Author: Roger Shepherd

1 TRANSPUTER INSTRUCTION SET

1.1 Notation

In this document the notation used is that of occam 2, with the assumption that the variables of type INT are infinite-bit two's complement integers.

Any particular processor is assumed to have a finite word length, each register in the processor holding the value of the corresponding variable in the following description. It is therefore natural to interpret a word as a fixed length two's-complement integer. Before and after execution of any instruction, the numerical value taken by each variable is correctly representable in the corresponding single word register.

Constants

The following constants are used in the description of the machine.

BitsInWord	The number of bits in a machine word.
Range	The number of distinct values storeable in a word. (**Range = 2**BitsInWord**)
MaxInt	The largest (most positive) value representable in a word. (**MaxInt = (Range/2) - 1**).
MinInt	The smallest (most negative) value representable in a word. (**MinInt = -(Range/2)**).

Procedures

The following two procedures are used. They do not affect the value held in a processor register; only the value of the corresponding variable. Consequently, they are used in the following description to change the interpretation of the register value, rather than the value itself.

```
PROC UnSign(INT reg)
  IF
    reg < 0
      reg := reg + Range
    TRUE
      SKIP
:

PROC Sign(INT reg)
  IF
    reg > MaxInt
      reg := reg - Range
    TRUE
      SKIP
:
```

The procedure **RestoreToRange** is used if the result of an operation (such as addition) may have taken the value stored in a register into the range **[-Range, Range-1]** rather than range permitted for a signed integer (**[-Range/2, (Range/2) - 1]**). The effect of this can be thought of as throwing away the bits of higher significance than the sign bit of the register.

```
PROC RestoreToRange(INT register)
  IF
    register > MaxInt
      register := register - Range
    register < MinInt
      register := register + Range
    TRUE
      SKIP
:
```

The procedure **Later** produces the value of **(T1 AFTER T2)**. This is dependant on the wordlength of the processor.

```
PROC Later(VAL INT T1, T2, BOOL laterFlag)
  INT timeDiff :
  SEQ
    timeDiff := T1 - T2
    RestoreToRange(timeDiff)
    laterFlag := (timeDiff > 0)
:
```

1.2 Summary of Registers, Flags and Special Locations

Timer:

`ClockReg[0]` the current value of the high priority processor clock

`ClockReg[1]` the current value of the low priority processor clock

`TPtrLoc[0]` either indicates that the level 0 timer is not in use or points to the first process on the level 0 timer queue

`TPtrLoc[1]` either indicates that the level 1 timer is not in use or points to the first process on the level 1 timer queue

`TNextReg[0]` indicates the time of the first event on the level 0 timer queue

`TNextReg[1]` indicates the time of the first event on the level 1 timer queue

`TEnabled[0]` indicates whether there is anything on the level 0 timer queue

`TEnabled[1]` indicates whether there is anything on the level 1 timer queue

Priority 0 Queue control:

`FptrReg[0]` pointer to front of active process list

`BptrReg[0]` pointer to back of active process list

Priority 1 Queue control:

`FptrReg[1]` pointer to front of active process list

`BptrReg[1]` pointer to back of active process list

Sequential process execution:

`IptrReg` pointer to next instruction to be executed

`WdescReg` process descriptor of the current proces

`Areg` top of evaluation stack

`Breg` middle of evaluation stack

`Creg` bottom of evaluation stack

`Oreg` operand register

`StatusReg` contains status information - see below

Initialisation, booting and analysis

`MemStart` this is the most negative word in store not used by the machine for any special purpose (eg as a link-channel process word, register save word or timer pointer).

Interrupt save area:

SaveBase the base address of the area of store used to save the registers of a low priority process while a high priority process is executing.

WdescIntSave the offset of the word containing the **Wdesc** register of an interrupted process within the save area.

IptrIntSave the offset of the word containing the **Iptr** register of an interrupted process within the save area.

AregIntSave the offset of the word containing the **Areg** register of an interrupted process within the save area.

BregIntSave the offset of the word containing the **Breg** register of an interrupted process within the save area.

CregIntSave the offset of the word containing the **Creg** register of an interrupted process within the save area.

EregIntSave the offset of the word containing the **Ereg** register of an interrupted process within the save area.

STATUSIntSave the offset of the word containing the **STATUS** register of an interrupted process within the save area.

Extra registers

Ereg carries descriptor of process to be scheduled on completion of a message transfer. This is only be used during the execution of block move.

BMbuffer used to hold information between successive stages of a block move.

StatusReg

The only bits in the StatusReg of concern to the assembler programmer are the HaltOnErrorBit and the ErrorFlag. The procedures which are used to manipulate these are given below. The remaining bits in the STATUSreg are used by the processor to control the execution of interruptable instructions; their state only becomes visible when the STATUSreg is saved during the execution of a high priority process.

Bit	Name	Purpose
1	GotoSNPBit	causes processor to execute **StartNextProcess**
2	IOBit	set by Input and Output before entry to block move
3	MoveBit	indicates block move is being executed
4	TimeDelBit	indicates a deletion from the timer queue
5	TimeInsBit	indicates an insertion into the timer queue
6	DistAndInsBit	(does not appear in this description but is actually use in the processor)
7	HaltOnErrorBit	Cause processor to halt when an error is generated (this is edge triggered)
msb	ErrorFlag	

```
PROC SetErrorFlag()
  StatusReg := StatusReg BITOR ErrorFlag
:

PROC ClearErrorFlag()
  StatusReg := StatusReg BITAND (BITNOT ErrorFlag)
:

PROC ReadErrorFlag(BOOL state)
  state := ((StatusReg BITAND ErrorFlag) <> 0)
:

PROC SetHaltOnErrorFlag()
  StatusReg := StatusReg BITOR  HaltOnErrorBit
:

PROC ClearHaltOnErrorFlag()
  StatusReg := StatusReg BITAND (BITNOT HaltOnErrorBit)
:

PROC ReadHaltOnErrorFlag(BOOL state)
  state := ((StatusReg BITAND HaltOnErrorBit) <> 0)
:
```

The procedure **OverflowCheck** sets the **ErrorFlag** if its argument is not in the range of representable values and then forces its argument to lie within that range. It does this by ignoring bits beyond the most significant representable bit.

```
PROC OverflowCheck(INT register)
  IF
    (register < MinInt) OR (register > MaxInt)
      SEQ
        SetErrorFlag()
        register := register REM Range
        RestoreToRange(register)
    TRUE
      SKIP
:
```

1.3 Workspace

In the following description, the process descriptor of the current process is also held as two variables **Wptr** and **Priority**. These are updated as follows

```
PROC UpdateWdescReg(VAL INT NewWdescReg)
  SEQ
    WdescReg := NewWdescReg
    Wptr     := WdescReg BITAND (-2)
    Priority := WdescReg BITAND   1
:
```

Consequently, **Wptr** always holds a pointer to the current process workspace, and **Priority** always holds the priority of the current process.

For each concurrent process, a number of locations are used to hold scheduling information. These locations are accessed using fixed word offsets from the workspace pointer, as follows:

```
Iptr.s      =   -1

Link.s      =   -2

State.s     =   -3
Pointer.s   =   -3

TLink.s     =   -4

Time.s      =   -5
```

Local 0 is used by the instructions which implement **ALT**ernative.

1.4 Special values

The special value taken by a channel location:

```
NotProcess.p =   MinInt
```

The special values taken by the State location in the implementation of channel guards are:

```
Enabling.p   =   MinInt + 1
Waiting.p    =   MinInt + 2
Ready.p      =   MinInt + 3
```

The special values taken by the Tlink location in the implementation of timer guards are:

```
TimeSet.p      MinInt + 1
TimeNotSet.p   MinInt + 2
```

The values of true and false are:

```
MachineTRUE    1
MachineFALSE   0
```

The value used stored in local 0 to indicate that no selection has been made during an **ALT**ernative input:

```
NoneSelected.o  =   -1
```

1.5 Memory Access Procedures

In the description of the processor and instruction the following memory access procedures are used:

`AtWord(Base, N, A)`	sets A to point at the Nth word past Base
`AtByte(Base, N, A)`	sets A to point at the Nth byte past Base
`RIndexWord(Base, N, X)`	sets X to the value of the Nth word past Base
`RIndexByte(Base, N, X)`	sets X to the value of the Nth byte past Base
`WIndexWord(Base, N, X)`	sets the value of the Nth word past Base to X
`WIndexByte(Base, N, X)`	sets the value of the Nth byte past Base to X

Memory addresses start from MinInt, the process locations of the links and the event channel occupying the first few locations in memory. The number of process locations used for the links and the event pin is **`LinkChans`**.

An address is a single word value divided into two parts:

a word address

a byte selector

The byte selector occupies the least significant bits in the word. The number of bits used for the byte selector is **`BselLength`**, where

```
BselLengthTab = TABLE [ 0, 0, 1, 2, 2, 3, 3, 3, 3 ]
BselLength    = BselLengthTab [ BitsInWord / 8 ]
BselMask      = ( 1 << (BselLength+1) ) - 1
```

1.6 Processor and Link-Channel interactions

Overview and terminology

The link-channels operate concurrently with, and are controlled by, the processor.

When a process executes an *output message* instruction which specifies a link-channel the processor must cause the link-channel to transfer the specified message from the transputer's memory. To do this, the processor makes a `PerformIO` request on the link-channel. This request specifies a pointer to the message, the length of the message and the priority of the process. When the message has been transferred, the link-channel signals the processor with a `RunRequest`. This will cause the processor to run the process which output the message.

When a process excutes an *input message* instruction the interactions between the processor and an input link-channel are similar. The processor makes a `PerformIO` request as before and when the message has been transferred, the link-channel signals the processor with a `RunRequest` as before.

When a process refers to an input link-channel in a guard of an alternative construct the processor makes use of two further requests on the link-channel.

The first of these, called an `Enable` request, specifies the priority of the process performing the alternative and 'enables' the link-channel. When an 'enabled' link-channel starts to receive a message it signals the processor with a `ReadyRequest`.

The second, called a `StatusEnquiry`, does two things. Firstly, it causes the link-channel to send a message to the processor indicating if it has yet started to receive a message and, secondly, it 'disables' the link-channel if it is enabled.

To reset a link-channel, the processor makes a `ResetRequest` on the link-channel. The link will return to its reset state, and, if it was not already signalling the processor with a `ReadyRequest` or a `RunRequest`, it will acknowledge with an `AckReset`.

Occam description

The processor and the link-channels are each described as separate concurrent processes. Each connection between the processor and a link-channel uses 3 channels. For the i'th link-channel these are

```
ProcessorToLink[i]
LinkToProcessor[i][0]
LinkToProcessor[i][1]
```

The processor sends requests and their parameters to the i'th link-channel on `ProcessorToLink[i]`. The i'th link-channel uses `LinkToProcessor[i][0]` to signal the processor when it is at priority 0 (high priority) and `LinkToProcessor[i][1]` when it is at priority 1 (low priority).

Each input link-channel is also connected to two further channels, `LinkInData` and `LinkInAck`. These carry the data recieved by the link-channel and the acknowledges sent by the channel.

Each output link-channel is also connected to two further channels, `LinkOutData` and `LinkOutAck`. These carry the data sent by the link-channel and the acknowledges received by the channel.

Messages on ProcessorToLink

The possible messages on **ProcessorToLink[i]** are

1 **PerformIO <priority> <pointer> <count>**

This requests the link-channel to transfer a message of **<count>** bytes starting at **<pointer>**. The priority of the link-channel for this transfer is **<priority>**. (Because a link-channel is one directional there is no need for the processor to specify the transfer direction).

2 **Enable <priority>**

This requests an input link-channel to become enabled and sets the priority of the link-channel to **<priority>**.

3 **StatusEnquiry <priority>**

This asks an input link-channel if it has started to receive a message. It also disables the link-channel if it was enabled. The link-channel responds by sending **ReadyRequest** if it has started to receive a message, **ReadyFALSE** otherwise.

4 **ResetRequest <priority>**

This is sent to reset a link-channel. The link-channel responds by returning **AckReady**, unless it was already sending a **ReadyRequest** or a **RunRequest**.

5 **AckReady**

The processor sends this to acknowledge a **ReadyRequest** made by the link-channel.

Messages on LinkToProcessor[i][0] and LinkToProcessor[i][1]

The possible message on **LinkToProcessor[i][0]** and **LinkToProcessor[i][1]** are

1 **RunRequest**

This signals that a link-channel has completed passing a message. The processor will either acknowledge the request with an **AckRun** or will reset the link-channel with a **ResetRequest**.

2 **ReadyRequest**

This signals that a link-channel has started to receive a message. It is sent either when an enabled link-channel starts to receive a message, or in response to a **StatusEnquiry**.

3 **ReadyFALSE**

This is sent in reply to a **StatusEnquiry** when the link-channel has not started to receive a message.

4 **AckReset**

This is sent in reply to **ResetRequest**.

Summary of message interactions

To clarify the processor and link-channel interactions, a trace of the behaviour of a link-channel is given below for all possible interactions. The traces given below all involve low priority process interacting with the i'th link-channel;the interactives involving high priority processor are similar but have **LinkToProcessor[i][0]** substituted for **LinkToProcessor[i][1]** and **0** substitued for **1** whenever the processor send a priority to the link-channel.

Reset

When the processor resets the i'th link-channel the interaction is :

```
SEQ
  ProcessorToLink[i] ? request; priority -- ResetRequest; 1
  LinkToProcessor[i][1] ! response
```

The **response** sent will be **AckReset**, **RunRequest** or **ReadyRequest**.

