CnC Network Accessors

Latex of CMNA.tex on November 14, 1990
This document is owned by Cindy Spiller
It may be found in /p1/cm5/doc/cnc/CMNA.tex

Contents

1 Introduction

C and C, or *CnC*, is the name given to C functions which provide access to the hardware in order to facilitate communication between the pe's and the scalar. This document specifies the C functions in the scalar, and the C functions in the pe (C and C) required to acces each of the networks.

1.1 Related Documents

The following documents are related and should be read in conjunction with this one:

- CM5 System Architecture Specification (currently danny's Darpa proposal addendum)
- NI Architecture (/p1/cm5/doc/concepts/ni-architecture.tex)
- Control network architecture: /p1/cm5/doc/concepts/cn-architecture.tex
- data router architecture: /p1/cm5/doc/concepts/data-router.text
- Diagnostics Network Specification (/p1/cm5/doc/modules/dn/dn.tex)

2 CnC Overview

2.1 Coding Conventions

CandC Network Accessors are considered the lowest level programing language of the CM5, or the assembly language. CandC functions and variables use all lower case letters for the each word of a name. Underscores are used between words, and each function name starts with CMNA_. Ex. CMNA_dr_send_msg.

- Function names and variables use lower case letters for each character in words of name.
- Constants are all capital letters with underscores separating words.
- Typedefs are all capital letters with underscores separating words and a _T at the end.

2.2 Definition

CandC provides functions for the data router, combine, broadcast, global, and sync networks. Communication on these networks is controlled by the NI chip. CnC's main objective is to provide an abstract access to the control of this chip, yet be as efficient as possible. The users of CnC will be the C* compiler, Fortran compiler, CM Run Time, diagnostics, the debugger, and operating system developers. CnC may also be made available to other users as the low level (assembly type) language.

Note that the term /em function will be used to express the action to be performed by CnC. However, this term is used loosely, and the action may be performed by a true C function interface, or whenever possible, by a macro.

CandC is provided as a library to be linked with users code. The header file `/p1/cm5/os/include/CMNA.h` specifies the constants required when linking with CandC Network Accessors.

CandC functions are called by the processors in a partition, called pe's, as well as by the scalar processor. The scalar processor is usually a large capacity cpu which is physically distinct from the pe's. Throughout this document, a description of the action performed will be dependent on whether the function is being called from a pe or from the scalar, or if it is the same when called by all processors. Whenever the term 'all processors' is used, this refers to all pe's and scalar. The term 'all pe's' refers to all the processors in the partition except the scalar.

There is another distinction regarding processor status. There can be a pe within the partition which obtains the status of 'broadcaster'. The term broacaster simply means that this pe has control of the broadcast network, and it can send broadcasts while everyone else can receive them. Generally the scalar will be the broadcaster, but this power can be relinquished to any pe in the partition. Of course this power can only be given to one pe at any one time. This pe will be refered to as the 'broadcaster'.

The scalar usually does not participate in contributing to combine operations, but receives the result of a combine operation which has been 'reduced'. However, any pe in the partition can be set up to not combine, yet have the reduce result sent to him.

3 Constants

- MAX_ROUTER_MSG_WORDS Maximum number of words which can be placed in the router fifo for one message. Current maximum is 6 words including the destination.
- MAX_BROADCAST_MSG_WORDS Maximum number of words which can be placed in the broadcast fifo for one message. Current maximum is 15 words.
- MAX_COMBINE_MSG_WORDS Maximum number of words which can be placed in the combine fifo for one message. Current maximum is 4 words.
- etc.

4 Variables

4.1 CMNA_self_address

int CMNA_self_address;

Relative address of this PE or scalar within the partition.

4.2 CMNA_scalar_address

int CMNA_scalar_address;

Address of scalar processor.

4.3 CMNA_partition_size

int CMNA_partition_size;

Number of PE's in the partition. This does not count the scalar, spares, or IO devices in the partition.

5 Participating and Abstaining from network activity

Each processor in the partition is connected to the following networks: Broadcast, Combine, Router, and Global. The networks assume that since the PE is connected to the network, that it will participate in all activity on the network. If the pe wishes to not participate in some activity, it can set a bit to indicate which activity it wishes to abstain from.

The logical activities that a processor would wish to abstain from are the ones which require synchronization from all the processors before the operation will complete. If a processor does not plan to contribute, then it must set its abstain bit to indicate this so that the operation will complete without it.

The processor can abstain from receiving or sending to the broadcast network. However, only one processor (called the broadcaster) can be broadcasting at any time, so any processor enabling its abstain bit to allow sending broadcasts has to be coordinated with all the other processors.

The processors can abstain from contributing to the combine operation, and can abstain from receiving combine reduce results.

Since the router network does not require that every processor participate in an operation to complete, there are no abstain bits for the router network.

5.1 initial configuration

Note: add hardware reset state.

Initially, the abstain bits are set such that the scalar is the broadcaster, and all pe's will receive the broadcast. All the pe's will participate in the combine operations, and all the pe's will receive reduce results. The PE's will also participate in the Sync operation. The scalar will not participate in Sync, combine, or reduce receive.

