A TMS320C30-Based Digital Signal Processing Board for a Prototype NuMesh Module

by

Karim M. Abdalla

B.S., Massachusetts Institute of Technology (1988)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1990

Signature of Author ....
Department of Electrical Engineering and Computer Science
May 6, 1994

Certified by ....
Stephen A. Ward
Professor, Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

A TMS320C30-Based Digital Signal Processing Board for a Prototype NuMesh Module

by
Karim M. Abdalla

Submitted to the Department of Electrical Engineering and Computer Science
on May 6, 1994, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

The backplane bus is an inadequate mechanism for supporting interprocessor communication in large multiprocessor systems. The principle shortcoming of the backplane bus is that its communication bandwidth does not scale with the number of communication nodes attached to it. Project NuMesh is a proposal for scalable, modular, three-dimensional interconnect that is intended as a communications substrate for large multiprocessor systems using a standard set of interfaces.

A NuMesh system thus consists of an array of modules plugged into each other. Each NuMesh module consists of a standard communications processor (CP), and an unspecified local processing element (PE). The CP is responsible for routing data between itself and the identical CPs of adjacent modules, while the PE is responsible for data processing within the application. Since each PE can communicate with it's local CP, all PEs can communicate with each other via the network.

The focus of this document is the development of a specific TMS320C30-based PE for a prototype machine based on a two-dimensional version of the CP. The TMS320C30 digital signal processor (DSP) combines high scalar performance and many peripheral functions on a single chip, making it an ideal candidate for the implementation of a physically small uniprocessor board.

Thesis Supervisor: Stephen A. Ward
Title: Professor, Department of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Steve Ward for first getting me interested in digital electronics and subsequently supervising both my Bachelor's and Master's theses. Ricardo Jenez for encouragement, advice, dependable friendship, and last, but not least, for routing the board. Jim Olsen for soldering parts on the first board, and stomping on the ground bug. Many thanks to Christie Winterton for always being very helpful and efficient in sorting out administrative problem. Thanks also to John (Kubi) Kubiatowics and Andy Singer for always being ready to impart their computer wisdom unto me. The Rock of Boston, WBCN (104.1 FM) and Boston Phoenix, WFNX (101.7 FM) for making long typing hours less tedious. Mr. Coffee for keeping me up.

This TMS320C30 chips used in this project were kindly donated by Texas Instruments, Inc.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Communication in Parallel Processing

In any computation that can be viewed as a cooperative effort of multiple functional units, there will be data constraints and timing constraints. In the parallel processing world, these constraints are satisfied by adhering to protocols generally referred to as *communication* and *synchronization* mechanisms, respectively.

The amount of useful work done by individual functional units between synchronization or communication points is generally referred to as the *grain size* of the computation. The optimal grain size is normally a function of how much overhead is involved in inter-process communication and synchronization. For example, if there is a high overhead, then one would have to do a lot of work within a single functional unit, in order to justify the overhead of waiting for inputs to arrive from, or outputs to be consumed by, other functional units. The cost of having a large computation grain size is the serialization of that section of the computation, since it is by definition being executed by a single functional unit.

Figure 1-1 shows a hypothetical computation being shared by multiple functional units. One way of exploiting parallelism would be to start the computation at *F1* as soon as value A arrives, and at *F3* as soon as both A and B are present at its inputs. This example clearly illustrates that the computations at *F1* and *F3* can proceed concurrently. Additionally, *F3* represents a synchronization point for its inputs, since *F3* can proceed only when the two values "meet" in time.

Of course, internal to an individual functional unit, there may be additional parallelism exploited by *sub* functional units, but further analysis would be analogous to trying to understand a recursive program by attempting to trace each recursive call.

The essential difference between todays varied parallel programming models lies in the choice of grain size, and how the communication and synchronization issues are addressed. The success of a machine supporting a particular programming model depends on the implementation of computation, communication and synchronization mechanisms which appropriately reflect the model. Usually different programming models (and their implementations) are better suited for solving different classes of problems.

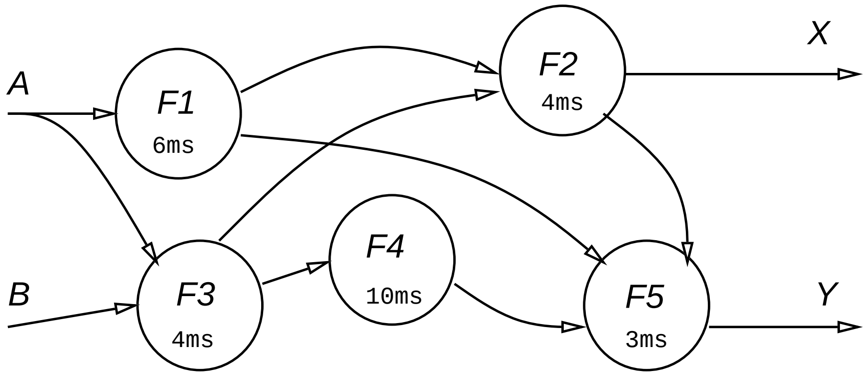

Figure 1-1: A Shared Computation

1.2 The Backplane Bus: Traditional Communications Substrate

1.2.1 Bus Appeal

Today, the majority of computer architectures rely on the backplane bus for supporting communication between functional units (or "modules"). The appeal of a well designed bus lies in its

- Simplicity — In a stripped-down bus, there is a set of wires for data and address (these may even share the same physical wires), and a single control signal specifying if the transaction is a read or a write. In such a bus, it becomes the responsibility of the module to decode the broadcast address in order to determine if the request is being directed at it.
- Modularity — New modules can be designed and plugged in directly into any bus, as long as they adhere to the standard (and hopefully simple) protocol for that type of bus.
- Extensibility — As long as the bus is physically long enough and has enough connectors, additional modules can be included simply by plugging them in. There is no need to redesign the bus or modify the existing modules.

1.2.2 Bus Limitations

For many architectures, where the number of modules is small, and the need for simultaneous transactions between different pairs of modules rare or nonexistent, the backplane bus serves its duty quite adequately. The limitations of the bus become apparent when the need arises for:

- Simultaneous communication between different pairs of modules — Multiple simultaneous transactions simply cannot occur since any transaction requires the entire bus as a resource. All inter-module communication must therefore be serialized. The resulting *communication bottleneck* in such situations is the main limitation of the backplane bus.

- Supporting a large number of modules — In order to add modules to a bus, the bus must be physically long enough to accommodate the width of module to bus connector. In order to support many modules, the bus has to be very long. As the bus gets longer, it must be treated as transmission line with an increasing maximum propagation delay. Furthermore, transmission characteristics are degraded by multiple branches to the specific modules. For this reason, all but trivially short buses have very limited bandwidth.