Input and Output

When the processor executes either an 'input message' or 'output message' instruction the interaction is:

```
SEQ
  ProcessorToLink[i] ? interaction; priority -- PerformIO
  ProcessorToLink[i] ? pointer; count
```

There are then three possible further traces.

1 The link-channel completes its IO :

```
SEQ
  LinkToProcessor[i][1] ! RunRequest,
  ProcessorToLink[i] ? interaction    -- AckRun
```

2 The link-channel is reset before completion of its IO :

```
SEQ
  ProcessorToLink[i] ? interaction;  priority -- ResetRequest; 1
  LinkToProcessor[i][1] ! AckReset
```

3 The link-channel is reset at the same time as it completes of its IO :

```
PAR
  ProcessorToLink[i] ? interaction;  priority -- ResetRequest; 1
  LinkToProcessor[i][1] ! RunRequest
```

Alternative Input

When the processor makes a **StatusEnquiry** on the i'th link-channel the interaction is

```
SEQ
  ProcessorToLink[i] ? token; priority -- StatusEnquiry; 1 or 0
  LinkToProcessor[i][1] ! response
```

The **response** will be **ReadyRequest** if the link-channel has started to receive a message, **ReadyFALSE** if it has not.

The processor enables the i'th link-channel as follows :

```
SEQ
  ProcessorToLink[i] ! Enable; priority
```

There are 5 possible interactions between an enabled link-channel and the processor :

1 The link-channel is not ready and the processor makes a **StatusRequest.** The trace of this interaction is described above, the link-channel returning **ReadyFALSE.**

2 The i'th link-channel signals it is ready before the processor makes another request :

```
SEQ
  LinkToProcessor[i][1] ! ReadyRequest
  ProcessorToLink[i] ? interaction
  --                   AckReady
```

3 The processor makes a **StatusEnquiry** at the same time as the link-channel sends a **ReadyRequest**

```
PAR
  LinkToProcessor[i][1] ! ReadyRequest
  ProcessorToLink[i] ? interaction;   priority
  --                   StatusEnquiry; 1
```

4 The processor makes a **ResetRequest** before the link-channel becomes ready. In this case the interaction is as described above with the link-channel responding with **AckReset.**

5 The processor makes a **ResetRequest** at the same time as the link-channel sends a **ReadyRequest**

```
PAR
  LinkToProcessor[i][1] ! ReadyRequest
  ProcessorToLink[i] ? interaction;  priority
  --                   ResetRequest; 1
```

Link-channel behaviour

```
PROC LinkOut(CHAN LinkOutData, LinkOutAck,
             CHAN FromProcessor, [2]CHAN ToProcessor)
  INT  priority, pointer, count :
  BYTE byte :
  BOOL ready, requested :
  SEQ
    requested := FALSE -- transfer requested
    ready     := TRUE  -- ready to output a byte
    WHILE TRUE
      INT token :
      PRI ALT
        FromProcessor ? token
          SEQ
            FromProcessor ? priority
            IF
              token = PerformIO
                SEQ
                  FromProcessor ? pointer; count
                  requested := TRUE
              token = ResetRequest
                SEQ
                  ready     := TRUE
                  requested := FALSE
                  ToProcessor[priority] ! AckReset

        (ready AND requested) & SKIP
          IF
            count = 0          -- No more data to be output
              INT oldPriority :
              SEQ
                requested := FALSE
                oldPriority := priority
                PAR
                  ToProcessor[oldPriority] ! RunRequest
                  INT interaction :
                  SEQ
                    FromProcessor ? interaction
                    IF
                      interaction = AckRun
                        SKIP
                      interaction = ResetRequest
                        FromProcessor ? priority

            TRUE -- Output a byte; set ready to FALSE
              SEQ
                RIndexByte(pointer, 0, byte)
                AtByte(pointer, 1, pointer)
                count := count - 1
                LinkOutData ! byte
                ready := FALSE -- wait for acknowledgement

        LinkOutAck ? token        -- AckData
          ready := TRUE
:
```

```
PROC LinkIn(CHAN LinkInAck, LinkInData,
            CHAN FromProcessor, [2]CHAN ToProcessor)

  INT  priority, pointer, count :
  BYTE byte :
  BOOL ready, requested, enabled :
  SEQ
    ready     := FALSE     -- Has a byte has been input
    requested := FALSE     -- Is transfer pending?
    enabled   := FALSE     -- Is link enabled ?
    WHILE TRUE
      INT token :
      PRI ALT
        LinkInData ? byte
          ready := TRUE

        FromProcessor ? token
          ...  deal with processor request

        (requested AND ready) & SKIP
          ...  acknowledge and store byte

        (enabled AND ready) & SKIP
          ...  inform processor that link is ready
:
```

where the folds are as follows:

```
...  deal with processor request

SEQ
  FromProcessor ? priority
  IF
    token = Enable
      enabled := TRUE
    token = StatusEnquiry
      SEQ
        enabled := FALSE
        IF
          ready
            ToProcessor[priority], ReadyRequest
          TRUE
            ToProcessor[priority], ReadyFALSE
    token = PerformIO
      SEQ
        FromProcessor ? pointer; count
        requested := TRUE
    token = ResetRequest
      SEQ
        ready     := FALSE
        enabled   := FALSE
        requested := FALSE
        ToProcessor[priority] ! AckReset
```

```
...  acknowledge and store byte

SEQ
  LinkInAck ! AckData              -- Acknowledge
  WIndexByte(pointer, 0, byte)
  AtByte(pointer, 1, pointer)
  count := count - 1
  IF
    count = 0                      -- Transfer completed
      INT oldPriority :
      SEQ
        requested := FALSE
        oldPriority := priority
        PAR
          ToProcessor[oldPriority] ! RunRequest
          INT interaction :
          SEQ
            FromProcessor ? interaction
            IF
              interaction = AckRun
                SKIP
              interaction = ResetRequest
                FromProcessor ? priority
    TRUE
      SKIP
  ready := FALSE

...  inform processor that link is ready

INT oldPriority :
SEQ
  enabled := FALSE
  oldPriority := priority
  PAR
    ToProcessor[oldPriority] ! ReadyRequest
    INT interaction :
    SEQ
      FromProcessor ? interaction
      IF
        interaction = AckReady
          SKIP
        interaction = StatusEnquiry
          FromProcessor ? priority
        interaction = ResetRequest
          SEQ
            FromProcessor ? priority
            ready     := FALSE
```

1.7 Initialisation

More information on the subject of initialisation is available in the Initialisation, Booting, Analysing and Checking section.

The following registers and special location are not set when the machine powers on or is reset.

```
ClockReg[0]
ClockReg[1]
TPtrLoc[0]
TPtrLoc[1]
TNextReg[0]
TNextReg[1]
FptrReg[0]
BptrReg[0]
FptrReg[1]
BptrReg[1]
msb of the StatusReg (ie the errorflag)
bit 7 of the StatusReg (ie the HaltOnErrorFlag)
```

The ClockRegs do not increment after a power-on, reset or analyse until a store timer instruction has been executed. The states of the other registers are set as below:

```
Areg        = IptrReg
Breg        = WdescReg
Oreg        = 0
```

If the machine is booting from external memory then

```
WdescReg    = MemStart BITOR 1
IptrReg     = MaxInt - 1
Creg        = ANY
```

If the machine is booting from a link-channel then

```
WdescReg    = (first word after boot program) BITOR 1
IptrReg     = MemStart
Creg        = pointer to boot channel
```

1.8 Processor operation

The processor performs a sequence of actions. Each action may be (i) to execute an instruction (or part of an instruction) on behalf of the current process, (ii) to act on a request by a link-channel, or, (iii) to deal with a timer which has become ready. An action which is performed on behalf of a high priority process, on behalf of a link-channel operating at high priority or on behalf of the high-priority timer is called a "high priority action". A "low priority action" is similarly defined.

The actions which may occur for the currently executing process are the execution of the procedures (defined below) **StartNextProcess**, **InsertMiddleStep**, **BlockMoveMiddleStep**, **DeleteMiddleStep** or the fetching, decoding and execution of an instruction.

The action which may be performed by the processor on behalf of a link-channel is the execution of the procedure **HandleChannelRequest**. The action which may be performed by the processor of behalf of the timer is the execution of the procedure **HandleTimerRequest**.

When the processor has completed one action it will choose its next action as follows (this is defined more precisely in the program given below):

> The processor will execute the procedure **StartNextProcess** if the GotoSNPbit of the **StatusReg** is set, otherwise it will perform a high priority action if there is one that can be performed. Otherwise it will perform a low priority action if there is one which can be performed. Otherwise it will wait until there is a request from a timer or a link-channel.
>
> The processor selects an action at a particular priority according to the following rules. The processor will execute **DeleteMiddleStep** if the TimeDelBit of the **StatusReg** is set. Otherwise it will execute **InsertMiddleStep** if the TimeInsBit of the **StatusReg** is set. Otherwise the processor will execute the procedure **BlockMoveMiddleStep** if the MoveBit of the **StatusReg** is set. Otherwise it will handle any channel request. Otherwise it will handle any timer request. Otherwise the processor will fetch, decode an execute an instruction.

In the following description the procedures **Primary** and **Secondary** decode and execute primary and secondary instructions.

```
WHILE active
  VAL INT interruptable IS GotoSNPBit \/ (IOBit \/ (MoveBit \/
                              (TimeInsBit \/ TimeDelBit))) :
  SEQ
    -- completed indicates if current instruction has terminated
    completed := (StatusReg /\ interruptable) = 0
    validProcess := Wptr <> NotProcess.p

    PRI ALT
      (StatusReg /\ GotoSNPBit) <> 0 & SKIP
        StartNextProcess ()

      (Priority = 0) AND (NOT (TNextReg[0] AFTER ClockReg[0])) AND
                                                  completed & SKIP
          HandleTimerRequest (0)

      ALT hc = 0 FOR LinkChans
        (Priority = 0) AND completed & FromChan[hc][0] ? token
          HandleChannelRequest (token, hc)

      (Priority = 1) AND
                      (NOT (TNextReg[0] AFTER ClockReg[0])) & SKIP
        HandleTimerRequest (0)

      ALT hc = 0 FOR LinkChans
        (Priority = 1) & FromChan[hc][0] ? token
          HandleChannelRequest (token, hc)

      (Priority = 1) AND (NOT (TNextReg[1] AFTER ClockReg[1]))AND
                                            completed & SKIP
        HandleTimerRequest (1)

      ALT hc = 0 FOR LinkChans
        (Priority = 1) AND completed & FromChan[hc][1] ? token
          HandleChannelRequest (token, hc)

      validProcess & SKIP
        IF
          (StatusReg /\ TimeDelBit) <> 0
            DeleteMiddleStep (Breg, Creg)
          (StatusReg /\ TimeInsBit) <> 0
            InsertMiddleStep (Areg, Breg, Creg)
          (StatusReg /\ MoveBit)    <> 0
            BlockMoveMiddleStep (Creg, Breg, Areg)
          TRUE
            SEQ
              BuildNextInstruction (IptrReg, Oreg, code)
              IF
                code <> f.opr
                  Primary (code)
                code  = f.opr
                  Secondary (Oreg)
              Oreg := 0
```

Prioritised scheduling

The execution of a low priority process can be interrupted when a high priority process becomes runnable as defined above. In particular certain instructions are interruptable:

move message // input message // output message //

timer alt wait // timer input //

disable timer

When a low priority process is interrupted by a high priority process certain of the processor registers are written to the transputer's memory, freeing those registers for use by the high priority process. When there are no more high priority processes to be executed the registers are restored and execution of the low priority process recomences.