5.2 changing participation

At various times after initialization, different processors will need to change their participation. These functions provide that capability. The activity that can be participated in or which can be abstained from include:

- NI_REDUCE_RECEIVE
- NI_BC_RECEIVE
- NI_COMBINE
- NI_SYNC_GLOBAL

```
CMNA_participate_in(activity)
int     activity;

Example:

CMNA_participate_in (NI_REDUCE_RECEIVE | NI_SYNC_GLOBAL);

CMNA_abstain_from(activity)
int     activity;

Example:

CMNA_abstain_from(NI_SYNC_GLOBAL);
```

5.2.1 EstablishBroadcaster

NOTE: This section not available yet.

This function must be synchronized amoung all processors, therefore all processors in the partition (including the scalar) must execute this function before it will return. The processor whose address is specified will become the broadcaster.

5.2.2 OnlyPesCMNA_com

Sets up the combine network so that only the pe's in the partition (not the scalar) contribute to the combine operation.

5.2.3 ScalarCMNA_comsAlso

All pe's and the scalar contribute to the combine operation.

5.2.4 ReduceToScalarOnly

A combine reduce result is available in the scalar only.

5.2.5 ReduceToPesOnly

The combine reduce result is available to the pe's in the partition only.

5.2.6 ReduceToAll

The combine reduce result is available to all pe's as well as the scalar.

6 Global

The global network or interface provides a bit which can be globally updated by every processor in the partition including the scalar. There are three bits available. The synchronous bit, and two asynchronous bits, one for the user and one for the supervisior. The asynchronous bits are updated almost instantly without the need for input from any other processor. The synchronous bit is updated only after every processor has contributed its bit.

6.1 Global Sync

The synchronous global facility provides a way for each processor in the partition to OR a bit with every other processor in the partition. Each processor presents a bit which the hardware then OR's with a bit from every other processor which is participating (not abstaining) in the sync. When all the participating processors have presented their bits, then the hardware indicates that the sync is complete. The result is available to be read.

6.1.1 CMNA_or_global_sync_bit

This function takes a 32 bit quanity, of which only the least significant bit is relevant. This bit is presented to be OR'ed with the other processors' bits.

```
CMNA_or_global_sync_bit(value)
unsigned int value;
```

6.1.2 CMNA_global_sync_complete

After CMNA_or_global_sync_bit() is called, the hardware has to coordinate all the other processors to obtain the result. When this result is available CMNA_global_sync_complete will return true. It will return false until this result is available, and will become false again after the next call to CMNA_or_global_sync_bit().

```
int CMNA_global_sync_complete()
```

6.1.3 CMNA_global_sync_read

Note: explain what happend if this is executed before sync complete .

After CMNA_global_sync_complete() returns true, the result can be read. CMNA_global_sync_read returns a 32bit quantity which only the least significant bit is relevant.

```
int CMNA_global_sync_read()
```

6.1.4 CMNA_global_sync_read_ready

After CMNA_or_global_sync_bit() has been called, the user can call CMNA_global_sync_readWhenComplete() which will wait until the result is available, and then return the result.

```
int CMNA_global_sync_readWhenComplete()
```

6.1.5 CMNA_global_sync

The entire global sync operation can be performed in one call. CMNA_global_sync takes the 32 bit quantity, presents it to the hardware, waits for the operation to complete, and then returns the result.

```
int CMNA_global_sync(value)
unsigned int value;
```

6.2 Global Async

This facility is like the Global Sync in that one bit is presented from a processor, OR'ed with the bits from every other processor, and the result is updated in all pe's. The major difference is that one or more pe's can present its bit, and the result is calculated without the the need for all processors in the partition to participate. Therefore there is no need for abstaining from Global Async, just do not participate.

6.2.1 CMNA_or_global_async_bit

```
CMNA_or_global_async_bit(value)
unsigned int    value;
```

This function takes a 32 bit quanity, of which only the least significant bit is relevent. This bit is OR'ed with the other processors' bits, however, the other processors do not have to all update this bit before the result becomes available.

6.2.2 CMNA_global_async_read

```
int CMNA_global_async_read()
```

When a processor presents its bit, the hardware must OR this with the bits previously presented by all the other processors. This takes a few cycles to perform, and there is no indication of when this is completed. If this bit is going to be written and read immediately (calling CMNA_or_global_async_bit() and then immediately calling CMNA_global_async_read()), CMNA_global_async should be called. CMNA_global_async_read should be used only when inspecting the current value of the global async bit.

6.2.3 CMNA_global_async

Note: explain worst case cycles.

This function logically calls CMNA_or_global_async_bit() and then guarantees (by waiting for worst case cycles) that the result must be valid, and returns the result.

```
int CMNA_global_async(value)
unsigned int value;
```

7 Broadcast Network Accessors

The following routines allow broadcasting units of data between the broadcaster and all other pe's. Any pe or the scalar can acquire the control of the broadcast network and issue a broadcast. This must be synchronized, so that only one processor in the partition controls the network at any one time. All other processors in the partition can receive the message from the broadcast net. The processors can abstain from receiveing any broadcast messages if requested.