Usually, these two requirements are the two faces of the same coin, that is, adding more modules often implies increased inter-module communication needs. However, as more modules are added, the communication bandwidth capacity of the bus remains constant or may even degrade, rendering the bus an unviable solution for inter-module communication within large systems.

1.3 Summary of Chapter

Given that there are fundamental speed-of-light limitations on how fast a sequential computer can be, computers must exploit increasing amounts of parallelism in order to improve overall system throughput. However, exploiting such parallelism usually requires increasing amounts of intermodule communication. With this trend in mind, computer architects are finding the need for a high bandwidth, scalable communications substrate to replace the backplane bus.

The remainder of this thesis describes an alternative to the bus, and how such an alternative communications substrate is exploited to implement a prototype multiprocessor system. The emphasis of this thesis is on the individual processing element of the prototype system.

Chapter 2 introduces the NuMesh as an alternative to the bus, and Chapter 3 describes the general features of the prototype multiprocessor system based on the NuMesh. Chapter 4 details the development of the processing element subsystem for the prototype, while Chapter 5 describes the board design process and the results.

Chapter 2

The NuMesh

2.1 A Scalable, Modular, 3D Interconnect

This extract, and following one in section 2.4.1 are taken from the original NuMesh proposal. [1]

> ...the development of a communications substrate which affords Tinkertoy-set modularity and very high performance communication ...The approach involves standardizing the mechanical, electrical, and logical interconnect among modules arranged in a (partially populated) 3D mesh whose lowest-level communications follow pre-compiled systolic patterns. The attractiveness of the scheme derives from the separation of its communications and processing components, and the standardization of the interface between them. This decoupling of computation from the communication substrate allows the continued exploitation of mass-produced processing elements, e.g. contemporary signal processing chips.

2.2 Modules

A NuMesh system, as in a bus-based system, is composed of separate modules which have the ability to communicate with each other — except that the bus has been replaced by the mesh. As before, different modules can fulfill different functions such as computational, I/O, or graphics.

Each module is ideally a small (perhaps two-inch, but conceivably much smaller) cube, with a separate port connector on each face for interfacing with its six nearest neighbors. Figure 2-1 shows what a NuMesh system might look like. Of course, other topologies are also possible. Constructing a system out of such modules would only take a few hours to plug all the modules together, Tinkertoy-set style.

2.3 Standard Interfaces

As with bus-compatible modules, each NuMesh module is responsible for adhering to a set of standard, low-level communication protocols. Each module can be considered to consist of two

[1] The NuMesh: A Scalable, Modular, 3D Interconnect, draft, 2/15/89, **Ward, Dally, Agarwal.**

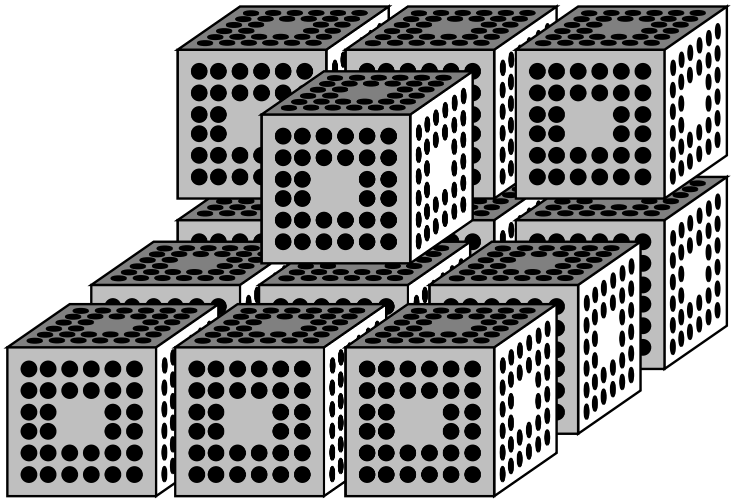

Figure 2-1: Artist's Conception of an Ideal NuMesh System

parts: a standard communications processor (CP), and an unspecified local processing element (PE). The CP is responsible for routing data between itself and the identical CPs of adjacent modules, while the PE is responsible for data processing within the application. Since each PE can communicate with it's local CP, all PEs can communicate with each other via the network.

Ideally, the NuMesh system developer would be able to buy a "standard" module that contains everything a NuMesh module should have, except for the local processor. Thus, the developer buys a small, cubic package, with a mesh connector on each face, and an instance of the standard communications processor. The standard module has a single internal socket labeled "your processor goes here". All that the developer needs to do is plug his specific processor into the internal socket, satisfying a very simple, but again, standard protocol.

The protocol for the afore-mentioned local processor / communications processor interface should be made very simple; certainly no more complicated than the protocol of a typical modern bus (such as the NuBus), and possibly as simple as that for a standard memory-mapped register.

The supplied communications processor is responsible for satisfying the more complex NuMesh inter-module protocol at the six ports. The common circuitry in the standard module is also responsible for a variety of other functions, such as trimming clock skew, and processor / communications synchronization.

2.4 Mesh Communication

Each communications processor is controlled by a *simple* [2] finite state machine, henceforth referred to as a CFSM. The entire array of CFSMs is synchronized to a global fast clock.

The CFSM clock period is trimmed down to the minimum time necessary to transfer a data word between adjacent modules. Minimizing inter-module distance thus relieves an important

[2] NuMesh takes a RISC attitude towards communication.

constraint on the the minimum clock period, further motivating the minimization of module size.

2.4.1 Static Communication Patterns

The lowest level of communication relies on each CFSM following a periodic, predefined sequence of I/O transfers, that normally [3] depends only on the relatively static CFSM transition table. The control store for the CFSM consists of RAM, which is loaded with CFSM code at boot time from the host processor via the network.

Mesh communication synchronization relies on each CFSM having a notion of real time i.e. how many clock periods have elapsed since reset.

> The general idea is that each module's communications FSM be programmed to follow a periodic pattern of interactions with neighbors. Although the interactions may vary among processors, the periods will be identical. If module A transfers a word to its right-hand neighbor B on clock 37 of each period, then A's FSM will be programmed to drive its lines to B on that clock, while B will be programmed to load in data from A. By appropriate design of transition tables, arbitrary systolic communications patterns may be implemented among processors. In some cases, words loaded by a module are destined to be read subsequently by that modules DSP; in other cases, they are routed (typically on the next clock) to another neighbor without DSP intervention or even awareness.

While certain limited connectivity algorithms (such as graphics and digital signal processing applications) may map directly onto NuMesh's lowest level communications mechanisms described above, general-purpose multiprocessing dictates the need for dynamic (i.e. run-time status dependent) routing.

2.4.2 Support for Dynamic Communication Patterns

Ultimately, we wish NuMesh to support dynamic message routing — for example, module A, on computing a result R, may decide to send R to module B if R is negative, or to module C if R is positive. Not only does module A have to decide where to send the result, but both modules B and C have to decide whether they are receiving valid data (either destined for themselves or to be forwarded to a neighbor that is closer to the ultimate destination of the data) or if they should carry on, ignoring their respective input lines from A for that cycle.