The following procedures are used to save and restore registers when an interrupt occurs:

```
PROC SaveRegisters(VAL BOOL SaveEreg)
  -- Save processor registers for interrupt
  SEQ
    WIndexWord(SaveBase, WdescIntSave, WdescReg)
    IF
      WdescReg <> (NotProcess.p BITOR 1)  -- Low Priority
        SEQ
          WIndexWord(SaveBase, IptrIntSave, IptrReg)
          WIndexWord(SaveBase, AregIntSave, Areg)
          WIndexWord(SaveBase, BregIntSave, Breg)
          WIndexWord(SaveBase, CregIntSave, Creg)
          WIndexWord(SaveBase, STATUSIntSave, StatusReg)
      TRUE
        SKIP
    IF
      SaveEreg
        WIndexWord(SaveBase, EregIntSave, Ereg)
      TRUE
        SKIP
:
```

```
PROC RestoreRegisters()
  -- Restore processor registers after interrupt
  SEQ
    INT temp :
    SEQ
      RIndexWord(SaveBase, WdescIntSave, temp)
      UpdateWdescReg(temp)
    IF
      WdescReg <> (NotProcess.p BITOR 1)  -- Low Priority
        SEQ
          RIndexWord(SaveBase, IptrIntSave, IptrReg)
          RIndexWord(SaveBase, AregIntSave, Areg)
          RIndexWord(SaveBase, BregIntSave, Breg)
          RIndexWord(SaveBase, CregIntSave, Creg)
          RIndexWord(SaveBase, STATUSIntSave, StatusReg)
      TRUE
        SKIP
    IF
      (StatusReg BITAND MoveBit) <> 0
        RIndexWord(SaveBase, EregIntSave, Ereg)
      TRUE
        SKIP
:
```

Action performed by processor when timer becomes ready

```
PROC HandleTimerRequest(VAL INT QueueId)
  INT frontProcess :
  SEQ
    TEnabled[QueueId] := FALSE
    RIndexWord(TptrLoc[QueueId], 0, frontProcess)
    SEQ
      INT secondProcess : --   update queue
      SEQ
        RIndexWord(frontProcess, TLink.s, secondProcess)
        WIndexWord(frontProcess, TLink.s, TimeSet.p)
        WIndexWord(TptrLoc[QueueId], 0, secondProcess)
        IF
          secondProcess = NotProcess.p
            SKIP
          TRUE
            SEQ
              RIndexWord(secondProcess, Time.s, TNextReg[QueueId])
              TEnabled[QueueId] := TRUE
      INT status : --   schedule process as appropriate
      SEQ
        RIndexWord(frontProcess, Pointer.s, status)
        IF
          status = Ready.p
            SKIP
          status = Waiting.p
            SEQ
              WIndexWord(frontProcess, Pointer.s, Ready.p)
              Run(frontProcess BITOR QueueId)
:
```

Action performed by processor as result of link-channel request

```
PROC HandleChannelRequest(VAL INT Request, hc)
  -- handles a request from a channel to the processor
  -- hc is index of hard channel in(occam) channel array
  IF
    Request = RunRequest
      INT channelContent :
      SEQ
        ToChan[hc] ! AckRun
        RIndexWord(PortBase, hc, channelContent)
        IF
          channelContent = NotProcess.p  -- after Reset
            SKIP
          TRUE
            SEQ
              WIndexWord(PortBase, hc, NotProcess.p)
              Run(channelContent)
    Request = ReadyRequest
      INT channelContent, procPtr, status :
      SEQ
        -- Needed to make the cancellable ReadyRequest work
        ToChan[hc] ! AckReady
        RIndexWord(PortBase, hc, channelContent)
        procPtr := channelContent BITAND (-2)
        RIndexWord(procPtr, Pointer.s, status)
        IF
          status = Enabling.p
            WIndexWord(procPtr, Pointer.s, Ready.p)
          status = Ready.p
            SKIP
          status = Waiting.p
            SEQ
              WIndexWord(procPtr, Pointer.s, Ready.p)
              Run(channelContent)
:
```

1.9 Clocks and timeslicing

The processor contains two clock registers, one for each priority. These registers start incrementing after the processor has been reset or analysed only once a *store timer* instruction has been executed. In the occam description of the processor in this document it is assumed that clock registers either are or are not incrementing as appropriate. In the description of the *store timer* instruction the call on the procedure **StartTimer** indicates that the clock registers should start incrementing.

The high priority clock register increments every 1uS, the low priority clock increments every 64uS.

The processor will timeslice low priority processes when the clock registers are incrementing. The mechanism works by checking, during the execution of the *jump* and *loop end* instructions whether the process has been executing for more than a timeslice period, if it has then the following process is executed

```
SEQ
  WindexWord(Wptr, Iptr.s, Iptr)
  Run(WdescReg)
  StatusReg := StatusReg \/ GotoSNPbit
```

The performance of the check and the (possible) subsequent execution of the above process is indicated in the desription of the *jump* and *loop end* instructions by the calling of the procedure **TimeSlice**.

1.10 Procedures used in the description of the instruction set

Procedures related to scheduling

```
PROC Enqueue(VAL INT ProcPtr, INT Fptr, Bptr)
  -- add a process to a scheduling list
  SEQ
    IF
      Fptr = NotProcess.p
        Fptr := ProcPtr
      TRUE
        WIndexWord(Bptr, Link.s, ProcPtr)
    Bptr := ProcPtr
:

PROC Dequeue(VAL INT Level)
  -- Take a process from a scheduling list
  SEQ
    UpdateWdescReg(FptrReg[Level] BITOR Level)
    IF
      FptrReg[Level] = BptrReg[Level]
        FptrReg[Level] := NotProcess.p
      TRUE
        RIndexWord(FptrReg[Level], Link.s, FptrReg[Level])
:

PROC ActivateProcess()
  -- Starts a process executing
  SEQ
    Oreg := 0
    RIndexWord(Wptr, Iptr.s, IptrReg)
:
```

```
PROC StartNextProcess()
  -- This starts execution of the next runnable process (if one exists).
  SEQ
    StatusReg := StatusReg BITAND (BITNOT GotoSNPBit)
    IF
      Priority = 0
        IF
          FptrReg[0] <> NotProcess.p
            SEQ
              Dequeue(0)
              ActivateProcess()
          TRUE
            SEQ -- no further high priority processes
              RestoreRegisters()
              IF
                -- no interrupted process
                (Wptr = NotProcess.p) AND
                               (FptrReg[1] <> NotProcess.p)
                  SEQ
                    Dequeue(1)
                    ActivateProcess()
                -- no low priority processes at all
                (Wptr = NotProcess.p) -- no processes
                  SKIP
                -- interrupted process was doing block move
                (StatusReg BITAND MoveBit) <> 0
                  BlockMoveFirstStep(Creg, Breg, Areg)
                -- continue with block move
                TRUE
                  SKIP
      Priority = 1
        IF
          FptrReg[1] <> NotProcess.p
            SEQ
              Dequeue(1)
              ActivateProcess()
          TRUE
            UpdateWdescReg(NotProcess.p BITOR 1)
:

PROC Wait()
  SEQ
    WIndexWord(Wptr, State.s, Waiting.p)
    WIndexWord(Wptr, Iptr.s,  IptrReg)
    StatusReg := StatusReg BITOR GotoSNPBit
:
```

```
PROC Run(VAL INT ProcDesc)
  -- Schedule a process
  INT procPriority :
  INT procPtr :
  SEQ
    procPriority := ProcDesc BITAND   1
    procPtr      := ProcDesc BITAND (-2)
    IF
      Priority = 0   -- Machine at high priority; queue process
        Enqueue(procPtr, FptrReg[procPriority], BptrReg[procPriority])
      Priority = 1   -- Machine at low priority
        IF
          procPriority = 0 -- High priority process; execute it
            SEQ
              SaveRegisters( (StatusReg BITAND MoveBit) <> 0 )
              UpdateWdescReg(ProcDesc)
              StatusReg := StatusReg BITAND
                               (ErrorFlag BITOR HaltOnErrorBit)
              ActivateProcess()
          procPriority = 1 -- Low priority process; queue it
            IF
              Wptr = NotProcess.p
                SEQ
                  UpdateWdescReg(ProcDesc)
                  ActivateProcess()
              TRUE
                Enqueue(procPtr, FptrReg[1], BptrReg[1])
:
```

Procedures concerned with Timer queue manipulation

The four procedures **InsertFirstStep**, **InsertMiddleStep**, **InsertFinalStep** and **InsertTest** combine to cause the current process to be inserted into the timer queue. This will happen due to execution of either the *timer input* or *timer alternative wait* instructions.

```
PROC InsertFinalStep(INT time, previous, subsequent)
  SEQ
    -- Enqueue new timer process
    WIndexWord(previous, 0,       Wptr)
    WIndexWord(Wptr,     TLink.s, subsequent)
    WIndexWord(Wptr,     Iptr.s,  IptrReg)

    -- Ensure the earliest time is in TNextReg
    RIndexWord(TimerBase, Priority, previous)
    RIndexWord(previous,  Time.s,   TNextReg[Priority])
    TEnabled[Priority] := TRUE

    -- Finished insertion, start next process
    StatusReg := StatusReg BITAND (BITNOT TimeInsBit)
    StatusReg := StatusReg BITOR GotoSNPBit
:

PROC InsertTest(INT time, previous, subsequent)
  -- Used by Insert Middle and First Steps
  SEQ
    RIndexWord(previous, 0, subsequent)
    IF
      subsequent = NotProcess.p
        InsertFinalStep(time, previous, subsequent)
      subsequent <> NotProcess.p
        INT subsequentTime :
        BOOL  laterFlag :
        SEQ
          RIndexWord(subsequent, Time.s, subsequentTime)
          Later(time, subsequentTime, laterFlag)
          IF
            laterFlag
              SKIP
            TRUE
              InsertFinalStep(time, previous, subsequent)
:

PROC InsertMiddleStep(INT time, previous, subsequent)
  -- Test for Insertion before next process on timer queue
  SEQ
    AtWord(subsequent, TLink.s, previous)
    InsertTest(time, previous, subsequent)
:
```

```
PROC InsertFirstStep(INT time, previous, subsequent)
  -- Areg is time
  -- Breg is previous
  -- Creg is subsequent
  -- "previous" points at the location to be updated if the current
  -- process is to be inserted before the process pointed to by
  -- "subsequent".
  SEQ
    -- Start insertion, set local registers
    StatusReg := StatusReg BITOR TimeInsBit
    WIndexWord(Wptr, State.s, Waiting.p)
    WIndexWord(Wptr, Time.s, time)

    -- Test for Insertion before first process on timer queue
    AtWord(TimerBase, Priority, previous)
    InsertTest(time, previous, subsequent)
:
```

The four procedures **DeleteFirstStep**, **DeleteMiddleStep**, **DeleteFinalStep** and **DeleteTest** combine to cause the current process to be deleted from the timer queue. This will happen due to execution the *disable timer* instruction.

```
PROC DeleteFinalStep(INT previous, subsequent)
  SEQ
    -- Delete the current process from the timer queue
    RIndexWord(Wptr, TLink.s, subsequent)
    WIndexWord(previous, 0,   subsequent)
    WIndexWord(Wptr, TLink.s, TimeNotSet.p)

    -- Ensure the earliest time is stored in TNextReg
    RIndexWord(TptrLoc[Priority], 0, previous)
    IF
      previous =  NotProcess.p
        SKIP
      previous <> NotProcess.p
        SEQ
          RIndexWord(previous, Time.s, TNextReg[Priority])
          TEnabled[INT Priority] := TRUE

    -- Finish Deletion
    StatusReg := StatusReg BITAND (BITNOT TimeDelBit)
:

PROC DeleteTest(INT previous, subsequent)
  -- Used by Delete First and Middle Steps
  SEQ
    RIndexWord(previous, 0, subsequent)
    IF
      subsequent = Wptr
        DeleteFinalStep(previous, subsequent)
      TRUE
        SKIP
:

PROC DeleteMiddleStep(INT previous, subsequent)
  -- Test for Deletion before next process on timer queue
  SEQ
    AtWord(subsequent, TLink.s, previous)
    DeleteTest(previous, subsequent)
:

PROC DeleteFirstStep(INT previous, subsequent)
  SEQ
    -- Start deletion, set TEnabled to FALSE (pending completion)
    StatusReg := StatusReg BITOR TimeDelBit
    TEnabled[Priority] := FALSE

    -- Test for deletion before first process on timer queue
    previous := TptrLoc[Priority]
    DeleteTest(previous, subsequent)
:
```

Procedure used in alternative input

```
PROC IsThisSelectedProcess()
  -- this is used by all the disable instructions
  INT disableStatus :
  SEQ
    RIndexWord(Wptr, 0, disableStatus)
    IF
      disableStatus = NoneSelected.o
        SEQ
          WIndexWord(Wptr, 0, Areg)
          Areg := MachineTRUE
      disableStatus <> NoneSelected.o
        Areg := MachineFALSE
:
```