7.1 Broadcast Basic Accessors

Any broadcast architecture functionality can be accomplished by applying a combination of the following functions (plus the abstain functions) into the proper algorithm. In order to use these functions, you must proclaim yourself an expert of the entire architecture of the CM5. These are the building blocks used to implement the macros and functions in the remainder of this section. It is highly recommended that the macros and functions be used instead of these. If the architecture changes, these functions will most likely become invalidated.

7.2 Checking Broadcast Network Status

CMNA_bc_status

Purpose: Returns the status of the broadcast network.

Prototypes:

```
extern CMNA_bc_status _AP(());
```

Definitions: The `CMNA_bc_status` routine returns returns the value in the broadcast status register. Bit maps are provided to map out any combination of the values available in the status register. Also, macros are provided to obtain all the fields in the status register.

Performance Notes: *missing*

Networks Used: Broadcast network.

Restrictions: *missing*

Bitmaps:

- NI_SEND_SPACE_P NI_SEND_SPACE_L
- NI_REC_OK_P NI_REC_OK_L
- NI_SEND_OK_P NI_SEND_OK_L
- NI_SEND_EMPTY_P NI_SEND_EMPTY_L
- NI_REC_LENGTH_LEFT_P NI_REC_LENGTH_LEFT_L

Macros:

- SEND_SPACE(status)
- RECEIVE_OK(status)
- SEND_OK(status)
- SEND_EMPTY(status)
- RECEIVE_LENGTH_LEFT(status)

Example of using these Macros:

```
int  LengthLeft;
LengthLeft = RECEIVE_LENGTH_LEFT(CMNA_bc_status());
```

Example of using the bit maps:

```
#define RECEIVE_LENGTH_LEFT(status) \
  ((status >> NI_REC_LENGTH_LEFT_P) & ~(~0 << NI_REC_LENGTH_LEFT_L))
```

7.2.1 CMNA_bc_send_first

CMNA_bc_send_first

Prototypes:

```
extern CMNA_bc_send_first _AP((int length, unsigned msg));
```

Formats:

length: The length in words to be sent.

msg: Pointer to the data to be sent. Data must be word aligned.

Definitions: The `CMNA_bc_send_first` routine sets up the `length` to be sent, and sends the first word into the broadcast network.

Performance Notes: *missing*

Networks Used: Broadcast network.

Restrictions: *missing*

CMNA_bc_send_word

Prototypes:

```
extern CMNA\_sbc\_word _AP((int data));
```

Formats:

data: 32 bits of data to be sent.

Definitions: The **CMNA_bc_send_word** routine sends 32 bits of data into the broadcast network.

Performance Notes: *missing*

Networks Used: Broadcast network.

Restrictions: *missing*

CMNA_bc_receive_word

Prototypes:

```
extern _AP((unsigned data)) CMNA\_bc\_receive\_word ;
```

Definitions: The `CMNA_bc_receive_word` routine returns 32 bits of data from the broadcast network. Receive Ok must be true, and there must be a word in the receive fifo to extract.

Performance Notes: *missing*

Networks Used: Broadcast network.

Restrictions: *missing*

If the broadcaster has some data which can be sent out in one message, i.e. is less then MAX_BC_MSG_WORDS, and is already constructed, then it can use CMNA_bc_send_fifo_amount to send the data. After assuring that it has control of the network, the broadcaster issues a CMNA_bc_send_fifo_amount to broadcast a unit of data, and the pe's execute a CMNA_bc_receive to receive the unit of data being broadcast.

7.2.4 CMNA_bc_send_fifo_amount

CMNA_bc_send_fifo_amount, CMNA_bc_send_msg

Purpose: Broadcasting data from Broadcaster to rest of partition.

Prototypes:

```
extern CMNA_bc_send_fifo_amount _AP((void*msg, int length));
extern CMNA_bc_send_msg _AP((void*msg, int length));
```

Formats:

msg: Pointer to the data to be sent. Data must be word aligned

length: The length in WORDS to be written.

Definitions: The `CMNA_bc_send_fifo_amount` routine writes `length` words into broadcast network. `length` must be less than or equal to the `MAX_BC_FIFO_WORDS` which is the maximum words which will fit in one fifo. The `CMNA_bc_send_msg` routine writes `length` words into the broadcast network. It takes care of packaging large messages into fifo amounts. The corresponding function `CMNA_bc_receive` must be called in all the other PE's which are participating in broadcast receive.

Performance Notes: *missing*

Networks Used: Broadcast network.

Restrictions: *missing*

CMNA_bc_receive

Purpose: Receiving data from the broadcast network.

Prototypes:

```
extern CMNA_bc_receive _AP((void*msg, unsigned length));
```

Formats:

msg: Pointer to the data to be received. Data must be word aligned. If the pointer is null, then the length will be read from the network and discarded.

length: The length in WORDS to be read.

Definitions: The `CMNA_bc_receive` routine reads `length` words from the broadcast network. Since words do not arrive in packets, there is no notion of fifo amount no receive.

Performance Notes: *missing*

Networks Used: Broadcast network.