As can be surmised, dynamic routing involves more decision making, which, in order to support, requires either increased decision making time, increased decision making hardware, or both. In adhering to the RISC philosophy of NuMesh communication, we wish for a solution that requires minimal additional hardware, and does not significantly increase the CFSM clock cycle time.

If we assume that under most circumstances, communication patterns can be efficiently implemented over the underlying static routing mechanism, then we can justify building a machine optimized for static routing, while minimizing the cycle time. Following this assumption, we

[3] In order to provide hooks for dynamic routing, there may also be provisions for CFSM transitions that depend on the run-time system status.

can insert a minimal amount of hardware "hooks" so that in the rare situations that *do* call for dynamic routing, the system *can* make the dynamic decisions and actions, but possibly taking several clock cycles about doing so — perhaps due to the cost of flushing a pipeline.

2.5 Summary of Chapter

This chapter introduced NuMesh as a communications substrate that permits functional modules to communicate with each other. Not surprisingly, most of the emphasis has been on inter-module communication, rather than on module functionality. It is by design that NuMesh allows us to discuss the processing and communications aspects of modules separately.

Chapter 3

The NuMesh Prototype

Any useful computer system, NuMesh-based or otherwise, has to be able to do real work at each module. A complete NuMesh module thus consists of a processing element (PE) and a communications processor (CP). The NuMesh prototype is based upon a two-dimensional version of the network described in chapter 2.

Figure 3-1 shows how a prototype module consists of a PE board that plugs into a CP board. The CP connects to its four neighboring CPs via port connectors on the north, east, west and south edges of the board. There is an additional connector port facing up, which mates with a corresponding connector on the PE.

This chapter describes the general features of the prototype NuMesh module; the processor of choice, the general features of the communications board (and hence the network itself), and the interface between them.

3.1 The TMS320C30 Digital Signal Processor

3.1.1 Choice of Processor

The choice of processing element in a parallel computer is related to a number of factors such as the computation grain size, price, board real estate, and power and cooling requirements. The following factors helped in making the choice of processing element for the prototype NuMesh machine:

- **DSP** — We would like the prototype system to display it's potential for very high system throughput, which means that it should be programmed with applications that can directly exploit the static routing mechanism. Typically digital signal processing (DSP) applications fall into this category. Thus, given that the Speech Processing and Computer Architecture groups at M.I.T. are in close communication with each other, the first NuMesh applications will most likely be in digital speech processing.

- **Compact** — The processor board and communications board should be about the same size. The initial estimate for either prototype board was a square whose sides were approximately four inches long. The board size constraint dictated a the use of processor that needed minimal peripheral hardware support.

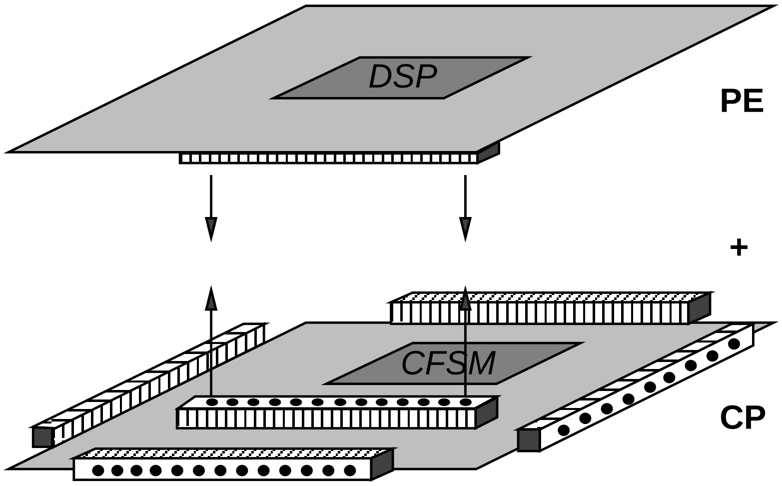

Figure 3-1: A Prototype NuMesh Module

- μ**Processor** — Although network communication is "fast" (suggesting "fine" grain computation to the naïve reader), it is constrained by the static routing hardware. This restriction suggests that "medium" grain computations (i.e. those requiring some internal dynamic communication) be kept within a single general-purpose microprocessor with sufficiently flexible addressing modes.

- **Simple** — We wanted a simple, reliable, easily debugged design so that we could get the prototype machine up and running the minimum amount of time.

- **TMS...** — Relations of this research group with with Texas Instruments, Inc.

Taking the previous factors into account, Texas Instruments' TMS320C30 digital signal processor (DSP) was the chip chosen for basing the prototype NuMesh processing element around.

3.1.2 General Description

The TMS320C30 was chosen because it is a fast scalar processor, optimized for digital signal processing applications, yet is flexible enough for general-purpose uniprocessing. The chip features a fast cycle time, and a pipelined execution unit that can execute most instructions (including parallel instructions, which are, in effect, two concurrent normal instructions) in a single cycle. The execution unit can always be kept busy thanks to internal memory, instruction cache and multiple internal (and external) busses.

In addition to the floating point registers, the processor possesses a general-purpose register file, large address space, powerful RISC-style instruction set, and flexible addressing modes that are supported by separate addressing hardware.

The TMS320C30 also includes many on-chip peripheral devices, making it ideal for implementing an entire system without the need to add many external devices. The processor's bus interface is very simple and supports internal wait state generation, making it possible to add external devices using little additional interface logic.

Of particular noteworthiness is the emulation port on the TMS320C30. This set of pins allows bidirectional access to a serial scan path that includes the entire state of the processor. The emulation port can be connected to a card on an IBM-compatible PC to provide a DSP debugging environment, using the PC as the user interface. Furthermore, DSP data can be transferred between the DSP's memory system and the PC, making the test code development cycle trivially fast, without the need for any off-chip I/O hardware.

This section is intended as a superficial overview of the TMS320C30. For a more complete description of the TMS320C30 please refer to the user's guide. [1] The following subsections borrow freely from the user's guide.

3.1.3 Key Features

Some of the key features are summarized below.