Procedures used to implement block move

The routines **WritePartWord**, **Min**, **CalcShiftUp**, **Decode** and **Select** are used in the implementation of the block moving mechanism. The block move mechanism is initialised by execution of **BlockMoveFirstStep** (this will happen as a result of execution of a *block move* instruction, an *input message* instruction, an *output message* instruction or when the transputer restarts an interrupted block move. Once initialised the **BlockMoveMiddleStep** procedure is repeated executed until either the block move has completed or the block move is interrupted.

```
PROC WritePartWord(VAL INT Address, Word, StartByte, Length)
  -- insert bytes 'StartByte' through 'StartByte+Length-1' into
  -- the corresponding byte of the memory location 'Address'
  INT buffer, insert, keep :
  SEQ
    insert := 0
    SEQ byteIndex = StartByte FOR Length
      insert := insert BITOR (#FF << (byteIndex*8))
    keep   := BITNOT insert
    RIndexWord(Address, 0, buffer)
    buffer := (buffer BITAND keep) BITOR (Word BITAND insert)
    RestoreToRange(buffer)
    WIndexWord(Address, 0, buffer)
:

PROC Min(VAL INT Arg1, Arg2, INT result)
  IF
    Arg1 < Arg2
      result := Arg1
    TRUE
      result := Arg2
:

PROC CalcShiftUp(VAL INT SB, DB, INT shift)
  -- Calculate the Byte shift for the source to match the destination.
  SEQ
    shift := (DB - SB) REM BytesPerWord
    IF
      shift < 0
        shift := shift + BytesPerWord
      TRUE
        SKIP
:
```

```
PROC Decode(VAL INT Dest, Source, INT DB, SB)
  -- Extract Byte-select component of source and destination addresses
  SEQ
    DB := Dest   BITAND BselMask
    SB := Source BITAND BselMask
:

PROC Select(VAL INT P, C, ShiftUp, INT S)
  -- Forms a new word,
  --  with the ShiftUp-most-significant bytes from P at the
  --  least significant end, and the (BitsInWord/8) minus ShiftUp-
  --  least-significant bytes from C at the most significant end.
  -- Inserts 1's otherwise.
  INT lowWord, highWord :
  VAL ShiftUpBits IS ShiftUp * 8 :
  VAL Complement  IS BitsInWord - ShiftUpBits :
  SEQ
    lowWord  := (P >> Complement) BITOR ((-1) << ShiftUpBits)
    highWord :=  C                BITOR ((-1) << Complement)
    highWord := (highWord << ShiftUpBits) BITOR
      (BITNOT ((-1) << ShiftUpBits) )
    S := lowWord BITAND highWord
:
```

```
PROC BlockMoveFinalStep()
  -- NB Clear Flags BEFORE running Ereg !
  -- Run Ereg if IOBit set, clear IOBit and MoveBit
  IF
    (StatusReg BITAND IOBit) <> 0
      SEQ
        StatusReg := (StatusReg         BITAND
                      (BITNOT MoveBit)) BITAND
                      (BITNOT IOBit)
        Run(Ereg)
    TRUE
      StatusReg := (StatusReg         BITAND
                    (BITNOT MoveBit))
:

PROC BlockMoveFirstStep(INT source, dest, length)
  INT shiftUp :
  INT bytesToRead, bytesToWrite :
  INT DB, SB :
  INT current, selected :
  IF
    length = 0
      BlockMoveFinalStep()

    length > 0
      SEQ
        StatusReg := StatusReg BITOR MoveBit
        Decode(dest, source, DB, SB)
        CalcShiftUp(SB, DB, shiftUp)
        RIndexWord(source, 0, current)
        Min((BitsInWord/8) - SB, length, bytesToRead)
        Min((BitsInWord/8) - DB, length, bytesToWrite)
        IF
          bytesToRead >= bytesToWrite
            Select(current, current, shiftUp, selected)
          bytesToRead <  bytesToWrite
            SEQ
              BMbuffer := current
              -- Must do another read before we write
              RIndexWord(source, 1, current)
              Select(BMbuffer, current, shiftUp, selected)
        -- Write
        WritePartWord(dest, selected, DB, bytesToWrite)
        -- Update pointers and buffer
        AtByte(dest, bytesToWrite, dest)
        length := length - bytesToWrite
        AtByte(source, bytesToWrite, source)
        -- Update buffer
        BMbuffer := current
:
```

```
PROC BlockMoveMiddleStep(INT source, dest, length)
  INT shiftUp :
  INT bytesToWrite :
  INT DB, SB :
  INT current, selected :
  IF
    length = 0
      BlockMoveFinalStep()

    length > 0
      SEQ
        -- Read word
        Decode(dest, source, DB, SB)
        CalcShiftUp(SB, DB, shiftUp)
        IF
          length > shiftUp
            -- First choose which word to read
            IF
              shiftUp = 0
                RIndexWord(source, 0, current)
              shiftUp <> 0
                RIndexWord(source, 1, current)
          TRUE
            SKIP
        -- Write appropiate section
        -- Selection can be omitted in the ShiftUp = 0 case
        Select(BMbuffer, current, shiftUp, selected)
        Min((BitsInWord/8) - DB, length, bytesToWrite)
        WritePartWord(dest, selected, DB, bytesToWrite)
        -- Update pointers and buffer
        AtByte(dest, bytesToWrite, dest)
        length := length - bytesToWrite
        AtByte(source, bytesToWrite, source)
        BMbuffer := current
:
```

Procedures used for input and output

```
PROC HandShake(VAL INT I, INT token)
  -- Required for resetting a link which might be
  -- operating at high priority
  ALT pri = 0 FOR 2
    FromChanL[I][pri] ? token
      SKIP
:

PROC SaveRegsPendingSoftIO()
  SEQ
    WIndexWord(Breg, 0, WdescReg)
    WIndexWord(Wptr, Iptr.s,     IptrReg)
    WIndexWord(Wptr, Pointer.s, Creg)
:

PROC HardChannelInputOutputAction(VAL INT portNo)
  SEQ
    WIndexWord(Breg, 0, WdescReg)
    WIndexWord(Wptr, Iptr.s,    IptrReg)
    ToChan[portNo] ! PerformIO; Priority; Creg; Areg
:

PROC ChanOffset(VAL INT reg, INT chanNum)
  -- Extract a "channel number", starting from MinInt = 0
  chanNum := (reg - MinInt) >> BselLength
:
```

```
PROC Input()
  -- Areg is count, Breg is channel, Creg is pointer.
  INT chanNum :
  SEQ
    ChanOffset(Breg, chanNum)
    IF
      chanNum >= LinkChans                  -- soft(Breg)
        INT procDesc :
        SEQ
          RIndexWord(Breg, 0, procDesc)
          IF
            procDesc = NotProcess.p         -- Not ready; wait
              SEQ
                SaveRegsPendingSoftIO()
                StatusReg := StatusReg BITOR GotoSNPBit
            procDesc <> NotProcess.p        -- Ready; transfer
              INT sourcePtr, procPtr :
              SEQ
                -- Reset channel -- NB ok to do this here
                WIndexWord(Breg, 0, NotProcess.p)
                procPtr := procDesc BITAND (-2)
                RIndexWord(procPtr, Pointer.s, sourcePtr)
                -- Set up the block move
                Ereg := procDesc
                Breg := Creg
                Creg := sourcePtr
                StatusReg := StatusReg BITOR
                                     (MoveBit BITOR IOBit)
                BlockMoveFirstStep(Creg, Breg, Areg)
                -- When completed, BlockMove will Run(Ereg)
      chanNum < LinkChans                   -- hard(Breg)
        SEQ
          HardChannelInputOutputAction(chanNum)
          StatusReg := StatusReg BITOR GotoSNPBit
:
```

```
PROC Output()
  -- Areg is count, Breg is channel, Creg is pointer.
  INT chanNum :
  SEQ
    ChanOffset(Breg, chanNum)
    IF
      chanNum >= LinkChans                  -- Internal channel
        INT procDesc :
        SEQ
          RIndexWord(Breg, 0, procDesc)
          IF
            procDesc = NotProcess.p         -- Not ready; wait
              SEQ
                SaveRegsPendingSoftIO()
                StatusReg := StatusReg BITOR GotoSNPBit
            procDesc <> NotProcess.p        -- Ready
              INT destPtr, procPtr :
              SEQ
                procPtr := procDesc BITAND (-2)
                RIndexWord(procPtr, Pointer.s, destPtr)
                IF                  -- scheduler interlock for ALT
                  destPtr = Enabling.p
                    SEQ
                      WIndexWord(procPtr, Pointer.s, Ready.p)
                      SaveRegsPendingSoftIO()
                      StatusReg := StatusReg BITOR GotoSNPBit
                  destPtr = Waiting.p
                    SEQ
                      WIndexWord(procPtr, Pointer.s, Ready.p)
                      SaveRegsPendingSoftIO()
                      StatusReg := StatusReg BITOR GotoSNPBit
                      Run(procDesc)
                  destPtr = Ready.p
                    SEQ
                      SaveRegsPendingSoftIO()
                      StatusReg := StatusReg BITOR GotoSNPBit
                  TRUE                      -- valid pointer
                    SEQ
                      -- Reset channel
                      WIndexWord(Breg, 0, NotProcess.p)
                      -- Set up registers for the block move
                      Ereg := procDesc
                      Breg := destPtr
                      StatusReg := StatusReg BITOR
                                         (MoveBit BITOR IOBit)
                      BlockMoveFirstStep(Creg, Breg, Areg)
                      -- When completed, BlockMove will Run(Ereg)
      chanNum < LinkChans                   -- link-channel
        SEQ
          HardChannelInputOutputAction(chanNum)
          StatusReg := StatusReg BITOR GotoSNPBit
:
```

Other procedures used in the instruction descriptions

```
PROC ArithmeticRightShift(VAL INT Operand, Shift, INT result)
  IF
    Operand >= 0
      result := Operand >> Shift
    Operand <  0
      SEQ
        result := BITNOT Operand
        result := result >> Shift
        result := BITNOT result
:
```

1.11 Function Set

The instructions executed by the procesor include direct functions, the prefixing functions pfix and nfix, and an indirect function opr which uses the operand register Oreg to select one of a set of operations.

The set of direct functions and operations is as follows:

Direct, Prefixing and Indirect Functions

Code	Abbreviation	Name
#07	ldl	load local
#0D	stl	store local
#01	ldlp	load local pointer
#03	ldnl	load non-local
#0E	stnl	store non-local
#05	ldnlp	load non-local pointer
#0C	eqc	equals constant
#04	ldc	load constant
#08	adc	add constant
#00	j	jump
#0A	cj	conditional jump
#09	call	call
#0B	ajw	adjust workspace
#02	pfix	prefix
#06	nfix	negative prefix
#0F	opr	operate

Operations

Code	Size	Abbreviation	Name
#00	short	rev	reverse
#20	long	ret	return
#1B	long	ldpi	load pointer to instruction
#3C	long	gajw	general adjust workspace
#06	short	gcall	general call
#42	long	mint	mimimum integer
#21	long	lend	loop end
#13	long	csub0	check subscript from 0
#4D	long	ccnt1	check count from 1
#29	long	testerr	test error false and clear
#10	long	seterr	set error
#55	long	stoperr	stop on error
#57	long	clrhalterr	clear halt-on-error
#58	long	sethalterr	set halt-on-error
#59	long	testhalterr	test halt-on-error
#02	short	bsub	byte subscript
#0A	short	wsub	word subscript
#34	long	bcnt	byte count
#3F	long	wcnt	word count
#01	short	lb	load byte
#3B	long	sb	store byte
#4A	long	move	move message
#46	long	and	and
#4B	long	or	or
#33	long	xor	exclusive or
#32	long	not	bitwise not
#41	long	shl	shift left
#40	long	shr	shift right
#05	short	add	add
#0C	short	sub	subtract
#53	long	mul	multiply
#2C	long	div	divide
#1F	long	rem	remainder
#09	short	gt	greater than
#04	short	diff	difference
#52	long	sum	sum
#08	short	prod	product
#0D	short	startp	start process
#03	short	endp	end process
#39	long	runp	run process
#15	long	stopp	stop process
#1E	long	ldpri	load current priority

Code	Size	Abbreviation	Name
#07	short	in	input message
#0B	short	out	output message
#0F	short	outword	output word
#0E	short	outbyte	output byte
#12	long	resetch	reset channel
#43	long	alt	alt start
#44	long	altwt	alt wait
#45	long	altend	alt end
#49	long	enbs	enable skip
#30	long	diss	disable skip
#48	long	enbc	enable channel
#2F	long	disc	disable channel
#22	long	ldtimer	load timer
#2B	long	tin	timer input
#4E	long	talt	timer alt start
#51	long	taltwt	timer alt wait
#47	long	enbt	enable timer
#2E	long	dist	disable timer
#3A	long	xword	extend to word
#56	long	cword	check word
#1D	long	xdble	extend to double
#4C	long	csngl	check single
#16	long	ladd	long add
#38	long	lsub	long subtract
#37	long	lsum	long sum
#4F	long	ldiff	long diff
#31	long	lmul	long multiply
#1A	long	ldiv	long divide
#36	long	lshl	long shift left
#35	long	lshr	long shift right
#19	long	norm	normalise
#2A	long	testpranal	test processor analysing

Code	Size	Abbreviation	Name
#3E	long	saveh	save high priority queue registers
#3D	long	savel	save low priority queue registers
#18	long	sthf	store high priority front pointer
#50	long	sthb	store high priority back pointer
#1C	long	stlf	store low priority front pointer
#17	long	stlb	store low priority back pointer
#54	long	sttimer	store timer
#63	long	unpacksn	unpack single length fp number
#6D	long	roundsn	round single length fp number
#6C	long	postnormsn	post-normalise correction of single length fp number
#71	long	ldinf	load single length infinity
#73	long	cflerr	check single length fp infinit
y or NaN #72	long	fmul	fractional multiply
#28	long	teststd	store to Dreg for testing
#27	long	testste	store to Ereg for testing
#26	long	teststs	store to StatusReg for testing
#25	long	testldd	load to Dreg for testing
#24	long	testlde	load to Ereg for testing
#23	long	testlds	load to StatusReg for testing
#19B	long		single step TimeOut for testing
#2D	long	testhardchan	test hard chanel stack

Direct Functions

load local

```
SEQ
  Creg := Breg
  Breg := Areg
  RIndexWord(Wptr, Oreg, Areg)
```

store local

```
SEQ
  WIndexWord(Wptr, Oreg, Areg)
  Areg := Breg
  Breg := Creg
```

load local pointer

```
SEQ
  Creg := Breg
  Breg := Areg
  AtWord(Wptr, Oreg, Areg)
```

load non-local

```
RIndexWord(Areg, Oreg, Areg)
```

store non-local

```
SEQ
  WIndexWord(Areg, Oreg, Breg)
  Areg := Creg
```

load non-local pointer

```
AtWord(Areg, Oreg, Areg)
```

equals constant

```
IF
  Areg = Oreg
    Areg := MachineTRUE
  Areg <> Oreg
    Areg := MachineFALSE
```

load constant

```
SEQ
  Creg := Breg
  Breg := Areg
  Areg := Oreg
```

add constant

```
SEQ
  Areg := Areg + Oreg
  OverflowCheck(Areg)
```

jump

```
SEQ
  AtByte(IptrReg, Oreg, IptrReg)
  TimeSlice()
```

conditional jump

```
IF
  Areg = 0
    AtByte(IptrReg, Oreg, IptrReg)
  Areg <> 0
    SEQ
      Areg := Breg
      Breg := Creg
```

call

```
SEQ
  WIndexWord(Wptr, -1, Creg)
  WIndexWord(Wptr, -2, Breg)
  WIndexWord(Wptr, -3, Areg)
  WIndexWord(Wptr, -4, IptrReg)
  Areg := IptrReg
  INT temp :
  SEQ
    AtWord(Wptr, -4, temp)
    UpDateWdescReg(temp BITOR Priority)
  AtByte(IptrReg, Oreg, IptrReg)
```

adjust workspace

```
INT temp :
SEQ
  AtWord(Wptr, Oreg, temp)
  UpDateWdescReg(temp BITOR Priority)
```