Restrictions: *missing*

7.3 Broadcasting C types

These broadcast functions are more efficient if the unit being broadcast is of the type: int, unsigned int, float, or double.

CMNA_bc_read_*

Purpose: Reading C types from Broadcast Network.

Prototypes:

```
extern int CMNA_bc_read_int \_AP(());
extern unsigned CMNA_bc_read_uint \_AP(());
extern float CMNA_bc_read_float _AP(());
extern double CMNA_bc_read_double _AP(());
```

Performance Notes: *missing*

Networks Used: Broadcast network.

Restrictions: *missing*

BroadcastRead* is called by the pe's to accept a data unit from the broadcast network. The scalar must be executing a Broadcast* to inject this unit of data into the broadcast network.

CMNA_bc_write_*

Purpose: Writeing C types to the broadcast network.

Prototypes:

```
extern CMNA_bc_write_int _AP((int data));
extern CMNA_bc_write_uint _AP((unsigned data));
extern CMNA_bc_write_float _AP((float data));
extern CMNA_bc_write_double _AP((double data));
```

Formats:

data: C type

Definitions: The `CMNA_bc_write_*` routines write C types to the broadcast network.

Performance Notes: *missing*

Networks Used: Broadcast network.

Restrictions: *missing*

8 Router

The router network functions are higher level routines which correctly incorporate the Network Interface (NI) accessors to perform the steps necessary to perform a router command.

This section needs more work. More functions are to be defined.

8.1 Router basics

Any router architecture functionality can be accomplished by applying a combination of the following functions (plus the abstain functions) into the proper algorithm. In order to use these functions, you must proclaim yourself an expert of the entire architecture of the CM5. These are the building blocks used to implement the macros and functions in the remainder of this section. It is highly recommended that the macros and functions be used instead of these. If the architecture changes, these functions will most likely become invalidated.

8.1.1 CMNA_dr_status

CMNA_dr_status, CMNA_ldr_status, CMNA_rdr_status

Purpose: Returning the status of the Router Network

Prototypes:

```
extern CMNA_dr_status _AP(());
extern CMNA_ldr_status _AP(());
extern CMNA_rdr_status _AP(());
```

Definitions: The `CMNA_dr_status` returns the value in the router status register. Bit maps are provided to map out any combination of the values available in the status register. Macros are provided to use extract the fields in the status register. `CMNA_ldr_status` returns the status of the left data router, and `CMNA_rdr_status` returns the status of the right data router.

Performance Notes: *missing*

Networks Used: Router network.

Restrictions: *missing*

These bitmaps include:

- NI_SEND_SPACE_P NI_SEND_SPACE_L
- NI_REC_OK_P NI_REC_OK_L
- NI_SEND_OK_P NI_SEND_OK_L
- NI_REC_LENGTH_LEFT_P NI_REC_LENGTH_LEFT_L
- NI_REC_LENGTH_P NI_REC_LENGTH_L
- NI_DR_REC_TAG_P NI_DR_REC_TAG_L
- NI_DR_SEND_STATE_P NI_DR_SEND_STATE_L
- NI_DR_REC_STATE_P NI_DR_REC_STATE_L

Macros:

- SEND_SPACE(status)
- RECEIVE_OK(status)
- SEND_OK(status)
- DR_ROUTER_DONE(status)
- RECEIVE_LENGTH_LEFT(status)
- RECEIVE_LENGTH(status)
- DR_RECEIVE_TAG(status)
- DR_SEND_STATE(status)
- DR_RECEIVE_STATE(status)

Example of using these Macros:

```
int  Tag;

Tag = DR_RECEIVE_TAG(CMNA_dr_status());
```

Example of using the bit maps:

```
#define DR_RECEIVE_TAG(status) \
  ((status >> NI_REC_TAG_P) & ~(~0 << NI_REC_TAG_L))
```

CMNA_xdr_send_first

Purpose: Set up the length of the packet and send the destination address.

Prototypes:

```
extern CMNA_dr_send_first _AP((unsigned tag, int length, unsigned msg));
extern CMNA_ldr_send_first _AP((unsigned tag, int length,
unsigned msg));
extern CMNA_rdr_send_first _AP((unsigned tag, int length,
unsigned msg));
```

Formats:

msg: Destination processor number.

length: The length in WORDS of the packet to be written.

Definitions: The `CMNA_xdr_send_first` routine sets up the `length` words to be sent to the router network. The destination PE address `msg` is pushed to the fifo. This word is not counted in the `length`. `CMNA_dr_send_first` sends into the middle data router, `CMNA_ldr_send_first` sends into the left data router, and `CMNA_rdr_send_first` sends into the right data router.

Performance Notes: *missing*

Networks Used: Router network.

Restrictions: *missing*

CMNA_xdr_send_word

Purpose: Pushes a word onto the Router fifo.

Prototypes:

```
extern CMNA_dr_send_word _AP((unsigned msg))
extern CMNA_ldr_send_word _AP((unsigned msg))
extern CMNA_rdr_send_word _AP((unsigned msg))
```

Formats:

msg: 32 bits of data to be pushed.

Definitions: The `CMNA_xdr_send_word` routine pushes 32 bits onto the router fifo after `CMNA_xdr_send_first` has been called.

