tag; fold.number
```

If the fold creation was successful, the tag returned is

```
fsd.number.of.folds, otherwise fsd.error
```

```
toFiler ! uf.make.filed; fold.number; 8; "filename"
fromFiler ? tag; result
```

```
toFiler ! uf.open.data.write; fold.number
fromFiler ? tag; status
```

## Reading Data From a File 1

Communication with the file data streamer is in the form of a sequence of requests and replies.

The process reading a file starts by sending an **fsc.read** command to the data stream.

This usually causes the data stream to return one of the following:

- 1    **fsd.record** - this is followed by the next record in the file in the form **length** then an array of **length** bytes.
- 2    **fsd.endstream** - indicates the end of the file

## Reading Data From a File 2

In response to the `fsd.read` tag the data stream can also return

`fsd.error; status` or `fsd.number; integer`

The `fsd.number` tag will probably never be encountered as it is no longer used by the TDS.

In response to any of these, the user may send another `fsc.read` to read the next record or `fsc.close; fi.ok` to close the file.

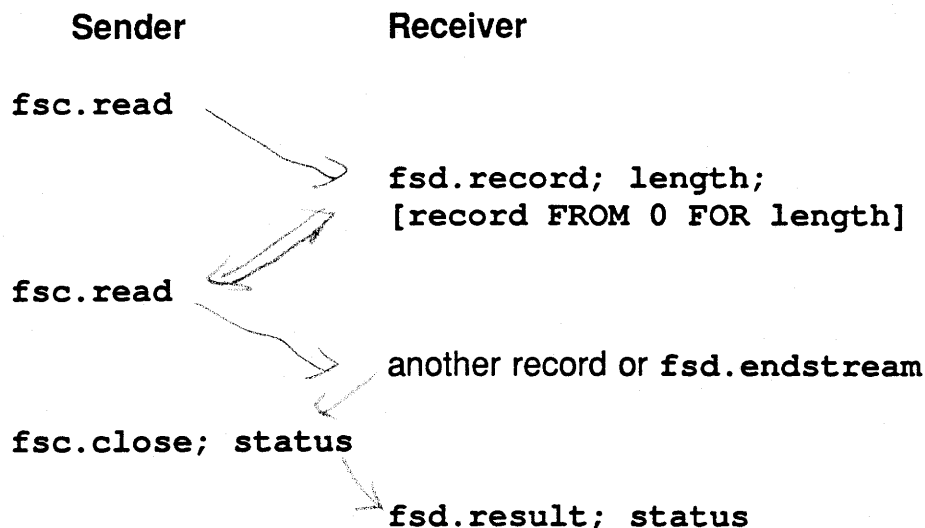
## Writing Data To a File

To write to a file, the sequence of operations is identical to reading but with the roles of filer and user reversed.

The filer starts by sending an `fsc.read`  
the user sends either `fsd.record` or `fsd.endstream`.

After an endstream or a filing system error the  
filer will send `fsc.close` requesting the user to stop writing.

## Data Stream Summary



## KERNEL FILER CHANNELS

Programs which use keyboard and screen can run standalone.

The user filer needs the editor and fold manager

Standalone programs only have the server.

Applications programs should not include the TDS.

The TDS server supports reading/writing of 1 DOS file.

This requires special kernel channels.

```

{{{ EXE
{{{F main.prog
PROC main.prog (CHAN OF ANY keyboard, screen,
 [4]CHAN OF ANY from.filer, to.filer,
 CHAN OF ANY privatel, private2,
 CHAN OF ANY from.kernel, to.kernel)
 ... direct file access to server
:
}}}
```

## KERNEL FILER USAGE

The kernel filer is accessed in

`c:\tdsiolib\msdos.tsr`

*procedure*

Constants required are in

`c:\tdsiolib\filerhdr.tsr`

Further information is in `c:\server`

and the server documentation (in beta2 release).

## KERNEL FILER LIBRARY PROCEDURES 1

```
PROC test.exists (CHAN OF ANY from.filer, to.filer,
 VAL INT name.len,
 VAL [63]BYTE name, BOOL exists)
... tests whether file of name already created
:

PROC make.id (CHAN OF ANY from.filer, to.filer,
 VAL []BYTE name, -- source string
 INT id.len, [63]BYTE abs.id, -- filename
 INT type, content, -- default foldnames
 INT result) -- 0 = fi.ok
... create DOS filename from given string
:
```

## KERNEL FILER LIBRARY PROCEDURES 2