- 60-ns single-cycle instruction execution time
  - 33.3 MFLOPS (million floating-point operations per second)
  - 16.7 MIPS (million instructions per second)
- Two 1K $\times$ 32-bit single-cycle dual-access on-chip RAM blocks
- 64 $\times$ 32-bit two-way set associative instruction cache
- 32-bit instruction and data words, 24-bit addresses
- 40/32-bit floating-point/integer multiplier and ALU
- 32-bit barrel shifter
- Eight extended precision registers (accumulators)
- Two address generators with eight auxiliary registers and two auxiliary register arithmetic units
- On-chip Direct memory Access (DMA) controller for concurrent I/O and CPU operation
- Integer, floating-point, and logical operations
- Orthogonal two- and three-operand instructions
- Dual parallel ALU, multiplier and load/store instructions in single cycle
- Block repeat capability
- Zero-overhead loops with single-cycle branches
- Conditional calls and returns
- Interlocked instructions for multiprocessor support

[1] Third-Generation TMS320 User's Guide, Copyright © 1988, **Texas Instruments Incorporated**

- Two serial ports to support 8/16/32-bit transfers
- Two 32-bit timers
- Two general-purpose external flags, four external interrupts
- 180-pin grid array (PGA) package; 1 μm CMOS

3.1.4 Architectural Overview

Central Processing Unit

The heart of the Central Processing Unit (CPU) constitutes the multiplier and Arithmetic / Logic Unit (ALU) with barrel shifter. Both units have single-cycle latencies. The multiplier performs multiplications on 24-bit integer and 32-bit floating-point operands, yielding 32-bit and 40-bit results respectively. The ALU performs 32-bit integer arithmetic / logical operations, 40-bit floating-point operations, and conversions between the two formats.

Four internal busses, CPU1 & CPU2 and REG1 & REG2, can each carry two operands from memory and two operands from the register file, thus allowing parallel multiplies and adds/subtracts on four integer or floating-point operands in a single cycle.

Two auxiliary register arithmetic units, ARAU0 & ARAU1, can generate two addresses using a number of addressing modes in a single cycle, in parallel with the multiplier and ALU.

The TMS320C30 provides 28 registers in a multiport register file. All of these registers can be operated upon by the multiplier and ALU. Except for the extended precision registers, which are 40 bits long each, all registers are 32 bits long. The register set comprises the following general and special-purpose registers.

- R0–R7 — Extended precision registers
- AR0–AR7 — General-purpose auxiliary registers
- DP — Data page pointer which makes up the address MSBs of directly addressed data. This is necessary since immediate data words (including direct addresses) are only 16 bits long.
- IR0, IR1 — index registers used in some addressing modes
- BK — Block size for circularly addressed blocks
- SP — Stack pointer
- ST — Status register
- IE — CPU/DMA interrupt enable register. This is often referred to as the interrupt *mask* in common microprocessors — interrupts are globally enabled by a bit in the ST.
- IF — CPU interrupt flags. A 1 in a particular bit position indicates that the corresponding interrupt is set. Traps can be implemented by writing to this register, and interrupt lines can be polled by reading it.

- IOF — I/O flags register, which specifies the direction of the dedicated external pins, XF0 and XF1, and allows their values to be read and written.
- RS, RE, RC — Repeat start address, end address and counter. RS and RE delimit a block of code that is executed RC times.
- PC — Program counter

Memory Organization

The total address space of the TMS320C30 is 16M (million) 32-bit words. All addresses and memory sizes mentioned refer to 32-bit words (i.e. "1K" refers to 2^{10} 32-bit words). Program, data and I/O space are all contained within this 16M-word address space.

In addition to the 64-word instruction cache, the TMS320C30 has two internal 1K RAM banks, each capable of two accesses per cycle. This is due to the fact that what is normally referred to as a "cycle" is made up of two minor cycles of the raw clock input. The TMS320C30 also sports 4K of internal ROM, but it cannot be used for our purposes, since its contents are neither defined nor field-programmable.

Figure 3-2 shows how the memory space is organized on the TMS320C30.

Internal Bus Operation

Three internal busses connect the memory system to the Instruction Fetch Unit (IFU), the CPU, and the DMA respectively. The three internal busses are bidirectionally multiplexed onto the two external busses. The TMS320C30s multiple internal (and external) busses, coupled with well-tuned code that makes judicious use of the I-cache and internal RAM blocks, allow the IFU, CPU and DMA to operate concurrently. Under most circumstances, concurrent operation of these units can proceed without any one of them being starved for memory access. Memory conflicts are handled automatically according to a prioritized list of data consumers.

External Bus Operation

The TMS320C30 provides two interfaces to external busses, the primary bus and the expansion bus. Both consist of a 32-bit data bus and a set of control signals. The primary bus has a 24-bit address bus, whereas the expansion bus has a 13-bit address bus. Both busses can be used to address external program/data memory or I/O space. Both busses have an external $\overline{(X)RDY}$ signal for external wait state generation. Additional wait states may be inserted under software control. Refer to section A.1.1 for details of bus timing.

Peripherals

The TMS320C30 peripherals include two internal timers and two serial ports. All CPU communication (control, status, and data) with the peripherals is via memory mapped registers on an internal peripheral bus. The only exception to the above is the TMS320C30's interrupt mechanism.

The two timers are general-purpose 32-bit timers/event counters. Each timer can be configured to count internal clock transitions, or external events, on its respective TCLK pin.

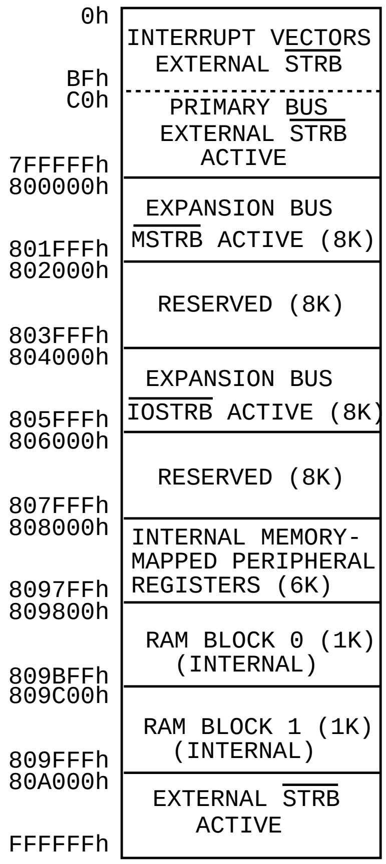

Figure 3-2: TMS320C30 Memory Map

Conversely, the timer can output a periodic signal on the pin. The pin may also be used for general purpose I/O.

The two identical serial ports can be configured to transfer 8-, 16-, 24-, or 32-bit words using internal or external baud rate generators. The serial ports can also be configured as timers and the associated external pins can be configured as general-purpose inputs or outputs.

Direct Memory Access

The on-chip Direct Memory Access (DMA) controller can bidirectionally access any location in the memory map without interfering with the operation of the CPU. The DMA controller possesses its own transfer counter and address-generation hardware. A dedicated DMA address and data bus allow for concurrent operation of DMA and CPU. Concurrent operation of CPU and DMA is transparent to the user, even in the case of simultaneous attempts to access to a single resource. In this case, internal hardware gives the CPU priority by stalling the DMA until the resource is freed.