Performance Notes: *missing*

Networks Used: Router network.

Restrictions: *missing*

CMNA_xdr_receive_word

Purpose: Pushes a word onto the Router fifo.

Prototypes:

```
extern _AP((unsigned msg)) CMNA_dr_receive_word
extern _AP((unsigned msg)) CMNA_ldr_receive_word
extern _AP((unsigned msg)) CMNA_rdr_receive_word
```

Definitions: The `CMNA_xdr_receive_word`routine pops 32 bits from the router fifo.

Performance Notes: *missing*

Networks Used: Router network.

Restrictions: *missing*

8.2 Router functions and macros

These following functions are the ones recommended to be used. These functions combine the router basic functions in the correct algorithm to perform the given task.

8.2.1 CMNA_set_count_mask

The router only counts messages which tags have been set in the count mask register. This function provides a way to ensure that your router messages will be accounted for when doing a RouterEmpty. (This may only be available to the supervisior in the future). Currently, tags 8 through 15 are available for the user.

CMNA_set_count_mask

Purpose: Sets the mask which causes messages to be counted.

Prototypes:

```
extern CMNA_set_count_mask _AP((unsigned mask))
```

Formats:

mask: Sets the mask of messages to be counted

Definitions: The `CMNA_set_count_mask` routine

Performance Notes: *missing*

Networks Used: Router network.

Restrictions: *missing*

8.2.2 CMNA_dr_send_msg

This function sends word_length words that are pointed to by source_base, and sends them to the specified destination processor. This function takes care of chopping up large data streams into packets that fit into the send fifo, and sequencing them to be re-assembled at the receiving end. CMNA_dr_receive_msg must be called in the destination processor to receive this message and re-assemble it.

CMNA_dr_send_msg

Prototypes:

```
extern CMNA_dr_send_msg _AP((unsigned dest_proc, void*source_base,
     int word_length,
     unsigned tag));
extern CMNA_ldr_send_msg _AP((unsigned dest_proc, void*source_base,
     int word_length,
     unsigned tag));
extern CMNA_rdr_send_msg _AP((unsigned dest_proc, void*source_base,
     int word_length,
     unsigned tag));
```

Formats:

dest_proc: Destination processor number.

source_base: Word aligned pointer to message to be sent.

word_length: Length in words of message to be sent.

tag: Tag of message being sent.

Definitions: The `CMNA_xdr_send_msg` sends `length` words pointed to by `source_base` onto the Router Network using the tag specified. This function takes care of packaging long messages into fifo amounts and sending these into the router network. `CMNA_xdr_receive_msg` must be called on the processor which this message is intended for.

Performance Notes: *missing*

Networks Used: Router network.

Restrictions: *missing*

8.2.3 CMNA_dr_receive_msg

This function receives fifo length packets which were sent by CMNA_dr_send_msg, and asembles them back into the original message. It returns when the word_length specified has been received.

CMNA_dr_send_msg

Prototypes:

```
extern void CMNA_dr_receive_msg _AP((void*base, int word_length));
extern void CMNA_ldr_receive_msg _AP((void*base, int word_length));
extern void CMNA_rdr_receive_msg _AP((void*base, int word_length));
```

Formats:

base: Word aligned pointer for message to be received into.

word_length: Length in words of message to be received.

Definitions: The `CMNA_xdr_receive_msg` receives **word_length** from the Router Network and stores it in the location pointed to by **base**. If **base** is null, the data is thrown away. This function takes fifo amounts off the router network in any order and then re packages them into the correct order. It must be receiving data from the `CMNA_xdr_send_msg` function.

Performance Notes: *missing*

Networks Used: Router network.

Restrictions: *missing*

This function indicates that the user is going to stop sending messages, and wait until the router network is emptied of all messages. The processor will receive all messages still in the network. This is a synchronization function, therefore requires that all processors in the partition call this function before the emptying can complete. The router empty operation uses the combine network. Any combines issued after a CMNA_start_emptying_router will not complete until after CMNA_router_emptied returns true.

CMNA_start_emptying_router

Prototypes:

```
extern CMNA_start_emptying_router _AP(());
```

Definitions: The `CMNA_start_emptying_router` initiates a Router Done operation.

Performance Notes: *missing*

Networks Used: Combine network.

Restrictions: *missing*

CMNA_router_emptied

Prototypes:

```
extern _AP(()) CMNA_router_emptied _AP(());
```

Definitions: Returns true when all processors have called CMNA_start_emptying_router, and all messages in the router have been received by the appropriate processor (the router is empty).

Performance Notes: *missing*

Networks Used: Combine network.

Restrictions: *missing*

9 Combine (Scan)

The Combine interface allows the pe's to perform a given pattern and combine operation on a unit of data. The allowable patterns are: SCAN_FORWARD, SCAN_BACKWARD, SCAN_REDUCE, and the combines are: OR_SCAN, XOR_SCAN, ADD_SCAN, UADD_SCAN, and MAX_SCAN. For example, this amounts to the pe's OR'ing their data with the data from everyone else in the partition. The result of this operation can be sent to the scalar by using the SCAN_REDUCE combiner.