```
PROC file.lock (CHAN OF ANY from.filer, to.filer,
 VAL INT lock.command,
 -- VAL tkf.lock.read IS 1, tkf.lock.write IS 2:
 VAL INT parent.id.len,
 VAL [63]BYTE parent.id,
 -- not named file stores, set parent.id.len = 0
 INT file.id.len, [63]BYTE file.id,
 INT result)
 ... prevent multiple access to files
:

PROC file.release (CHAN OF ANY from.filer, to.filer,
 VAL INT id.len,
 VAL [63]BYTE file.id, INT result)
 ... release file from protection
:
```

## KERNEL FILER LIBRARY PROCEDURES 3

```
PROC open.file (CHAN OF ANY from.filer, to.filer,
 VAL INT open.command,
 -- VAL tkf.open.read IS 4, tkf.open.write IS 5:
 INT name.len, [63]BYTE name,
 -- Not VALs as name may change, eg truncated
 INT result)
 ... open DOS file for reading or writing
:

PROC close.file (CHAN OF ANY from.filer, to.filer,
 INT result)
 ... close file opened on these channels
:
```

## KERNEL FILER LIBRARY PROCEDURES 4

```
PROC write.block (CHAN OF ANY from.filer, to.filer,
 VAL INT length, VAL []BYTE block,
 INT result)
 ... write given data block to disk
:
PROC read.block (CHAN OF ANY from.filer, to.filer,
 INT length, []BYTE block, INT result)
 ... read block from disk
:
PROC read.line (CHAN OF ANY from.filer, to.filer,
 INT block.length, []BYTE block,
 INT block.pointer,
 -- initialise to greater than block.length
 INT line.length, []BYTE line,
 BOOL last.block, end.of.line, INT result)
 ... read a line from user defined buffer
:
```

## FOLDED FILE FORMAT

A file in the TDS is made from a sequence of elements.

- 1 *number.element*  
Followed by a non-negative number.
- 2 *record.element*  
Followed by a record length and byte sequence.
- 3 *startlist.element { element} endlist.element*  
Marks beginning and end of list structure.
- 4 *startfold.element { element} endfold.element*  
Marks beginning and end of a fold.
- 5 *startfiled.element { element} endfiled.element*  
Marks the beginning and end of a filed fold.

## FILE ELEMENT ENCODING

All elements are identified by their tag bytes.

Tag bytes are made from a 2 bit tag  
and a least significant 6 bit data field.

|                         |         |                     |
|-------------------------|---------|---------------------|
| <i>record.element</i>   | tag = 0 | size and record     |
| <i>number.element</i>   | tag = 1 |                     |
| <i>function.element</i> | tag = 2 | specials eg folds   |
| <i>prefix.byte</i>      | tag = 3 | extends data fields |

Defined function numbers

|                           |         |          |        |
|---------------------------|---------|----------|--------|
| <i>startlist.element</i>  | tag = 2 | data = 0 | ie #80 |
| <i>endlist.element</i>    | tag = 2 | data = 1 | ie #81 |
| <i>startfold.element</i>  | tag = 2 | data = 2 | ie #82 |
| <i>endfold.element</i>    | tag = 2 | data = 3 | ie #83 |
| <i>startfiled.element</i> | tag = 2 | data = 4 | ie #84 |
| <i>endfiled.element</i>   | tag = 2 | data = 5 | ie #85 |

## EXAMPLE OF TDS FORMATTED FILE

We now look at *identifier.csc* files created by the compiler from SC bundles. The format is defined as follows:

```
INT32::[]BYTE interface descriptor string
INT32::[]BYTE compiler identifier string
INT32 target processor type
INT32 compiler type/version
INT32 workspace requirement
INT32 entry point offset
INT32::[]BYTE the codelength and code
```

The TDS adds *startlist.element {record.element} endlist.element* which might from the MS-DOS point of view be:

```
80 30 01 00 00 00 02 1B 00 00 00 6F 63 63 61 6D
20 32 2C 20 62 65 74 61 2E 31 61 2C 20 31 32 2D
4D 61 79 2D 38 36 04 00 00 00 04 00 00 00 05 00
00 00 08 00 00 00 00 10 00 00 00 0C 60 BD 71 70
F5 D2 B3 22 F0 50 50 50 04 00 00 00 00 81
```

## CREATING CODE FOLDS