3.2 Prototype Communications Processor

This section describes the general features of the prototype communications processor (CP). Much of the prototype CP's control paths are implemented in programmable array logic (PAL) devices. There are two main reasons for this. First, chip count is reduced by implementing random logic and auxiliary finite state machines in PALs. The second reason pertains to the idea of prototyping: since these devices are reprogrammable, we maintain a certain flexibility in the lower levels of inter-module and DSP / CFSM communications protocols.

This section is included to provide additional context for the processing element, and is not intended to fully document the communications processor. A more complete description of the CP is given in a related document.[2]

3.2.1 Data Paths

Ideally, we would like to be able transfer a data word between any two ports in a single clock cycle. Since each CP has five ports (North, South, East, West, and Up), it should also be possible to have a pair of simultaneous transactions within a single CP. To maximize network bandwidth, we could also require that each port be able to support a pair simultaneous transactions, one in each direction. Such an ideal CP data path requires $5! \times 2 = 240$ internal data busses, and a proportional number of signals to control all permutations of transactions between them.

The prototype CP trades off some of the performance of an ideal CP-internal network by dispensing with all but two busses, thereby greatly reducing the amount of wires that would be required on the board, consequently permitting a much more compact implementation.

The main data paths of the CP consist of a north-south bus and an east-west bus. The two busses are connected via a registered transceiver pair. [3] Additionally, there is a transceiver at

[2] NuMesh Prototype Hardware Description, 12/16/89, **Ken Mackenzie**

[3] The transceivers used on the prototype CP are Fairchild's 29F52 devices. Internally these parts consist of a pair of back-to-back octal registers with separate clocks, load-enables, and output-enables. Since all NuMesh

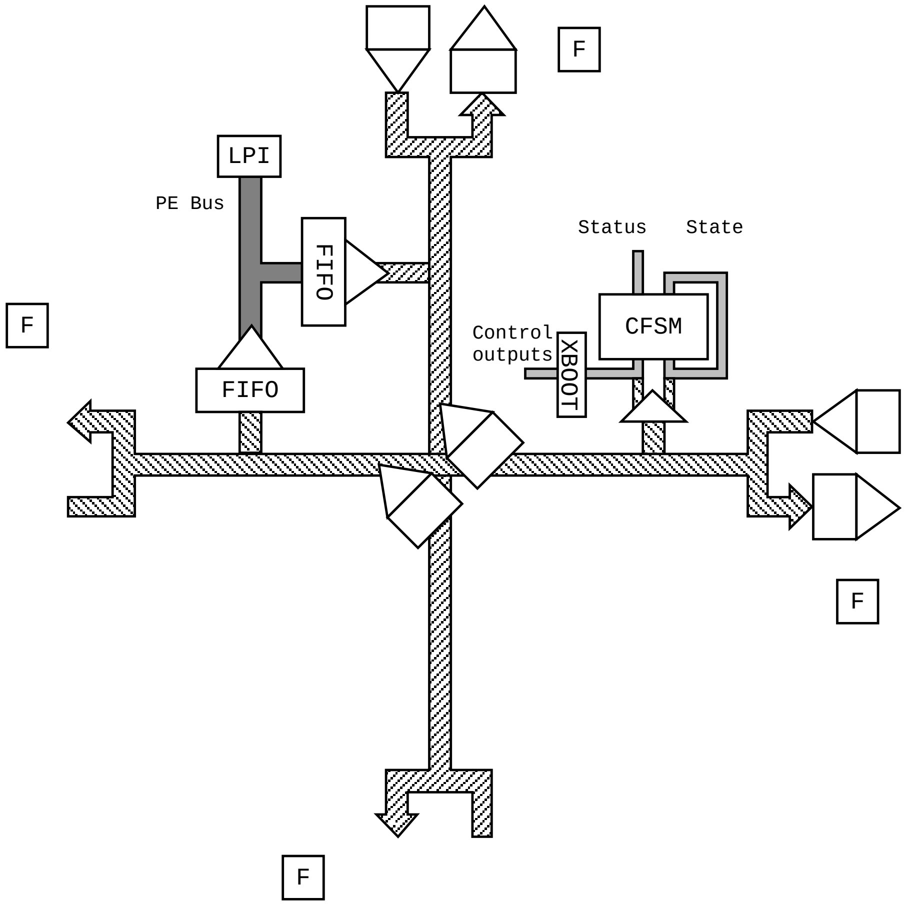

Figure 3-3: Communications Processor Basic Data Paths

the boundary between any pair of adjacent CPs. By placing transceivers at the north and east edges of each physical CP board, a connected array of CPs will have exactly one transceiver on each CP-to-CP boundary.

Associated with both registers of each boundary transceiver is a full bit, which is asserted if it contains valid data. The full bit is set whenever the data register is loaded, and cleared when it is read (outputs enabled). The full bits are status inputs to the CFSM, which can use them to provide some run-time error detection. Physically, each CP board houses only the full bits of it's outgoing registers.

Finally, there is a 32-bit wide FIFO bank on each of the two busses. One of the FIFOs has its inputs on one of the busses, and its outputs on the LPI connector, while the the other FIFO has its inputs connected to the same LPI connector data pins, and its outputs connected to the other bus.

Figure 3-3 shows the basic data paths of the communications processor. For clarity, the

communication is synchronous, all transceiver clock inputs are theoretically (i.e. ignoring clock-skew, and / or the clock distribution scheme) the same signal. Since both of these busses on all CPs are 32 bits wide, four 29F52s in parallel are required between any pair of connected busses.

control paths are omitted from this figure. This topology thus allows simultaneous unidirectional single-cycle transfers on each of the orthogonal busses. In order to "turn a corner", a data word has to first be loaded into the cross transceiver, requiring a total of two clock cycles for the transaction. Presumably, the common case does not involve use of the cross transceiver, and can thus be effected in a single cycle.

3.2.2 Network Reset and Boot

Booting the network [4] involves loading all the CFSM transition tables into the SRAM control stores on the respective CP boards. The control words are supplied to the CFSMs from the front-end processor via the (partially booted) network. For this purpose, the lowest 16 bits of the east-west bus are connected to the data pins of the control store SRAMs.

A simple boot FSM, "XBOOT", is implemented in a PAL to permit data transfer from the east-west bus to the control store of the CFSM. It is obvious that the CFSM cannot be used to control the data paths at this time, since its transition table is undetermined, and data is being *written* into the control store.

The full bits described in the data paths subsection above are most useful during this booting sequence. Used as inputs to XBOOT, the booting process can rely on the full bits as synchronization flags, permitting a deterministic result of the boot process that is independent of the inter-arrival times of words being supplied by the front-end processor. A special word must be supplied to indicate the end of the boot code for the current CP, and that control should be transfered from XBOOT to the CFSM.

Once the first CP is booted, it can start forwarding data words (as specified by it's current program) to it's neighbors, which are still being controlled by their own XBOOT as the first CP was. This system of booting and then forwarding words to neighbors would eventually allow the entire array of CFSM to be programmed.