The combine interface allows these operations to be performed on up to MAX_COMBINE_MSG_WORDS. The operations are performed as though the data is one long word which is length * 32 bits long.

The Scan network allows the pe's to perform forward and backward scans, and it allows the scalar and/or all the pe's to receive the reduce of any of these operations.

9.1 CMNA_com basics

Any combine architecture functionality can be accomplished by applying a combination of the following functions into the proper algorithm. In order to use these functions, you must proclaim yourself an expert of the entire architecture of the CM5. These are the building blocks used to implement the macros and functions in the remainder of this section. It is highly recommended that the macros and functions be used instead of these. If the architecture changes, these functions will most likely become invalidated.

9.1.1 CMNA_com_status

CMNA_com_status returns the value in the combine status register. Bit maps and macros are provided to map out any combination of the values available in the status register.

CMNA_com_status

Prototypes:

```
extern _AP((int))  CMNA_com_status();
```

Definitions: The `CMNA_com_status` returns the status fo the cmobine network.

Performance Notes: *missing*

Networks Used: Co dntrolmbine network.

Restrictions: *missing*

Bitmaps:

- NI_SEND_SPACE_P NI_SEND_SPACE_L
- NI_REC_OK_P NI_REC_OK_L
- NI_SEND_OK_P NI_SEND_OK_L
- NI_SEND_EMPTY_P NI_SEND_EMPTY_L
- NI_REC_LENGTH_LEFT_P NI_REC_LENGTH_LEFT_L
- NI_REC_LENGTH_P NI_REC_LENGTH_L
- NI_COM_OVERFLOW_P NI_COM_OVERFLOW_L

Macros:

- SEND_SPACE(status)
- RECEIVE_OK(status)
- SEND_OK(status)
- SEND_EMPTY(status)
- RECEIVE_LENGTH_LEFT(status)
- RECEIVE_LENGTH(status)
- COMBINE_OVERFLOW(status)

Example of using these Macros:

```
int  Overflow;

Overflow = COMBINE_OVERFLOW(CMNA\_com\_status());
```

Example of using the bit maps:

```
#define COMBINE_OVERFLOW(status) \
  (status & (1 << NI_COM_OVERFLOW_P))
```

CMNA_comr_send_first

Purpose: Set up the length of the packet and send the destination address.

Prototypes:

```
extern CMNA_com_send_first _AP((int combiner, int pattern, int length,
unsigned msg));
```

Formats:

combiner: Either of the combine types

pattern: Either SCAN_FORWARD, SCAN_BACKWARD, or SCAN_REDUCE

msg: pointer to message to be sent.

length: The length in words of the packet to be written.

Definitions: The `CMNA_com_send_first` routine sets up the `length` words to be sent to the control network. The first word of the message is pushed to the fifo.

Performance Notes: *missing*

Networks Used: Control network.

Restrictions: *missing*

CMNA_com_send_word

Purpose: Set up the length of the packet and send the destination address.

Prototypes:

```
extern CMNA_com_send_word _AP((unsigned data));
```

Formats:

data: word of data to be sent.

Definitions: The `CMNA_com_send_word` routine pushes a word onto the fifo, it can only be calledc after calling `CMNA_com_send_first`.

Performance Notes: *missing*

Networks Used: Control network.

Restrictions: *missing*

CMNA_com_receive_word

Purpose: Receives a word from the network.

Prototypes:

```
extern _AP((unsigned)) CMNA_com_receive_word();
```

Definitions: The `CMNA_com_receive_word` routine pops a word from the fifo. There must be a word to pop when this function is called.

Performance Notes: *missing*

Networks Used: Control network.

Restrictions: *missing*

9.2.1 CMNA_segment_start

CMNA_segment_start

Purpose: Sets the scan start bit to indicate that the combine operation will be performed with this PE at the start of a segment..

Prototypes:

```
extern CMNA_set_segment_start _AP((unsigned value));
```

Formats:

value: A 1 to set the segment or a 0 to unset the segment bit.

Definitions: The `CMNA_segment_start` routine specifies to the network that this processor starts a new segment.

Performance Notes: *missing*

Networks Used: Control network.

Restrictions: *missing*

CMNA_com_send

Purpose: Sends a combine operation into the combine network.

Prototypes:

```
extern CMNA_com_send _AP((int combiner, int pattern, void*data,
  int word_length));
```

Formats:

combiner: Either of the combine types

pattern: Either SCAN_FORWARD, SCAN_BACKWARD, or SCAN_REDUCE

data: pointer to message to be sent.

length: The length in words of the packet to be written.

Definitions: CMNA_com_send takes a maximum of MAX_COMBINE_MSG_WORDS, sets up the send first register, and then pushes length words onto the fifo. It then injects the fifo into the network, and returns when the message has been sent.

Performance Notes: *missing*

Networks Used: Control network.

Restrictions: *missing*

CMNA_com_receive

Purpose: Receives the message in the combine receive fifo

Prototypes:

```
extern CMNA_com_receive _AP((void*result));
```

Formats:

result: Pointer to location to store result. If null, the current fifo is emptied, but the data is discarded.