```
Compile all scs Creates target code from source

Configure PROGRAM Adds loader for booting down links

Extract PROGRAM Writes code with bootstrap to file
 Select the BOOTABLE option
 This creates a CODE PROGRAM fold
```

```
{{{ PROGRAM prog.tsr
...F prog.tsr source
...F prog.dcd code
...F prog.dds descriptor
...F CODE PROGRAM <-- Cursor here for 'makeboot'
}}}
```

CODE PROGRAM folds are in **userfiler** format.  
Use **EXE 'makeboot'** to create MS-DOS equivalent.

## ACCESSING THE B004 FROM THE IBM PC

The link adaptor on the B004 is mapped onto the 8086 I/O ports.  
ie the B004 is plugged into an expansion slot.

These can be accessed by the assembler instructions:

```
OUT DX, AL and IN AL, DX
```

DX contains the 16 bit Port address

AL is the 8 bit source and destination respectively.

Most compiled languages on the PC offer Port access  
although these features are often implementation dependent.

## USING THE B004 FROM TURBOPASCAL

TurboPascal is a trademark of Borland International Inc.  
A popular language for PCs.

The following are the Hex addresses for the B004

```
CONST
 linkBase = $150;
 inputData = $150; {8 bit data from C002}
 outputData = $151; {8 bit data to C002}
 inputStatus = $152; {Bit 0 = data ready from C002}
 outputStatus = $153; {Bit 0 = data sent to C002}
 resetT414 = $160; {Writing bit 0 raises reset}
 analyseT414 = $161; {Writing bit 0 raises analyse}
 error.status = $160; {Reading bit 0 is NOTerror}
```

## INITIALISING THE B004 FROM THE PC

```
PROCEDURE loopFor (i : INTEGER);
BEGIN
 WHILE i <> 0 DO {simple delay loop}
 i := i - 1;
 END;

PROCEDURE startWork;
BEGIN
 Port [analyseT414] := 0;
 loopFor (800);
 Port [resetT414] := 1;
 loopFor (3000);
 Port [resetT414] := 0;
 loopFor (1000);
 Port [inputStatus] := 0; {initialise the C002}
 Port [outputStatus] := 0;
END;
```

## TRANSMITTING DATA TO THE B004

```
PROCEDURE outByte (b : INTEGER);
BEGIN
 Port [outputData] := b;
 WHILE NOT ODD (Port [outputStatus]) DO BEGIN END;
END;

FUNCTION outputOrFail (b: INTEGER);
CONST failed = 32000
VAR loops : INTEGER
BEGIN
 loops := 0;
 Port [outputData] := b;
 WHILE (loops < failed) AND
 NOT ODD (Port[outputStatus]) DO BEGIN
 loops := loops + 1;
 END;
 outputOrFail := ODD (Port [outputStatus]);
END;
```

## RECEIVING FROM THE B004

```
FUNCTION inByte : INTEGER;
BEGIN
 WHILE NOT ODD (Port [inputStatus]) DO
 BEGIN
 END;
 inByte := Port[inputData];
 END;