Resetting the network thus entails placing all CPs in a state where each is being controlled by the respective XBOOT FSM, which is waiting to load the CFSM control store as words arrive (according to the full bits) and eventually transferring control over to the CFSM. Finally, booted CPs can start forwarding code to their local PE.

3.2.3 Static Routing

The prototype CP is designed to support the very simple static routing described in section 2.4.1 fairly efficiently. However there are two primitive mechanisms that may allow the network to benefit from run-time status.

First of all, full bits can be used for error detection (but not correction). In the event of an error, the CP would somehow signal the error to the outside world, possibly via the PE. For this purpose, two CP status signals, $\overline{CFSMRun}$ and $\overline{Misc}$ are provided as miscellaneous inputs / interrupts to the PE to provide some dynamic information about the network status.

This mechanism can detect errors where the static CFSM code dictates that a CP write to it's neighbor, but finds that it's neighbor input register is already full, or conversely, reading an input register and discovering that it is not yet full.

[4] Booting the network is a non-trivial matter which is the subject of a separate study by Rajeev Dujari. Included here is only the outline of an early proposal.

Full bits might also find some use in loosening the very tightly synchronized communication patterns described in section 2.4.1, at the cost of some network bandwidth. For instance, if a sending CP writes to an output register slightly (i.e. a few clock cycles) later than scheduled, the receiving CP could "spin" until the full bit is asserted, before actually reading the register.

This system would work if the receiving CP would not otherwise be busy while spinning, and if the delay caused by one CFSM spinning will not cause it to be permanently out of synch. With appropriate CFSM code, these conditions can be satisfied, at the expense of communication delay.

The other primitive mechanism that can have a dynamic effect on communication patterns, is a pair of signals, $\overline{Misc0}$ and $\overline{CFSMReset}$, outputs of the PE that are passed along the LPI connector as inputs to the CFSM on the CP. The two signals, $\overline{Misc0}$ and $\overline{CFSMReset}$, are connected respectively to XF1 and XF0, which are general-purpose I/O pins on the TMS320C30.

With CFSM code that monitors these inputs, the PE can interrupt the CP, and even download new CFSM code to dynamically reprogram the CFSM. To allow the PE to reprogram the CFSM, the latter has to be able to pass control of the CP back to XBOOT. Such a mechanism can be used to implement an inefficient form of dynamic routing, using the PE as the intelligence behind the router.

The signal $\overline{Misc0}$ is intended for the above dynamic routing mechanism, i.e. causing XBOOT to load new CFSM code from the PE. The $\overline{CFSMReset}$ signal is intended as a higher priority interrupt to the CP, to be used in case of a catastrophic system failure, requesting a cold-start by indicating to XBOOT to load CFSM code from the network as described in section 3.2.2.

3.3 Local Processor Interface

This section describes the "*simple* local processor / communications processor interface", which will which will henceforth be referred to as the Local Processor Interface (LPI).

One of the reasons for building a prototype is to evaluate various proposals for different protocols by first placing the appropriate hardware "hooks" and then varying only the control software/firmware appropriately. This section presents some low-level hardware mechanisms, and how they might be used to support higher-level LPI protocols and programming models.

3.3.1 Totally Synchronous Communication

The simplest protocol assumes that the concurrently running DSPs *and* CFSMs are totally synchronous, and that each DSP has explicit instructions within the application code that read/write a single word at a time from/to the local CP *at exactly the right time*.

Such a protocol implies that each DSP assumes that when the time comes to write a word to the communications processor, the latter will be ready to accept the data, and not otherwise busy. Similarly, when the DSP wants to read from the communications processor, it assumes that the data will be there.

When writing to the CP, the DSP cannot be too late, since the CFSM is blindly programmed to read a word from the LPI and forward it within the net. So far, we have not specified any validity-checking on anything that the CP reads. This is partly due to our inability to gracefully cope with errors since they essentially constitute dynamic communications requests. We will refer to a hazard of this type as DSP Write too Late (DSPWL).

When writing to the CP, the DSP cannot be too early in writing to the PE either, since, unless the CP can buffer the data, the data will be lost. This hazard will be referred to as DSP Write too Early (DSPWE).

Conversely, when the DSP reads from the CP, it cannot do so too early, or it will read invalid data. However, it would be easy for the DSP to do some some validity-checking in software (e.g. parity) on data that it reads. Unfortunately, even though the DSP may be able to detect an error, it cannot effectively deal with the problem (request a retransmit, for example). This is the DSP Read too Early (DSPRE) hazard.

Again, unless there exists some buffering, the DSP cannot read too late. We will refer to this hazard as DSP Read too Late (DSPRL).

Such a tight coupling between DSP and CP implies that the two elements be totally synchronous. Although it would be very easy and cheap to implement in hardware, this model represents a compilers nightmare because the execution time of the DSP code has to be determined down to the cycle at compile time. Compiling code for an array of DSPs and CPs would be close to impossible.

3.3.2 Buffering

We can relieve some of the afore-mentioned dependencies, by providing some hardware buffering between the two elements. The appropriate use of buffers also permits asynchronicity between the the clocks of the DSP and the CP.

We can all but eliminate DSPREs by intentionally programming the DSP to read the CP buffer *well after* we expect the CP to receive the data from the network. The danger of waiting too long is that the buffer might overflow, resulting in an effective DSPRL. We can reduce the chance of a DSPRL by increasing the buffer size.

Similarly, buffering writes to the CP would allow the DSP to write to the buffer without worrying too much about a DSPWE. Unfortunately, no amount of buffering can remedy a DSPWL, since the CP cannot wait.

The main disadvantage of relying on buffers is the increased effective network latency due to waiting. However, the DSP can still do useful work that does not depend on the data to be read to make up the waiting time.

3.3.3 Full Bits

Another way to deal with DSPREs is to provide a full bit along with the data presented to the DSP. The DSP could then wait on this signal until the data becomes available, whereupon the data is read and the full bit cleared. This eliminates the latency associated with waiting a safe time for the buffer to be ready, since the data is read from the CP as soon as it becomes available.

Similarly, the DSP can present a full bit along with data to the CP. The CP should clear this bit when it reads the buffer. It becomes the responsibility of the DSP to monitor this bit to determine if it may write to the CP.

One of the drawbacks of monitoring full bits, is the time that might be wasted by polling the status of the bits in software. There are two basic hardware mechanisms that can be added to alleviate the software overhead.

- *Extended Read and Write Cycles* — The TMS320C30 external busses can be configured to extend access cycles until the corresponding $\overline{(X)RDY}$ signal is asserted. One can therefore insert additional hardware to make access to the CP transparent to the DSP programming model, which simply assumes that every access is successful. Attempts to read an empty buffer or write to a full buffer are extended until the appropriate buffer is ready.