Register Manipulation Etc

reverse

```
SEQ
  Oreg := Areg
  Areg := Breg
  Breg := Oreg
```

return

```
SEQ
  RIndexWord(Wptr, 0, IptrReg)
  INT temp :
  SEQ
    AtWord(Wptr, 4, temp)
    UpDateWdescReg(temp BITOR Priority)
```

load pointer to instruction

```
AtByte(IptrReg, Areg, Areg)
```

general adjust workspace

```
INT temp:
SEQ
  temp := Wptr
  UpDateWdescReg(Areg BITOR Priority)
  Areg := temp
```

general call

```
INT temp:
SEQ
  temp := IptrReg
  IptrReg := Areg
  Areg := temp
```

minimum integer

```
SEQ
  Creg := Breg
  Breg := Areg
  Areg := MinInt
```

loop end

```
SEQ
  RIndexWord(Breg, 1, Creg)
  Creg := Creg - 1
  WIndexWord(Breg, 1, Creg)
  IF
    Creg > 0
      SEQ
        RIndexWord(Breg, 0, Creg)
        Creg := Creg + 1
        WIndexWord(Breg, 0, Creg)
        AtByte(IptrReg, -Areg, IptrReg)
    Creg <= 0
      SKIP
```

TimeSlice()

Checking

check subscript from 0

```
SEQ
  UnSign(Areg)
  UnSign(Breg)
  IF
    Breg >= Areg -- unsigned compare
      SetErrorFlag()
    TRUE
      SKIP
  Sign(Breg)
  Areg := Breg
  Breg := Creg
```

check count from 1

```
SEQ
  UnSign(Areg)
  UnSign(Breg)
  IF
    (Breg = 0) OR (Breg > Areg) -- unsigned comparison
      SetErrorFlag()
    TRUE
      SKIP
  Sign(Breg)
  Areg := Breg
  Breg := Creg
```

test error false and clear

```
BOOL errorSet :
SEQ
  Creg := Breg
  Breg := Areg
  ReadErrorFlag(errorSet)
  IF
    errorSet
      Areg := MachineFALSE
    NOT errorSet
      Areg := MachineTRUE
  ClearErrorFlag()
```

set error

```
SetErrorFlag()
```

stop on error

```
BOOL errorSet :
SEQ
  ReadErrorFlag(errorSet)
  IF
    errorSet
      SEQ
        WIndexWord(Wptr, Iptr.s, IptrReg)
        StatusReg := StatusReg BITOR GotoSNPBit
    NOT errorSet
      SKIP
```

clear halt-on-error

```
ClearHaltOnErrorFlag()
```

set halt-on-error

```
SetHaltOnErrorFlag()
```

test halt-on-error

```
BOOL flagSet :
SEQ
  Creg := Breg
  Breg := Areg
  ReadHaltOnErrorFlag(flagSet)
  IF
    flagSet
      Areg := MachineTRUE
    NOT flagSet
      Areg := MachineFALSE
```

Addressing

byte subscript

```
SEQ
  AtByte(Areg, Breg, Areg)
  Breg := Creg
```

word subscript

```
SEQ
  AtWord(Areg, Breg, Areg)
  Breg := Creg
```

byte count

```
Areg := Areg * (BitsInWord/8)
```

word count

```
SEQ
  Creg := Breg
  Breg := Areg BITAND BselMask
  ArithmeticRightShift(Areg, BselLength, Areg)
```

Data Access and Move

load byte

```
RIndexByte(Areg, 0, Areg)
```

store byte

```
SEQ
  WIndexByte(Areg, 0, Breg)
  Areg := Creg
```

move message

```
BlockMoveFirstStep(Creg, Breg, Areg)
```

Logic and Bits

and

```
SEQ
  Areg := Areg BITAND Breg
  Breg := Creg
```

or

```
SEQ
  Areg := Breg BITOR Areg
  Breg := Creg
```

xor

```
SEQ
  Areg := Breg >< Areg
  Breg := Creg
```

not

```
Areg := Areg >< (-1)
```

shift left

```
SEQ
  Unsign(Areg)
  IF
    Areg <= BitsInWord
      SEQ
        Unsign(Breg)
        Areg := (Breg << Areg) REM Range
        Sign(Areg)
  Breg := Creg
```

shift right

```
SEQ
  UnSign(Breg)
  IF
    Areg <= BitsInWord
      Areg := Breg >> Areg
  Sign(Areg)
  Breg := Creg
```

Basic Arithmetic

add

```
SEQ
  Areg := (Breg + Areg)
  OverflowCheck(Areg)
  Breg := Creg
```

subtract

```
SEQ
  Areg := (Breg - Areg)
  OverflowCheck(Areg)
  Breg := Creg
```

multiply

```
-- Signed multiply, Areg := Areg * Breg MOD Range.
-- OverflowCheck now handles ANY signed integer !
SEQ
  Areg := Breg * Areg
  OverflowCheck(Areg)
  Breg := Creg
```

divide

```
SEQ
  IF
    ((Breg = MinInt) AND (Areg = (-1))) OR (Areg = 0)
      SetErrorFlag()
    TRUE
      Areg := Breg / Areg
  Breg := Creg
```

remainder

```
SEQ
  IF
    ((Breg = MinInt) AND (Areg = (-1))) OR (Areg = 0)
      SetErrorFlag()
    TRUE
      Areg := Breg REM Areg
  Breg := Creg
```

Comparison and modulo arithmetic

greater than

```
SEQ
  IF
    Breg > Areg
      Areg := MachineTRUE
    Breg <= Areg
      Areg := MachineFALSE
  Breg := Creg
```

difference

```
SEQ
  Areg := (Breg - Areg)
  RestoreToRange(Areg)
  Breg := Creg
```

sum

```
SEQ
  Areg := Breg + Areg
  RestoreToRange(Areg)
  Breg := Creg
```

product

```
SEQ                          -- quick unchecked multiply
  UnSign(Areg)               -- short operand in Areg
  UnSign(Breg)
  Areg := Breg * Areg
  Areg := Areg REM Range
  Sign(Areg)
  Breg := Creg
```

Scheduling

start process

```
INT temp :
SEQ
  AtByte(IptrReg, Breg, temp)
  WIndexWord(Areg, Iptr.s, temp)
  Run(Areg BITOR Priority)
```

end process

```
INT temp :
SEQ
  RIndexWord(Areg, 1, temp)
  IF
    temp = 1
      SEQ
        RIndexWord(Areg, 0, IptrReg)
        UpDateWdescReg(Areg BITOR Priority)
    temp <> 1
      SEQ
        WIndexWord(Areg, 1, temp-1)
        StatusReg := StatusReg BITOR GotoSNPBit
```

run process

```
Run(Areg)
```

stop process

```
SEQ
  WIndexWord(Wptr, Iptr.s, IptrReg)
  StatusReg := StatusReg BITOR GotoSNPBit
```

load current priority

```
SEQ
  Creg := Breg
  Breg := Areg
  Areg := Priority
```

Communication

input message

```
Input()
```

output message

```
Output()
```

output word

```
SEQ
  WIndexWord(Wptr, 0, Areg)
  Areg := BytesPerWord
  Creg := Wptr
  Output()
```

output byte

```
SEQ
  WIndexWord(Wptr, 0, Areg)
  Areg :=    1
  Creg := Wptr
  Output()
```

reset channel

```
INT temp :
INT chanNum :
SEQ                          -- Channel ID in Areg
  RIndexWord(Areg, 0, temp)
  WIndexWord(Areg, 0, NotProcess.p)
  ChanOffset(Areg, chanNum)
  IF
    chanNum < LinkChans   -- Hard Channel
      INT token :
      PAR
        ToChan[chanNum] ! ResetRequest; Priority
        HandShake(chanNum, token)
    TRUE
      SKIP          -- no other action needed for soft channel
  Areg := temp      -- old process pointer
```

Timer Input

load timer

```
SEQ
  Creg := Breg
  Breg := Areg
  Areg := ClockReg[Priority]
```

timer input

```
BOOL laterFlag :
SEQ
  Later(ClockReg[Priority], Areg, laterFlag)
  IF
    laterFlag
      SKIP
    TRUE
      SEQ
        Areg := Areg + 1
        RestoreToRange(Areg)
        InsertFirstStep(Areg, Breg, Creg)
```

Alternative Input

alt start

```
WIndexWord(Wptr, State.s, Enabling.p)
```

alt wait

```
SEQ
  -- set up "NoneSelected.o" in local 0 to signify
  -- that the no ready process has been selected
  WIndexWord(Wptr, 0, NoneSelected.o)
  -- Is any channel or skip guard ready?
  RIndexWord(Wptr, State.s, Areg)
  IF
    Areg = Ready.p
      SKIP
    TRUE
      Wait()
```

alt end

```
INT temp :
SEQ
  RIndexWord(Wptr, 0, temp)
  AtByte(IptrReg, temp, IptrReg)
```

Skip Guards

enable skip

```
IF
  Areg <> MachineFALSE
    WIndexWord(Wptr, State.s, Ready.p)
  TRUE
    SKIP
```

disable skip

```
SEQ
  IF
    Breg <> MachineFALSE
      IsThisSelectedProcess()
    TRUE
      Areg := MachineFALSE
  Breg := Creg
```

Channel Guards

enable channel

```
SEQ
  IF
    Areg <> MachineFALSE
      INT chanNum :
      SEQ
        ChanOffset(Breg, chanNum)
        IF
          chanNum >= LinkChans            --  internal channel
            INT temp :
            SEQ
              RIndexWord(Breg, 0, temp)
              IF
                temp = NotProcess.p
                  WIndexWord(Breg, 0, WdescReg)
                temp = WdescReg
                  SKIP
                TRUE
                  WIndexWord(Wptr, State.s, Ready.p)
          chanNum <  LinkChans            --  link-channel
            INT token :
            SEQ                           -- is channel ready ?
              PAR
                ToChan[chanNum] ! StatusEnquiry; Priority
                FromChan[chanNum][Priority] ? token
              IF
                token = ReadyRequest
                  WIndexWord(Wptr, State.s, Ready.p)
                token = ReadyFALSE
                  SEQ
                    ToChan[chanNum] ! Enable; Priority
                    WIndexWord(Breg, 0, WdescReg)
    TRUE
      SKIP
  Breg := Creg
```

disable channel

```
IF
  Breg <> MachineFALSE
    INT chanNum :
    SEQ
      ChanOffset(Creg, chanNum)
      IF
        chanNum >= LinkChans          -- Internal channel
          SEQ
            RIndexWord(Creg, 0, Breg)
            IF
              Breg = NotProcess.p
                Areg := MachineFALSE
              Breg = WdescReg
                SEQ
                  WIndexWord(Creg, 0, NotProcess.p)
                  Areg := MachineFALSE
              TRUE
                IsThisSelectedProcess()
        chanNum <  LinkChans          -- Hard Channel
          INT token :
          SEQ
            WIndexWord(Creg, 0, NotProcess.p)
            -- Ask if channel is ready and hence switch off channel
            PAR
              ToChan[chanNum] ! StatusEnquiry; Priority
              FromChan[chanNum][Priority] ? token
            IF
              token = ReadyRequest
                IsThisSelectedProcess()
              token = ReadyFALSE
                Areg := MachineFALSE
  TRUE
    Areg := MachineFALSE
```

Alternative Timer Input

timer alt start

```
SEQ
  WIndexWord(Wptr, TLink.s, TimeNotSet.p)
  WIndexWord(Wptr, State.s, Enabling.p)
```

timer alt wait

```
SEQ
  -- NoneSelected.o in local 0 signifies that
  -- no process has yet been selected
  WIndexWord(Wptr, 0, NoneSelected.o)
  RIndexWord(Wptr, State.s,  Creg)
  IF
    Creg = Ready.p            -- a channel is ready
      WIndexWord(Wptr, Time.s, ClockReg[Priority])
    TRUE
      SEQ
        RIndexWord(Wptr, TLink.s, Breg)
        IF
          Breg = TimeNotSet.p
            Wait()            -- all timer guards FALSE
          Breg = TimeSet.p
            -- Either a timer guard is ready, or wait
            BOOL laterFlag :
            SEQ
              RIndexWord(Wptr, Time.s, Areg)
              Later(ClockReg[Priority], Areg, laterFlag)
              IF
                laterFlag
                  --  clock makes process ready
                  SEQ
                    WIndexWord(Wptr, State.s, Ready.p)
                    WIndexWord(Wptr, Time.s, ClockReg[Priority])

                TRUE
                  --  clock does not make process ready
                  SEQ
                    -- set Areg to time AT which process is ready
                    Areg := Areg + 1
                    RestoreToRange(Areg)
                    InsertFirstStep(Areg, Breg, Creg)
```

Timer Guards

enable timer

```
SEQ
  IF
    Areg <> MachineFALSE
      INT temp :
      SEQ
        RIndexWord(Wptr, TLink.s, temp)
        IF
          temp = TimeNotSet.p
            -- This is first timer guard encountered
            SEQ
              WIndexWord(Wptr, TLink.s, TimeSet.p)
              WIndexWord(Wptr, Time.s,  Breg)

          temp = TimeSet.p
            -- Update earliest time if this guard is earlier
            BOOL  laterFlag :
            SEQ
              RIndexWord(Wptr, Time.s, temp)
              Later(temp, Breg, laterFlag)
              IF
                laterFlag
                  WIndexWord(Wptr, Time.s, Breg)
                TRUE
                  SKIP

    Areg = MachineFALSE
      SKIP
  Breg := Creg
```

disable timer

```
IF
  Breg <> MachineFALSE
    SEQ
      RIndexWord(Wptr, TLink.s, Oreg)
      IF
        Oreg = TimeNotSet.p
          Areg := MachineFALSE
        Oreg = TimeSet.p
          -- See if this timer guard is ready
          BOOL laterFlag :
          SEQ
            RIndexWord(Wptr, Time.s, Oreg)
            Later(Oreg, Creg, laterFlag)
            IF
              laterFlag
                IsThisSelectedProcess()
              TRUE
                Areg := MachineFALSE

        TRUE
          SEQ
            -- process must be removed from timer queue
            DeleteFirstStep(Breg, Creg)
            Areg := MachineFALSE

  Breg = MachineFALSE
    Areg := MachineFALSE
```