FUNCTION tptrErrorSet : BOOLEAN;
BEGIN
 tptrErrorSet := NOT (ODD (Port [errorStatus]));
 {Note that this is the reverse of the expected}
 {The application might analyse the T414}
END;
```

## A SERVER FILE ON THE PC

The user's server file should have options to:

- 1 Open the DOS file made by the **makeboot** Utility.  
*Result transfer*
- 2 Output the contents byte-by-byte to the link adaptor.
- 3 Open bi-directional communications.
- 4 Add/remove tags for multiplexed channels.  
eg screen = 0, keyboard = 1, fileout0 = 3, etc
- 5 Monitor the error pin for analysis/message.
- 6 Be prepared to terminate.

# THE TDS SERVER

File running on the 286  
interfacing to MS DOS 2.0 or later.

Written in iAPX-86 assembly language  
Alternative server example written in microsoft C.

Configurations for IBM PC and NEC PC-9800.

[d:] [path] **SERVER** {option}

|                |                    |                                                          |
|----------------|--------------------|----------------------------------------------------------|
| <i>options</i> | -b <i>filename</i> | boot file onto board<br>default is <b>boot.b4</b>        |
|                | -t <i>filename</i> | create empty TDS <b>.top</b> file                        |
|                | -l <i>address</i>  | specify link adaptor address<br>eg <b>server -l #310</b> |

## SERVER OVERVIEW 1

The servers first task is reading the command parameters.  
Any unrecognised text is saved for program interrogation.

The server next tries to boot the transputer after  
outputting **Booting root transputer ...**

- 1 The server asserts and deasserts subsystem reset.
- 2 The file is sent byte by byte to the link.  
There are messages for file non-existence,  
File unopenable and link timeout.

After booting the transputer the keyboard, link and  
subsystem error are polled.

## SERVER OVERVIEW 2

- keyboard**
- The server reads the keyboard
  - keys may be *cooked*
  - Results are output to the link
  - cntrl break and cntrl Y interrupt
  - Choice of rebooting using analyse
- link**
- The server polls the link
  - channels are demultiplexed
  - for files, screen etc
- subsystem error**
- error set message to screen
  - Choice of rebooting using analyse

TDS documentation will include server protocol.

## SERVER PROTOCOL

All link communication to the server is tagged.

The tags are type INT, and denote the channel being used.

|                      |        |                   |
|----------------------|--------|-------------------|
| VAL to.kernel.chan   | IS 0:  | -- special server |
| commands             |        |                   |
| VAL from.kernel.chan | IS 1:  |                   |
| VAL to.terminal.chan | IS 2:  | -- screen         |
| VAL from.terminal    | IS 3:  | -- keyboard       |
| VAL to.comms         | IS 29: | -- serial line    |
| VAL from.comms       | IS 28: |                   |
| VAL to.filer0.chan   | IS 32: | -- filer channels |
| VAL from.filer0.chan | IS 48: |                   |

These are followed by the terminal tags, also of type INT.

eg LinkToServer ! to.terminal.chan; INT tt.goto; X; Y

## SIMPLE SCREEN AND KEYBOARD SUPPORT

```

PROC convert (CHAN OF ANY linkIn, LinkOut,
 keyboard, screen)
 PAR
 ... process to strip off from.terminal.chan
 {{{ process to add to.terminal.chan
 BYTE tt.tag:
 SEQ
 screen ? tt.tag
 LinkOut ! to.terminal.chan; INT tt.tag
 IF
 tt.tag = tt.out.byte -- the exception
 [2]BYTE data:
 SEQ
 screen ? data[0]
 LinkOut ! data
 TRUE
 ... check for tt.out.string, tt.goto etc
 }}}
 :

```

## THE KERNEL CHANNELS ~~CHANNELS~~

```

VAL INT32 k.terminate.server IS 7:
VAL INT32 k.store.item IS 8:
VAL INT32 k.retrieve.item IS 9:
VAL INT32 k.get.char IS 10:

```

ie LinkOut ! to.kernel.chan; k.terminate.server  
is the normal way to return control to DOS.

LinkOut ! to.kernel.chan; k.store.item; pos; val  
allows the storage of up to 4xINT32s in server

LinkOut ! to.kernel.chan; k.retrieve.item; pos  
will provoke LinkIn ? from.kernel.chan; val

Sending k.get.char is a command to tell the server to  
send back the next DOS command line character.  
Server specific commands are removed, ending in 0 (null).

## OTHER CHANNELS

The TDS refers to files by the cursor position.  
In stand alone mode there is no editor.

There is a file in `c:\tds\example` called `ops.tsr`

This behaves like the TDS in the use of files.  
Also a good example of server protocol conversion.

The best definition of the server interaction is  
the server source, supplied in C or 8086 assembler.

Progressive changes to the server and the program  
can be made to suit customer requirements.

## MAKING BOOTABLE FILES

- 1 Include interface converting tt/ft protocol
- 2 Configure as a **PROGRAM**  
Lowest transputer number for host transputer.  
**PLACE** channels to server on link to host.
- 3 Compile, configure and extract.  
Set output.fold parameter to **BOOTABLE**  
This creates a **CODE PROGRAM** fold
- 4 Get the **EXE "makeboot"**  
Run **"makeboot"** on the **CODE PROGRAM**  
Specify DOS filename with **.b4** extension
- 5 Enter DOS and type **server -b filename.b4**