- *Interrupt Driven* — The incoming and outgoing buffer status bits can be wired to the interrupt lines of the DSP. In an event-driven protocol such as this, the DSP can take the appropriate action based on *when* the buffer becomes ready. The event-driven programming model may be more complicated from the programmers standpoint, but it does not suffer the possibility of the wasted cycles due to extended accesses.

3.3.4 Direct Memory Access

Both of the two previously-mentioned mechanisms suffer some CPU delay. In the case of extended access, the problem lies in stalling the CPU until the access can be completed successfully. For the interrupt-driven case, dispatching to an event handler invariably involves some overhead.

Direct Memory Access (DMA) is a mechanism that allows the transfer of words, or blocks of words, between specified source and destination addresses. Usually DMA refers to a dedicated hardware module that can operate concurrently with the CPU, without slowing it down. The TMS320C30 has an on chip DMA module, so reaping the benefits of this mechanism does not involve any additional hardware.

The DMA can be used to transfer data between the CP and the DSPs RAM, without slowing down the DSP. DMA can be used in conjunction with either the extended access or interrupt-driven mechanisms; the TMS320C30 can be configured so that an interrupt triggers a DMA transfer without *any* CPU intervention.

3.4 Summary of Chapter

This chapter gave a system-level description of the prototype NuMesh module consisting of a processing element interfaced to a communications processor. Figure 3-4 shows how copies of this module will be connected together to yield a parallel computer. The remainder of this thesis gives a detailed account of the design and development of the processing element hardware.

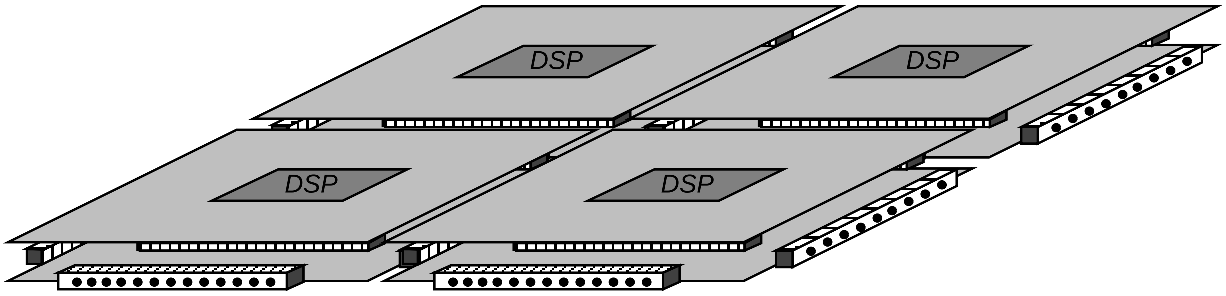

Figure 3-4: Prototype NuMesh System

Chapter 4

The Processing Element Board

4.1 What Should Go on the Board?

The TMS320C30 provides the majority of the functionality required of the processing element. This fact was a significant force affecting the choice of processor. However, there remains a need for some off-chip logic, namely for processor clock generation, providing more memory, supporting the local processor interface, and improving the testability of the overall system.

4.1.1 Clock Generation

The TMS320C30 is guaranteed by the manufacturer to run at a clock rate of up to 33.3 megahertz. To benefit from the maximum rating, the board is provided with a 11.1 megahertz crystal in a third overtone filter circuit that supplies the on-chip oscillator with a 33.3 megahertz signal.

To provide more flexibility, the board also comes with a jumper that provides the option of using an off-board oscillator that is wired to the LPI connector.

4.1.2 RAM

Although the TMS320C30 comes with 2K words of RAM, this amount might be insufficient for real (or experimental) applications. It is therefore desirable to augment the internal memory by adding an external RAM bank. Thus the PE board is provided with a single bank of zero-wait-state static RAM in the form of four CY7C185 35ns access-time 8K $\times$ 8-bit chips.

Having an external bank would permit DMA block transfers between CP buffer and external RAM, without interfering (by competing for memory access) with a program executing entirely out of internal RAM. Aside from the obvious speedups provided, such use of RAM would make execution times less dependent on communication synchronization.

4.1.3 ROM

Most system find it very useful to possess a small amount of non-volatile or read-only memory. The main use of ROM is for storing the lowest level of bootstrapping code. Often the function of the boot code is to initialize the system into a state where it can load the next level of code from secondary storage. For traditional systems, secondary storage refers to the disk storage subsystem. For NuMesh, the next level of code would be supplied by the front-end processor via the network.

See sections 4.2.1, 4.2.2, and 4.2.3 for a more in-depth discussion of the NuMesh PE bootstrapping process, and why some ROM storage is desirable.

Other uses of ROM include the ability of the PE to do some self-diagnostics on power up, and consequently to ease debugging at that time. The ability of the subsystems to independently test themselves can be used to provide useful debugging information in the case of a macrosystem failure.

The PE board is thus provided with a single bank of zero-wait-state ROM in the form of four CY7C185 35ns access-time 2K $\times$ 8-bit chips. One feature of the primary bus interface is the ability to accomodate slower ROMs (as EPROMS tend to be).

4.1.4 Local Processor Interface Controller

Large buffers, full bits, extended access, interrupts and DMA can be combined in various ways to yield the compound benefits of these mechanisms (discussed in section 3.3). Without à priori knowledge about the precise nature of the intended programming model, it seemed prudent to implement all of these mechanisms directly. Fortunately, much of the required hardware already exists on-chip on the TMS320C30; the bulk of the the remaining task being to appropriately connect the relevant pins on the TMS320C30, the CP, and the extra logic necessary for implementing extended access.

FIFO Buffers on the CP

Although the LPI buffers are physically on the CP board, the PE board designer must have a good understanding of the devices in order to use them.

The devices chosen were Cypress' CY7C429-40 asynchronous 2K $\times$ 9-bit FIFO (first in, first out) buffers. Since the LPI data word width is 32 bits, four devices in parallel are necessary for each of the two directions. The devices have separate read and write control lines, as well as empty and full status lines.

A buffer read cycle is initiated by strobing the $\overline{R}$ (*read*) line, moments after which, the outputs change from a high-impedance state to the data value. Further more, if the datum was the last item in the buffer, the $\overline{EF}$ (*empty flag*) is asserted by the device until another word is written to the FIFO.

A buffer write cycle is initiated by placing the data at the inputs and strobing the $\overline{W}$ (*write*) line. If the datum written results in the FIFO reaching full capacity, the $\overline{FF}$ (*full flag*) gets asserted by the device, until the FIFO is read.

Figure 4-1 shows how the main signals are connected to the two "sides" (i.e. PE or CP) of the buffers. Each side is connected to the output of one FIFO bank (the *read*-buffer for that side) and the input of another FIFO bank (the *write*-buffer for that side). Note that since the PE uses the the same data bus for reading and writing, the outputs of the read-buffer are connected to the inputs of the write buffer. In fact, both buffers are mapped at the same address, using TMS320C30's $R/\overline{W}$ to select between the read and write buffers. The control and status signals of interest to the PE are the read-buffer's $\overline{R}$ and $\overline{EF}$ lines, and the write-buffer's $\overline{W}$ and $\overline{FF}$ lines.