Partword arithmetic

extend to word

```
SEQ
  Unsign(Areg)
  IF
    (Breg < Areg)
      Areg := Breg
    TRUE
      Areg := Breg - (2*Areg)
  Breg := Creg
```

check word

```
SEQ
  Unsign(Areg)
  IF
    (Breg >= Areg) OR (Breg < -Areg)
      SetErrorFlag()
    TRUE
      SKIP
  Areg := Breg
  Breg := Creg
```

Long arithmetic

extend to double

```
SEQ
  Creg := Breg
  IF
    Areg < 0
      Breg := -1
    Areg >=0
      Breg := 0
```

check single

```
SEQ
  IF
    ((Areg < 0) AND (Breg <> (-1))) OR
     ((Areg >= 0) AND (Breg <> 0 ))
      SetErrorFlag()
    TRUE
      SKIP
  Breg := Creg
```

long add

```
SEQ
  Areg := (Breg + Areg) + (Creg BITAND 1)
  OverflowCheck(Areg)
```

long subtract

```
SEQ
  Areg := (Breg - Areg) - (Creg BITAND 1)
  OverflowCheck(Areg)
```

long sum

```
SEQ
  UnSign(Areg)
  UnSign(Breg)
  Areg := (Breg + Areg) + (Creg BITAND 1)
  IF
    (Areg > Range)
      SEQ
        Breg := 1
        Areg := Areg - Range
    TRUE
      Breg := 0
  Sign(Areg)
```

long diff

```
SEQ
  UnSign(Areg)
  UnSign(Breg)
  Areg := (Breg - Areg) - (Creg BITAND 1)
  IF
    Areg >= 0
      Breg := 0
    Areg < 0
      SEQ
        Areg := Areg + Range
        Breg := 1
  Sign(Areg)
```

long multiply

```
SEQ
  UnSign(Areg)
  UnSign(Breg)
  UnSign(Creg)
  Areg := (Breg * Areg) + Creg
  Breg := Areg / Range
  Areg := Areg REM Range
  Sign(Areg)
  Sign(Breg)
```

long divide

```
SEQ
  UnSign(Areg)
  UnSign(Breg)
  UnSign(Creg)
  IF
    Creg >= Areg
      SetErrorFlag()
    Creg <  Areg
      INT temp :
      SEQ
        temp := (Creg << BitsInWord) + Breg
        Breg := temp REM Areg
        Areg := temp / Areg
        Sign(Areg)
        Sign(Breg)
```

normalise

```
IF
  (Breg = 0) AND (Areg = 0)
    Creg := 2*BitsInWord
  TRUE
    VAL MsbOfDoubleWord IS 1 << ((2*BitsInWord)-1) :
    SEQ
      UnSign(Areg)
      UnSign(Breg)
      Areg := (Breg << BitsInWord) + Areg
      Creg := 0
      WHILE (Areg BITAND MsbOfDoubleWord) = 0
        SEQ
          Areg := Areg << 1
          Creg := Creg + 1
      Breg := Areg / Range
      Areg := Areg REM Range
      Sign(Areg)
      Sign(Breg)
```

long shift left

```
SEQ
  UnSign(Areg)
  IF
    Areg <= (2*BitsInWord)
      SEQ
        UnSign(Breg)
        UnSign(Creg)
        Breg := (Creg << BitsInWord) + Breg
        Breg := Breg << Areg
        Areg := Breg REM Range
        Breg := (Breg / Range) REM Range
        Sign(Areg)
        Sign(Breg)
```

long shift right

```
SEQ
  Unsign(Areg)
  IF
    Areg <= (2*BitsInWord)
      SEQ
        UnSign(Breg)
        UnSign(Creg)
        Breg := (Creg << BitsInWord) + Breg
        Breg := Breg >> Areg
        Areg := Breg / Range
        Breg := Breg REM Range
        Sign(Areg)
        Sign(Breg)
```

Booting and analysing

test processor analysing

```
SEQ
  Creg := Breg
  Breg := Areg
  IF
    ResetNotAnalysed
    -- This flag indicates that the links were last reset,
    -- as opposed to analysed.
      Areg := FALSE
    TRUE
      Areg := TRUE
```

save high priority queue registers

```
SEQ
  WindexWord(Areg, 0, FptrReg[0])
  WindexWord(Areg, 1, BptrReg[0])
  Areg := Breg
  Breg := Creg
```

save low priority queue registers

```
SEQ
  WindexWord(Areg, 0, FptrReg[1])
  WindexWord(Areg, 1, BptrReg[1])
  Areg := Breg
  Breg := Creg
```

store high priority front pointer

```
SEQ
  FptrReg[0] := Areg
  Areg := Breg
  Breg := Creg
```

store high priority back pointer

```
SEQ
  BptrReg[0] := Areg
  Areg := Breg
  Breg := Creg
```

store low priority front pointer

```
SEQ
  FptrReg[1] := Areg
  Areg := Breg
  Breg := Creg
```

store low priority back pointer

```
SEQ
  BptrReg[1] := Areg
  Areg := Breg
  Breg := Creg
```

store timer

```
SEQ
  ClockReg[0] := Areg
  ClockReg[1] := Areg
  Areg := Breg
  Breg := Creg
  StartTimer()
```

Floating point support

The following constants are used in the floating point support instructions:

```
BitsInFrac = 24          -- number of bits in fraction
PackedLSB  = 1
RealExp    = #FF
RealInf    = #7F800000   -- +Inf
RealRBit   = #80
RealShift  = 8
RealXcess  = #7F
```

unpack single length floating point number

```
SEQ
  UnSign(Areg)
  Creg := Breg * 4
  Areg := ( ( Areg BITAND (BITNOT MinInt)) << (RealShift + 1) )
  Breg := Areg / Range
  Areg := ( Areg REM Range )
  Breg := Breg >> 1
  IF
    Breg = 0
      IF
        Areg = 0
          SKIP
        TRUE
          SEQ
            Creg := Creg + 1
            Breg := 1
    TRUE
      IF
        Breg = RealExp
          IF
            Areg = 0
              Creg := Creg + 2
            TRUE
              Creg := Creg + 3
        TRUE
          SEQ
            Creg := Creg + 1
            Areg := Areg BITOR MinInt
  Sign(Areg)
```

round single length fp number

```
SEQ
  UnSign(Areg)
  Unsign(Breg)
  IF
    Creg < RealExp
      INT temp :
      SEQ
        temp := Breg
        Breg := (Creg * Range) + ((Breg << 1) BITAND (Range - 1))
        Breg := Breg >> (RealShift + 1)
        IF
          (temp BITAND RealRBit) = 0
            SKIP
          (Areg BITOR ((temp BITAND RealXcess) BITOR
          (Breg BITAND PackedLSB))) = 0
            SKIP
          TRUE
            Breg := Breg + 1
        Areg := Breg
    TRUE
      Areg := RealInf
  Sign(Areg)
```

Post-normalise correction of single length fp number

```
SEQ
  UnSign(Areg)
  UnSign(Breg)
  Breg := (Breg * Range) + Areg
  INT temp :
  SEQ
    RIndexWord(Wptr, 0, temp)
    Creg := temp - Creg
  IF
    Creg < - (BitsInFrac - 1)
      SEQ
        Areg := 0
        Breg := 0
        Creg := 0
    Creg < 1
      SEQ
        Breg := Breg >> (1 - Creg)
        Creg := 0
    Creg < RealExp
      SEQ
    TRUE
      Creg := RealExp
  Areg := (Breg REM Range) BITOR Areg
  Breg := Breg / Range
  Sign(Areg)
  Sign(Breg)
```

load infinity

```
SEQ
  Creg := Breg
  Breg := Areg
  Areg := RealInf
```

check single length fp infinity or NaN

```
IF
  (Areg BITAND RealInf) = RealInf
    SetErrorFlag()
  TRUE
    SKIP
```

fractional multiply

```
VAL TwoToThe31 IS 1 << (31-1) :
VAL TwoToThe30 IS 1 << (30-1) :
INT P, L :
SEQ
  P := (Areg * Breg) / TwoToThe31
  UnSign(Areg)
  UnSign(Breg)
  L := (Areg * Breg) \ TwoToThe31
  IF
    L < TwoToThe30
      SKIP
    L = TwoToThe30
      IF
        (P BITAND 1) = 0
          SKIP
        (P BITAND 1) = 1
          P := P + 1
    L > TwoToThe30
      P := P + 1
  OverflowCheck(P)
  Areg := P
  Breg := Creg
```

Testing

The instructions in this section exist for testing the implementation of the transputer. They mainly make available some of the hidden, internal registers of the transputer. In the following descriptions these registers are as follows:

`Dreg`	an extra processor register
`DataReg[linkChans]`	the data registers of the link-channels
`PointerReg[linkChans]`	the pointer registers of the link-channels
`CountReg[linkChans]`	the count registers of the link-channels

store to D register for testing

```
SEQ
  Dreg := Areg
  Areg := Breg
  Breg := Creg
```

store to E register for testing

```
SEQ
  Ereg := Areg
  Areg := Breg
  Breg := Creg
```

store to StatusReg for testing

```
SEQ
  StatusReg := Areg
  Areg := Breg
  Breg := Creg
```

Load D register for testing

```
SEQ
  Creg := Breg
  Breg := Areg
  Areg := Dreg
```

Load E register for testing

```
SEQ
  Creg := Breg
  Breg := Areg
  Areg := Ereg
```

Load StatusReg for testing

```
SEQ
  Creg := Breg
  Breg := Areg
  Areg := StatusReg
```

single step TimeOut for testing

this instruction is very dependant on the actual implementation of the transputer and is not documented here.

test hard channel stack

```
INT chanNum :
SEQ
  ChanOffset(Areg, chanNum)
  Areg := DataReg[chanNum]
  DataReg[chanNum] := PointerReg[chanNum]
  PointerReg[chanNum] := CountReg[chanNum]
  CountReg[chanNum] := Breg
  Breg := Creg
```

2 INITIALISATION, BOOTING, ANALYSING AND CHECKING

2.1 Introduction

This section is concerned how a transputer system is initialised and debugged. The details of the initialisation of the external memory interface are described in a separate section.

2.2 Resetting and Analysing

A transputer is reset in order to initialise its internal state, external memory interface and then to boot. If a transputer is active when it is reset it stops operation immediately. A transputer is reset by pulsing the Reset pin whilst holding the Analyse pin low.

A transputer is analysed in order to investigate its internal state. It stops operation in a way that preserves much of its state and then starts to boot; it does not initialise its external memory interface. A transputer is analysed by taking and holding the Analyse pin high, then pulsing the Reset pin and then taking the Analyse pin low.

After a transputer has booted it is possible to tell whether the transputer was reset or analysed by executing the 'Test Processor Analysing' instruction. This will load the **`Areg`** register with **`MachineTRUE`** if the processor was analysed or with **`MachineFALSE`** if the processor was reset.

Analyse

The Analyse pin exists in order that the state of a transputer system can be investigated. This is achieved by bringing the system to a halt in such a way that the state of the individual transputers in that system can be examined.

A system is analysed by analysing all the transputers in the system in the following manner.

The Analyse pin is asserted which causes the system to come to a halt after a specifiable time. The Reset pin is then asserted, while continuing to assert the Analyse pin, for at least the specified Reset hold time and is then taken low, while still asserting the Analyse pin. The Analyse pin is then de-asserted and the transputer will boot. Note that the earliest time at which the transputer is guarenteed to be able to receive a message remains specified relative to the fall of Reset rather than the fall of Analyse.

Analysing a system brings it to a halt as a result of each transputer in the system coming to a halt. The components of transputer respond to the assertion of Analyse in the following manners:

Processor

The processor only responds to Analyse at certain points during its operation. When one of these point is reached the processor halts any process which is executing and then ignores any scheduling requests made by the links or the timer.

If the processor is not executing a process when analyse is asserted the processor responds at once and halts immediately.

If the processor is executing a process when analyse is asserted the processor responds by halting at either the next descheduling point (ie 'StartNextProcess') or the next point at which a low priority process would be timesliced (this will be an unconditional 'jump' or a 'loop end' instruction). Note that it is possible for a high priority process to pre-empt a low priority process after analyse has been asserted, in which case the processor will halt during the execution of the high priority process. The Iptr of a processor which has been halted in this manner will point to the byte of memory following the final byte of the instruction which caused the process to be halted. A list of instructions on which a process can halt is included at the end of this section.

Timer

The clock stops when analyse is asserted. Any processes waiting for the timer will either be scheduled or will remain on the queue.

Links

The assertion of Analyse has no effect on input links; they continue to operate normally, sending acknowledges and making scheduling requests as appropriate. (Any scheduling requests made after the processor has halted will not succeed).

The assertion of Analyse causes output links to output at most a few more data packets. They respond correctly to acknowledge packets and will make scheduling requests as appropriate. (Any scheduling requests made after the processor has halted will not succeed). The number of data packets which a link will output after Analyse is asserted is bounded by the number of bytes in a processor word.