Definitions: `CMNA_com_receive` copies the data out of the combine receive fifo, and places it into the receive buffer provided. The caller is expected to know how long this data is and what type it is. This function can be called from the scalar or pe.

Performance Notes: *missing*

Networks Used: Control network.

Restrictions: *missing*

CMNA_com

Purpose: Performs a combine operation both sending and receiving the combine data.

Prototypes:

```
extern CMNA_com _AP((int combiner, int pattern, void*data, int length,
    void*result));
```

Formats:

combiner: Either of the combine types

pattern: Either SCAN_FORWARD, SCAN_BACKWARD, or SCAN_REDUCE

data: pointer to message to be sent.

length: The length in words of the packet to be written.

result: Pointer to location to store result of combine operation.

Definitions: The `CMNA_com` routine performs a complete combine operation doing a send with the `combiner` and `pattern` specified for the `length` specified. The result is stored in `result`. This simply does a CMNA_com_send followed by a CMNA_com_receive.

Performance Notes: *missing*

Networks Used: Control network.

Restrictions: *missing*

10 Synchronous messages between Scalar and Pe's

10.1 Sending from Scalar to one PE

Write*ToProcessor is called from the scalar and results in this unit of data being sent to the specified pe. The pe's must be executing a Read*FromScalar simultaneously in order to maintain synchronization. Note that all the pe's will be executing this function, but only the specified one will accept the value. The contents of the variable 'value' will remain unchanged in the pe's which were not recipents of the write.

The write to processor functions take a known type and send it. These are macros which execute faster as the length is not required, and no type casting is necessary. WRITE MSG TO PROCESSOR could be used for all these functions instead.

10.1.1 Write*ToProcessor

```
CMNA\_write\_int\_to\_pe(Pe, Value)
unsigned int    Pe;
int             Value;


CMNA\_write\_uint\_to\_pe(Pe, Value)
unsigned int    Pe;
unsigned int    Value;

CMNA\_write\_float\_to\_pe(Pe, Value)
unsigned int    Pe;
float           Value;

CMNA\_write\_double\_to\_pe(Pe, Value)
unsigned int    Pe;
double          Value;
```

10.2 Read*FromScalar

```
int CMNA\_read\_int\_from\_scalar

uint CMNA\_read\_uint\_from\_scalar

float CMNA\_read\_float\_from\_scalar
```

10.3 Sending from one PE to Scalar

Read*FromProcessor is called by the scalar and results in this unit of data being received from the specified pe. The pe's must be executing a Write*ToScalar to maintain synchronization.

```
extern int  CMNA_read_int_from_pe _AP((unsigned pe));
extern unsigned int  CMNA_read_uint_from_pe _AP((unsigned pe));
extern float CMNA_read_float_from_pe _AP((unsigned pe));
extern double CMNA_read_double_from_pe _AP((unsigned pe));
```

10.3.1 WriteMsgToScalar

```
WriteMsgToScalar(Value, Length)
unsigned int    *Value;
int Length;
```

10.3.2 Write*ToScalar

```
CMNA\_write\_int\_to\_scalar(value)
int     value;

CMNA\_write\_uint\_to\_scalar(value)
usigned     value;

CMNA\_write\_float\_to\_scalar(value)
float   value;

CMNA\_write\_double\_to\_scalar(value)
double  value;
```

A Example Code

Code is seperated into seperate files to split scalar code from the pe code. The following file is the scalar code.