Information available after booting an analysed transputer

The value that the processor's Wdesc had when the processor halted is available in the Breg register. This will be **(NotProcess.p \ / 1)** if the processor were not active.

The value that was in the processor's **Iptr** when the processor halted is available in the processor's **Areg** register.

The **ErrorFlag** and **HaltOnErrorFlag** are in the same state as when the processor halted.

For some of the other information available after analysis to be meaningful it is necessary to initialise state after booting a transputer. (This is initialisation of state in addition to that needed to ensure correct operation of the transputer).

Provided that the process word associated with a link channel was initialised to **NotProcess.p** then if that process word contains a process descriptor then the channel was being used for output, (unconditional) input or alternative input when the processor halted.

Provided that the count register of an input link channel was initialised to 0 the value in that register will indicate whether the link was ready when Reset was asserted. If the count register contains 1 then the channel was ready. Otherwise (and less usefully) the state of the count register of a link channel will be as described below.

The value in a link channel's count register will be valid provided that the channel has not been used during booting other than for loading the boot program or writing to memory.

If the link channel was being used for output then the value in the count register indicates whether the message transfer had completed. If the count is 0 then the message transfer had completed and the process would have been scheduled if the processor had not halted.

If the link channel was being used for (unconditional) input then the value in the channel's count register indicates whether the message had completed; if the count is 0 or 1 then the message transfer had completed and the process would have been scheduled if the processor has not halted. If the count is 1 then the first byte of a following message had arrived.

NB In general it is NOT possible to perform this examination of a link's count registers!

If two processes are communicating and waiting on either end of a link then the message being transferred is held in the outputing transputer. If a process has input a message but has not yet resumed execution then the message is held correctly in the inputting transputer.

The timer list pointer words may be read and thus the contents of the timer queues may be determined.

The front and back pointers of the process queues may be read and thus the contents of the process queues may be determined.

2.3 Instructions where processor may halt

Instructions which may cause the processor to halt and the consequence of the processor halting on that instruction.

1	Jump	the jump would have been taken.
2	Loop end	the instruction has updated the count locations and the consequential jump would have occured.
3	End Process	the process count will have been updated and the process would have been descheduled
4	Stop Process	the process would have been descheduled
5	Stop On Error	the process would have been descheduled
6	Input Message	the process descriptor will have been left in the channel and the process would have been descheduled.
7	Output Message Output Byte	the process descriptor will have been has output to a channel from which another process was performing alternative input that other process will have been scheduled. The process would have been descheduled.
8	Timer Input	the process will have been inserted into the timer queue and would have been descheduled.
9	Alt Wait	the value Waiting.p will have been written into the State location and the process would have been descheduled.
10	Timer Alt Wait	the value Waiting.p will have been written into the State location, the process will have been inserted into the timer queue if appropriate and the process would have been descheduled.

2.4 Booting

The transputer will boot either as a result of being Analysed or of being Reset. A program can test whether the processor booted as a result of being Reset or Analysed by executing the 'Test Processor Analysing' instruction.

The way in which a transputer boots is controlled by the BootFromRomNotLinks pin; if this pin is held high then the transputer will boot from ROM, if it is held low the transputer will boot from a link.

Booting from ROM

The transputer starts executing in the following state

```
Iptr  = ResetCode       -- two bytes below the top of memory}
Wdesc = MemStart \/ 1   -- low priority, first free word of memory}

Areg = previous value of Iptr
Breg = previous value of Wdesc
Creg is undefined
```

The `ErrorFlag` and `HaltOnErrorFlag` are preserved.

The clocks are stopped.

The process queue pointer registers, timer queue locations and link count registers all contain their previous values.

The pointer, count and data registers of the link channels contain their previous values.

Booting from a link

The first link channel to become active determines the transputer's action.

If the value of the first byte received is 0 then a word of address is input, followed by a word of data which is written to that address. The transputer then determines its further action by the next byte recieved.

If the value of the first byte received is 1 then a word of address is input, a word of data is read from that address and output down the corresponding output link. This will destroy the content of the count register of the outputting link channel. The transputer then determines its further action by the next byte recieved.

If the value of the first byte received is 2 or greater then the transputer inputs that number of bytes into its memory, starting at MemStart and then starts executing in the following state

```
Iptr  = MemStart}
Wdesc = MemStart + (((CodeLength+3) / 4) * 4) \/ 1}
        -- low priority, first free word of memory}
```

`Areg` = previous value of `Iptr`
`Breg` = previous value of `Wdesc`
`Creg` = pointer to the link from which transputer booted

The `ErrorFlag` and `HaltOnErrorFlag` are preserved.

The clocks are stopped.

The values in the process queue pointer registers and timer queue locations are preserved.

The pointer, count and data registers of the link channels, other than the booting channel, contain their previous values.

The count register of the booting channel is preserved.

Actions to be performed by the booting program

The high and low priority front of queue registers must be initialised to `NotProcess.p`. This must occur before the booting program attempts to pass any messages or run any program.

The timer queue words must be initialised to `NotProcess.p` and the clock must be started by executing a 'Store Timer' instruction. This must be done before any attempt is made to wait on the timer.

The `ErrorFlag` and `HaltOnErrorFlag` must be initialised.

In order that the analysis system works properly the link channel process words should be initialised to `NotProcess.p`, the count registers of the link channels should be initialised to zero.

2.5 Error detection by hardware

Certain run time errors such as arithmetic overflow and subscript errors are checked by transputer instructions. These all signal the presence of an error by setting the sticky **ErrorFlag**. This may be explicitly set, cleared and tested by instructions.

The **ErrorFlag** is sticky only within a priority level.

The state of the **ErrorFlag** is brought out of the transputer via the Error pin.

There is mode of operation where whenever the **ErrorFlag** changes from a 0 (unset) to a 1 (set) the processor is brought to an immediate halt. This mode is selected via the **HaltOnErrorFlag** which may be explicitly set, cleared and tested by instructions.

The definition that the processor will halt on a 0 to 1 transition of the **ErrorFlag** ensures that a transputer which has been halted as the result of the **ErrorFlag** being set can be booted and analysed whilst preserving both the **ErrorFlag** and **HaltOnErrorFlag**. The act of clearing the Error bit re-enables the check.

When the processor halts as a result of the Error bit becoming set (ie after the **HaltOnErrorFlag** is set), the **Iptr** will point to the byte of memory which is two bytes beyond the last byte of the instruction which generated the error. (Note that this is not the same state as the **Iptr** of a processor which has been analysed). The processor does not execute any further instructions, does not respond to any Run or Ready requests from the links nor respond to any Timer requests. The timer continues to tick and the links continue to transfer data.

2.6 Instructions which may cause the Error flag to be set

1. Add Constant
2. Check subscript from 0
3. Check count from 1
4. Set Error
5. Add
6. Subtract
7. Multiply
8. Divide
9. Remainder
10. Check word
11. Check single
12. Long Add
13. Long Subtract
14. Long Divide
15. Fractional Multiply
16. Check Floating Point Error

2.7 Differences between halt-on-error and analyse

The state of the **IPtr** of a process which has 'halted on error' or has been analysed can be determined by examining the **Areg** of the processor when it is booted. However, the relationship between the value of the **Iptr** and the instruction which was being exeuted when the processor halted is different in these two cases.

Where a processor has been analysed the **Iptr** will point to the byte of memory following the final byte of the instruction which caused the process to be halted.

Where a processor has halted as a result of the Error bit becoming set (ie after the **HaltOnErrorFlag** is set), the **Iptr** will point to the byte of memory which is two bytes beyond the last byte of the instruction which generated the error.

3 MEMORY CONFIGURATION

3.1 Order of reading configuration information

The configuration register is loaded starting at bit 0 and finishing at bit 35.

3.2 Memory interface configuration address

The configuration addresses are word addresses. The values put out on the memory interface will have bits AD2 to AD31 corresponding to the word address. Bits AD1 and AD0 are 1.

Configuration information is held as close to the top of memory as possible. The two highest byte location of the address space are occupied by the ROM boot instructions so the first available full word is #7FFFFFF8. Therefore addresses #7FFFFF6C through #7FFFFFF8 are used to contain the memory interface configuration information.

In keeping with the standard 'little endian' convention used elsewhere in the transputer architecture the least significant bit corresponds with the least significant address. This means that #7FFFFF6C contains bit 0 and #7FFFFFF8 contains bit 35.

This gives the following association of addresses with bits in the configuration register.

Word address	Bit of configuration register	Function
#7FFFFFF6C	0	T1 lsb
#7FFFFFF70	1	T1 msb
#7FFFFFF74	2	T2 lsb
#7FFFFFF78	3	T2 msb
#7FFFFFF7C	4	T3 lsb
#7FFFFFF80	5	T3 msb
#7FFFFFF84	6	T4 lsb
#7FFFFFF88	7	T4 msb
#7FFFFFF8C	8	T5 lsb
#7FFFFFF90	9	T5 msb
#7FFFFFF94	10	T6 lsb
#7FFFFFF98	11	T6 msb
#7FFFFFF9C	12	notS1 lsb
#7FFFFFFA0	13	notS1
#7FFFFFFA4	14	notS1
#7FFFFFFA8	15	notS1
#7FFFFFFAC	16	notS1 msb
#7FFFFFFB0	17	notS2 lsb
#7FFFFFFB4	18	notS2
#7FFFFFFB8	19	notS2
#7FFFFFFBC	20	notS2
#7FFFFFFC0	21	notS2 msb
#7FFFFFFC4	22	notS3 lsb
#7FFFFFFC8	23	notS3
#7FFFFFFCC	24	notS3
#7FFFFFFD0	25	notS3
#7FFFFFFD4	26	notS3 msb
#7FFFFFFD8	27	notS4 lsb
#7FFFFFFDC	28	notS4
#7FFFFFFE0	29	notS4
#7FFFFFFE4	30	notS4
#7FFFFFFE8	31	notS4 msb
#7FFFFFFEC	32	RefreshInterval lsb
#7FFFFFFF0	33	RefreshInterval msb
#7FFFFFFF4	34	RefreshEnable
#7FFFFFFF8	35	LateWrite

3.3 Memory Map

```
                                      low                    high
Byte Address                         +-----------------------------+
#7FFFFFFE (ResetCodePtr)             |             | Reset Inst    |
                                     +-----------------------------+
#7FFFFFF6C                           |                             |
 .........                           ~    Memory configuration     ~
#7FFFFFFF8                           |                             |
                                     +-----------------------------+
                                     |                             |
                                     |                             |
                                     ~                             ~
                                     |                             |
#80000048 (MemStart)                 |                             |
                                     +-----------------------------+
#80000044 (EregIntSave)              |        Ereg Save Space      |
#80000040 (STATUSIntSave)            |   STATUSreg Save Space      |
#8000003C (CregIntSave)              |        Creg Save Space      |
#80000038 (BregIntSave)              |        Breg Save Space      |
#80000034 (AregIntSave)              |        Areg Save Space      |
#80000030 (IptrIntSave)              |        Iptr Save Space      |
#8000002C (WdescIntSave)             |       Wdesc Save Space      |
                                     +-----------------------------+
#80000028 (LTimerPtr)                | Timer Low  Priority Pointer |
#80000024 (HTimerPtr)                | Timer High Priority Pointer |
                                     +-----------------------------+
#80000020 (ChanTopAddr)              | Event Process Word          |
                                     +-----------------------------+
#8000001C                            | Link 3 Input  Process Word  |
#80000018                            | Link 2 Input  Process Word  |
#80000014                            | Link 1 Input  Process Word  |
#80000010                            | Link 0 Input  Process Word  |
#8000000C                            | Link 3 Output Process Word  |
#80000008                            | Link 2 Output Process Word  |
#80000004                            | Link 1 Output Process Word  |
#80000000 (MostNeg)                  | Link 0 Output Process Word  |
                                     +-----------------------------+
```

4 FUNCTION OF PADS AND PIN-OUTS

4.1 Function of pads

BootFromRomNotLinksPad (Input Pad)

When this input pad is high, the processor will boot itself from the external memory by executing the code at the byte address (**MaxInt- 2**). (**#7FFFFFFE** on the t414).

When this pad is low, the processor will boot itself from the first link to receive data. The first byte is a count value of the number of bytes of code to be received. This count value must not be zero or one. The channel should then receive code. This code is loaded from the first free address above the reserved words of the links, event channel timer queue pointers, and interrupt save locations.

Chan0SpeedPad (Input pad)

This pad controls the Baud rate of Link 0. When **Chan0SpeedPad** is low, Link 0 runs at 10MBaud. When **Chan0SpeedPad** is high, Link 0 runs at 20MBaud if **ChanSpeed50MhzNot25MhzPad** is high. When **Chan0SpeedPad** is high, Link 0 runs at 5MBaud if **ChanSpeed50MhzNot25MhzPad** is low.

Chan1To3SpeedPad (Input pad)

This pad controls the Baud rate of Links 1, 2, and 3. When **Chan1To3SpeedPad** is low, Links 1, 2, and 3 run at 10MBaud. When **Chan1To3SpeedPad** is high, Links 1, 2, and 3 run at 20MBaud if **ChanSpeed50MhzNot25MhzPad** is high. When **Chan1To3SpeedPad** is high, Links 1, 2, and 3 run at 5MBaud if **ChanSpeed50MhzNot25MhzPad** is low.

ChangePadsForTestPad (Input pad)

This pad is taken high only during test. When the pad is taken high, a number of other pads change their function to enable direct reading of the uCode Rom, parametric testing of the Link Output Pads, and checking of the internal link clocks from the link Phase lock loop. The pads change as follows:-

LinkInput[0] becomes **TestShiftIn** (used to shift in the uWord Address for testing the uCode Rom)

LinkInput[1] becomes **EnableuRomTest** (Enables the uCode Rom Test)

LinkInput[2] becomes **notDoDPDriversfromROM** (When high, enables the shift register for the Rom test. When low, allows the DataPath Drivers to read a value from the Rom)

LinkInput[3] becomes **TestShClk** (Shift clock for the uCode test shift register)

StatusErrorOutPad becomes **TestShiftOut** (used to shift out the uWord Data for testing the uCode Rom)

LinkOutput[0] is driven from the inverse of the value on **LinkInput[0]**

LinkOutput[1] is driven from the inverse of the value on **LinkInput[1]**

LinkOutput[2] is driven from the inverse of the value on **LinkInput[2]**

LinkOutput[3] is driven from the inverse of the value on **LinkInput[3]**

HighFromPhi1ToPhi3Pad is driven with the output of the Links Phase Lock Loop divided by 2.

ChanSpeed50MhzNot25MhzPad (Input pad)

This pad is used in conjunction with **Chan0SpeedPad** and **Chan1To3SpeedPad**. When this pad is high, the links can run at either 20MBaud or 10MBaud, depending on the value of the **Chan0SpeedPad** and **Chan1To3SpeedPad**. When this pad is low, the links can run at either 5MBaud or 10MBaud, depending on the value of the **Chan0SpeedPad** and **Chan1To3SpeedPad**.

ClockInPad (Input pad)

This pad is connected to the 5Mhz crystal source, and provides the frequency reference for the processor's phase lock loop. This input is TTL compatible.

DisableInternalRamPad (Input pad; not Bonded)

When this input pad is connected to vdd, both the internal RAM and the Test RAM are disabled and all the address space is given to the external memory interface. A fuse can be blown to have the same effect as taking **DisableInternalRamPad** high.

EventAckPad (Output pad)

When the event channel accepts a request the event channel asserts the **EventAckPad**. The **EventAckPad** will be deasserted when the **EventReqPad** has been deasserted. When the **EventAckPad** is deasserted, the Event Channel is ready to accept the next request.

EventReqPad (Input pad)

When this input pad is asserted a request is made to the event channel.

ExtMemPad_ADPin[31:0] (Bi-directional Pads)

These are the AddressDataPads for the External Memory Interface.

ExtMemPad_BACK (Output pad)

ExtMemPad_BACK going high Acknowledges the AddressDataPads (**ExtMemPad_ADPin[31:0]**) being high impedance after **ExtMemPad_BREQ** is taken high.

ExtMemPad_BREQ (Input pad)

When **ExtMemPad_BREQ** is taken high, then once the External Memory Interface has completed any outstanding processor, or refresh requests, the AddressDataPads (**ExtMemPad_ADPin[31:0]**) will be taken high impedance.

ExtMemPad_MCON (Input pad)

This pad is used to configure the external memory interface. **ExtMemPad_MCON** can be connected dirrectly to one of the AddressDataPads (**ExtMemPad_ADPin[31:0]**) or from the output of an inverter whose input is connected to one of the AddressDataPads. If **ExtMemPad_MCON** is connected dirrectly to an AddressDataPad, then the External Memory Interface will configure to the Pre-Programmed configuration whose address is the same as the number of the AddressDataPad **ExtMemPad_MCON** is connected to. If **ExtMemPad_MCON** is connected to an AddressDataPad using an inverter, then the External Memory Interface will configure from a configuration placed in external ROM.

ExtMemPad_notASPIN (Output pad)

When this signal is taken low, the AddressDataPads (**ExtMemPad_ADPin[31:0]**) hold the correct address for the commencing memory cycle.

ExtMemPad_notPSPIN[3:0] (Output pad)

These pads are user configurable strobes used by the external memory system.

ExtMemPad_notREFRESH (Output pad)

When **ExtMemPad_notREFRESH** is low, a Refresh cycle is in progress.

ExtMemPad_notREPIN (Output pad)

This signal is taken low when a read cycle is to drive the AddressDataPads (**ExtMemPad_ADPin[31:0]**) with the Data read from memory.

ExtMemPad_notWEPIN[3:0] (Output pad)

These signals are the External Memory Interface Byte Write Strobes. With one or more of these Pads taken low, the corresponing byte of Data is written to external memory.

ExtMemPad_WaitPad (Input pad)

This is the wait input for the memory interface.

Gnd

These pads supply 0v.

HighFromPhi1ToPhi3Pad (Output pad)

This pad is used to check that the four internal clocks (Clocks[4:1]) are functioning correctly. **HighFromPhi1ToPhi3** is also used to synchronise correctly for **ExtMemPad_WaitPad**.

LinkInputPad[3:0] (Input pad)

These four pads are the link input pads.

LinkOutputPad[3:0] (Output pad)

These four pads are the link output pads.

PLL_Buff5MhzPad (Output pad)

This is a output buffered version of the processor's input clock.

PLL_RefGndPad

A capacitor should be connected between this input pad and **PLL_RefVddPad**

PLL_RefVddPad

A capacitor should be connected between this input pad and **PLL_RefGndPad**

ProcTimes1notPLLSelPad (Input pad)

When this pad is asserted, the processor four phase clock generator takes its input from the **ClockIn** pad. When low, the four phase clock generator is fed from the processor's phase lock loop output, and it's input is taken from either **ClockInPad** if **Times1notPLLSelPad** is asserted, or **ClockInPad** divided by five if **Times1notPLLSelPad** is not asserted.

ResetPad (Input pad)

When this pad is taken high the processor is reset. If **SystemAnalysePad** is low, then when the **ResetPad** is taken high the external memory interface is also reset. If the external memory interface is reset, then when the reset pad is taken low, the external memory interface will be configured. After the external memory interface has configured, the processor will start to execute.

StatusErrorOutPad (Output pad)

This output pad is asserted when the Status Error bit is set in the processor.

SystemAnalysePad (Input pad)

When this input is taken high, the transputer will come to a clean halt ready for analysis after a system wide error.

Times1NotPLLSelPad (Input pad)

This is a test pad that disables the link phase lock loop. When this pad is high, the input clock is used to form the internal clocks without multiplication. The Clock register for the Timer ticks at one fifth the normal rate, and refresh cycles occur at one fifth normal rate.

Vdd

These pads supply 5v.

4.2 84 lead J-Bend pin-out

Pad Number	Pad Name
1	PLL_RefGndPad
2	Vdd
3	Times1NotPLLSelPad
4	Gnd
5	HoldToGnd
6	ChangePadsForTestPad
7	StatusErrorOutPad
8	BootFromRomNotLinksPad
9	ResetPad
10	DisableInternalRamPad
11	ProcTimes1notPLLselPad
12	SystemAnalysePad
13	ExtMemPad_ADPin[31]
14	ExtMemPad_ADPin[30]
15	ExtMemPad_ADPin[29]
16	Gnd
17	ExtMemPad_ADPin[28]
18	ExtMemPad_ADPin[27]
19	ExtMemPad_ADPin[26]
20	ExtMemPad_ADPin[25]
21	ExtMemPad_ADPin[24]
22	ExtMemPad_ADPin[23]
23	ExtMemPad_ADPin[22]
24	ExtMemPad_ADPin[21]
25	ExtMemPad_ADPin[20]
26	Vdd
27	ExtMemPad_ADPin[19]
28	ExtMemPad_ADPin[18]
29	ExtMemPad_ADPin[17]
30	ExtMemPad_ADPin[16]
31	ExtMemPad_ADPin[15]
32	ExtMemPad_ADPin[14]
33	ExtMemPad_ADPin[13]
34	ExtMemPad_ADPin[12]
35	ExtMemPad_ADPin[11]
36	ExtMemPad_ADPin[10]
37	Gnd
38	ExtMemPad_ADPin[9]
39	ExtMemPad_ADPin[8]
40	ExtMemPad_ADPin[7]
41	ExtMemPad_ADPin[6]
42	ExtMemPad_ADPin[5]
43	ExtMemPad_ADPin[4]
44	ExtMemPad_ADPin[3]
45	ExtMemPad_ADPin[2]
46	ExtMemPad_ADPin[1]
47	ExtMemPad_ADPin[0]

Pad Number	Pad Name
48	Gnd
49	ExtMemPad_notPSPIN[0]
50	ExtMemPad_notPSPIN[1]
51	ExtMemPad_notPSPIN[2]
52	ExtMemPad_notPSPIN[3]
53	Vdd
54	ExtMemPad_notASPIN
55	ExtMemPad_notREPIN
56	ExtMemPad_notWEPIN[0]
57	ExtMemPad_notWEPIN[1]
58	ExtMemPad_notWEPIN[2]
59	ExtMemPad_notWEPIN[3]
60	ExtMemPad_notREFRESH
61	ExtMemPad_WaitPad
62	ExtMemPad_BACK
63	ExtMemPad_BREQ
64	ExtMemPad_MCON
65	EventReqPad
66	Gnd
67	EventAckPad
68	LinkInputPad[3]
69	LinkOutputPad[3]
70	LinkInputPad[2]
71	LinkOutputPad[2]
72	LinkInputPad[1]
73	LinkOutputPad[1]
74	LinkInputPad[0]
75	LinkOutputPad[0]
76	Vdd
77	Chan1To3SpeedPad
78	HighFromPhi1ToPhi3Pad
79	Chan0SpeedPad
80	ChanSpeed50MhzNot25MhzPad
81	PLL_RefVddPad
82	PLL_Buff5MhzPad
83	DonotWire
84	ClockInPad

4.3 84 lead PGA pin-out

Pad Number	Pin Grid Pin Number	Pad Name
1	F1	ProcTimes1notPLLSelPad
2	F2	SystemAnalysePad
3	F3	ExtMemPad_ADPin[31]
4	G1	ExtMemPad_ADPin[30]
5	H1	ExtMemPad_ADPin[29]
6	G2	Gnd
7	J1	ExtMemPad_ADPin[28]
8	G3	ExtMemPad_ADPin[27]
9	K1	ExtMemPad_ADPin[26]
10	H2	ExtMemPad_ADPin[25]
11	J2	ExtMemPad_ADPin[24]
12	H3	ExtMemPad_ADPin[23]
13	J3	ExtMemPad_ADPin[22]
14	K2	ExtMemPad_ADPin[21]
15	K3	ExtMemPad_ADPin[20]
16	H4	Vdd
17	J4	ExtMemPad_ADPin[19]
18	K4	ExtMemPad_ADPin[18]
19	J5	ExtMemPad_ADPin[17]
20	H5	ExtMemPad_ADPin[16]
21	K5	ExtMemPad_ADPin[15]
22	K6	ExtMemPad_ADPin[14]
23	J6	ExtMemPad_ADPin[13]
24	H6	ExtMemPad_ADPin[12]
25	K7	ExtMemPad_ADPin[11]
26	K8	ExtMemPad_ADPin[10]
27	J7	Gnd
28	K9	ExtMemPad_ADPin[9]
29	H7	ExtMemPad_ADPin[8]
30	K10	ExtMemPad_ADPin[7]
31	J8	ExtMemPad_ADPin[6]
32	J9	ExtMemPad_ADPin[5]
33	H8	ExtMemPad_ADPin[4]
34	H9	ExtMemPad_ADPin[3]
35	J10	ExtMemPad_ADPin[2]
36	H10	ExtMemPad_ADPin[1]
37	G8	ExtMemPad_ADPin[0]
38	G9	Gnd
39	G10	ExtMemPad_notPSPIN[0]
40	F9	ExtMemPad_notPSPIN[1]
41	F8	ExtMemPad_notPSPIN[2]
42	F10	ExtMemPad_notPSPIN[3]
43	E10	Vdd

Pad Number	Pin Grid Pin Number	Pad Name
44	E9	ExtMemPad_notASPIN
45	E8	ExtMemPad_notREPIN
46	D10	ExtMemPad_notWEPIN[0]
47	C10	ExtMemPad_notWEPIN[1]
48	D9	ExtMemPad_notWEPIN[2]
49	B10	ExtMemPad_notWEPIN[3]
50	D8	ExtMemPad_notREFRESH
51	A10	ExtMemPad_WaitPad
52	C9	ExtMemPad_BACK
53	B9	ExtMemPad_BREQ
54	C8	ExtMemPad_MCON
55	B8	EventReqPad
56	A9	Gnd
57	A8	EventAckPad
58	C7	LinkInputPad[3]
59	B7	LinkOutputPad[3]
60	A7	LinkInputPad[2]
61	B6	LinkOutputPad[2]
62	C6	LinkInputPad[1]
63	A6	LinkOutputPad[1]
64	A5	LinkInputPad[0]
65	B5	LinkOutputPad[0]
66	C5	Vdd
67	A4	Chan1To3SpeedPad
68	A3	HighFromPhi1ToPhi3Pad
69	B4	Chan0SpeedPad
70	A2	ChanSpeed50MhzNot25MhzPad
71	C4	PLL_RefVddPad
72	A1	PLL_Buff5MhzPad
73	B3	DoNotWire
74	B2	ClockInPad
75	C3	PLL_RefGndPad
76	C2	Vdd
77	B1	Times1NotPLLSelPad
78	C1	Gnd
79	D3	HoldtoGnd
80	D2	ChangePadsForTestPad
81	D1	StatusErrorOutPad
82	E2	BootFromRomNotLinksPad
83	E3	ResetPad
84	E1	DisableInternalRamPad