combine_test.c

```
#include <CMNA.h>

typedef unsigned int word;
int expected_results[8][4] =
{
  0, 10, 21, 33,
  0x080000000, 10, 11, 12,
  0, 10, 11, 15,
  0, 10, 21, 33,
  0, 10, 1, 13,
  36, 25, 13, 0,
  13, 13, 13, 0x080000000,
  46, 46, 46, 46
};

#include "combines.intf"

/* This is the scalar stuff */
scalar_main()
{
  test_combines();
}

test_combines()
{
  extern pe_scan_test();

  if (!CMNA_participate_in(NI_REDUCE_RECEIVE))
    printf("ERROR, can't participate in reduce receive\n");

  /* tell PEs to run their half of scan test */
  pe_scan_test(1, '2', 3);

  /* run our half */
  scalar_scan_test();
}

int failurep = 0;
int verification = 1;
```

```
verify(index, string)
int index;
char *string;
{
  int pe;
  int result;
  int overflow;

  pe = 0;
  printf("%s\n\nResults: ", string);
  while(pe < CMNA_partition_size)
  {
    result = CMNA_read_int_from_pe((unsigned int) pe);
    overflow = CMNA_read_int_from_pe((unsigned int) pe);
    printf (" %d", result);
    if (overflow) printf("(OV)");
    if (verification)
    {
      if (expected_results[index][pe] != result)
      {
failurep = 1;
printf("<--ERROR");
      }
    }
    pe++;
  }
  printf("\n\n");
}

scalar_scan_test()
{
  unsigned *scan_read();
  unsigned int result;

  if (CMNA_partition_size != 4)
  {
    printf("WARNING: this test set up for 4 pe's in partition\n");
    printf(" No verification will be performed for this run!\n\n\n");
    verification = 0;
  }
  printf("Testing Combine network \n");

  verify(0, "Add scan forward");

  verify(1, "Max scan forward");

  verify(2, "Or scan forward");
```

```
  verify(3, "Uadd scan forward");

  verify(4, "Xor scan forward");

  verify(5, "Add scan backward");

  verify(6, "Max scan backward");

  CMNA_com_receive(&result);
  printf("Scalar received %d from add reduce\n", result);

  verify(7, "Add scan reduce");

  /* make sure everyone has synched up */
  CMNA_com_receive(&result);

  if (verification)
  {
    if (!failurep)
      printf("Tests ran successfully!\n");
    else
      printf("Tests FAILED!\n");
  }
  else printf("No verification was performed. Verify results manually\n");
}

/* Here's the PE stuff */

CMPE_pe_scan_test(arg1, arg2, arg3)
int arg1;
char arg2;
int arg3;
{
  int me = CMNA_self_address;
  unsigned int in;
  unsigned *scan();
  unsigned result;
  int    words_to_send;

  if ((arg1 != 1) || (arg2 != '2') || (arg3 != 3))
  {
    printf("ERROR: arguments are not being passed correctly\n");
    exit(-99);
  }

  srandom(me+time(0));
  in = CMNA_self_address + 10;
```

```
  CMNA_set_segment_start(random() % 2);
  CMNA_set_segment_start(0);

  words_to_send = sizeof (in) / sizeof (int);

  while (WAITING)
    if (CMNA_participate_in(NI_REDUCE_RECEIVE)) break;

  CMNA_global_sync((unsigned) TRUE);

  CMNA_com(ADD_SCAN, SCAN_FORWARD, &in, words_to_send, &result);
  SEND_RESULT_TO_SCALAR(result, COMBINE_OVERFLOW(CMNA_com_status()));

  CMNA_com(MAX_SCAN, SCAN_FORWARD, &in, words_to_send, &result);
  SEND_RESULT_TO_SCALAR(result, COMBINE_OVERFLOW(CMNA_com_status()));

  CMNA_com(OR_SCAN, SCAN_FORWARD, &in, words_to_send, &result);
  SEND_RESULT_TO_SCALAR(result, COMBINE_OVERFLOW(CMNA_com_status()));

  CMNA_com(UADD_SCAN, SCAN_FORWARD, &in, words_to_send, &result);
  SEND_RESULT_TO_SCALAR(result, COMBINE_OVERFLOW(CMNA_com_status()));

  CMNA_com(XOR_SCAN, SCAN_FORWARD, &in, words_to_send, &result);
  SEND_RESULT_TO_SCALAR(result, COMBINE_OVERFLOW(CMNA_com_status()));

  CMNA_com(ADD_SCAN, SCAN_BACKWARD, &in, words_to_send, &result);
  SEND_RESULT_TO_SCALAR(result, COMBINE_OVERFLOW(CMNA_com_status()));

  CMNA_com(MAX_SCAN, SCAN_BACKWARD, &in, words_to_send, &result);
  SEND_RESULT_TO_SCALAR(result, COMBINE_OVERFLOW(CMNA_com_status()));

  CMNA_com(ADD_SCAN, SCAN_REDUCE, &in, words_to_send, &result);
  SEND_RESULT_TO_SCALAR(result, COMBINE_OVERFLOW(CMNA_com_status()));

  while (WAITING)
    if (CMNA_abstain_from(NI_REDUCE_RECEIVE)) break;

  CMNA_com_send(ADD_SCAN, SCAN_REDUCE, &in, words_to_send);
}

SEND_RESULT_TO_SCALAR(result, overflow)
unsigned int result;
int overflow;
{
  int pe = 0;

  while (pe < CMNA_partition_size)
```

```
  {
    CMNA_write_int_to_scalar(result);
    CMNA_write_int_to_scalar(overflow);
    pe++;
  }
}
```

The following file is all of the pe code
combine_test_pe.c

B Compiling CandC for execution on simulators and hardware

It is highly recommended to run code on the simulator until it works correctly there. In fact, that is the only choice you have right now!

The simulators are being designed such that code written in CandC will not have to be modified in any way to run on the simulator, and also run on the hardware. The only difference in preparing code to run on the simulator versus preparing code to run on the hardware is to re-compile the code with a different CandC.h, and then link it with the appropriate library. The step of re-compiling with new CandC.h is only necessary because some of the CandC functions are macros.

Compiling the two example programs to run on the process simulator. This simulator provides a C/Saber debug environment for the scalar as well as the pe's in the partition.

```
cc -O -o combines combines.c -lpsim

cc -O -o combines_pe combines_pe.c -lpsim
```

It is more likely that the code will not run correctly at first and will have to be debugged. To debug this code efficiently, you must be familiar with Saber C. Copy saber init file /p1/cm5/psim/testsuite/.saberxwindows to your working directory, invoke saber, and load in the two files. You can type 'run' now, and a partition of four pe's will be established. See /p1/cm5/psim/doc/psimtutorial for more on this.