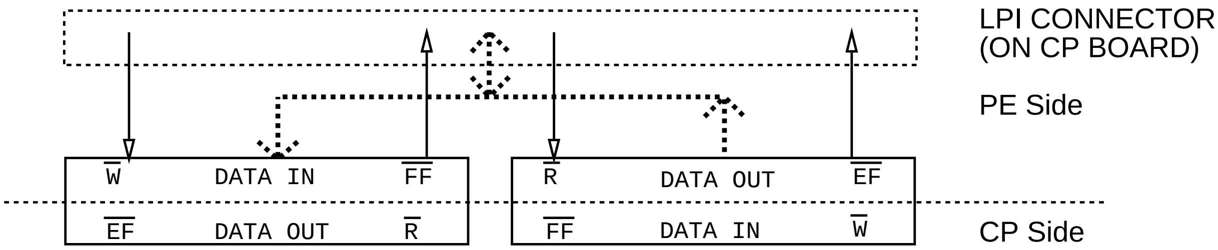

Figure 4-1: FIFO Signal Ownership

- **CP-Bound FIFO Full** — The PE uses the $\overline{FF}$ signal for two main functions. After being synchronized by a flip-flop within a PAL, the signal gets renamed to $\overline{FFS}$. The synchronized version of the signal is connected to one of the TMS320C30's interrupt pins, and an LED indicator. The interrupt might be used by the software as a warning that the PE is ahead of synch with the communications network, and to take appropriate action, such as simply waiting for a while. The LED is a visual debugging aid. The second main function of $\overline{FFS}$ is as an input to the FIFO finite state machine controller, which uses it as part of the extended access implementation.

- **PE-Bound FIFO Empty** — The PE uses the $\overline{EF}$ in a similar fashion. First, the raw signal is synchronized by an external flip-flop, and renamed to $\overline{EFS}$. The synchronized version of the signal is then used as an input to the PAL controller for the purposes of the extended access mechanism. One of the state bits signifies that the controller is in the FIFO-empty state, and this pin is tied to an LED indicator.

  The inverted output of the synchronizing flip-flop is tied to one of the interrupt pins on the TMS320C30. The sense of the interrupt signal is devised such that the processor can be interrupted as soon as data becomes available (i.e. the FIFO is *not* empty.

FIFO Finite State Machine Controller

The standard (i.e. $\overline{STRB}$ and $\overline{MSTRB}$ active) TMS320C30 bus access cycle [1] is initiated by first setting up the address lines, selecting between a read or a write operation by the polarity of $(X)R/\overline{W}$, and setting up the data lines in the case of a write. Well after all these signals are stable, the TMS320C30 asserts a $\overline{(M)STRB}$ signal to strobe the data into or out of memory.

However, as explained earlier, FIFO read and write cycles are initiated by asserting the $\overline{R}$ or $\overline{W}$ pins respectively. It is the falling edge of the appropriate signal that updates the internal queue pointer. Thus consecutive FIFO accesses require successive *edges*. This differs from consecutive standard reads where only the address lines change, while $\overline{(M)STRB}$ is continuously asserted throughout the sequence.

Clearly then, some translation logic is needed between the TMS320C30 bus control signals and the FIFO control signal. Further more, to implement extended access, the translation logic has to monitor the $\overline{EF}$ and $\overline{FF}$ status and return an appropriate $\overline{XRDY}$ back to the

[1] For a more precise description and timing diagram of standard access cycles, please refer to section A.1.1

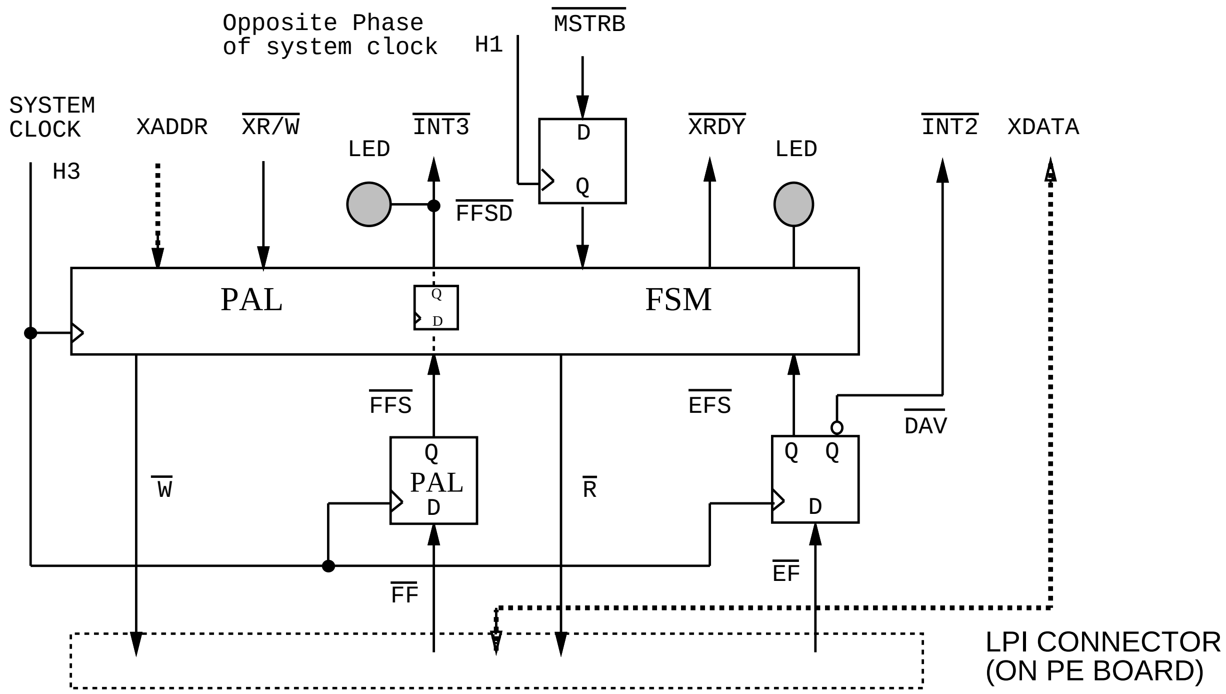

Figure 4-2: PE-to-FIFO Interface Logic

TMS320C30. To accomplish this dual control purpose, a finite state machine (FSM) controller is implemented in a PAL device.

Figure 4-2 shows the interface between the TMS320C30 and the FIFO. For a full specification of the FIFO control FSM and the associated timing diagrams, please refer to sections A.2.1 and A.2.2, respectively. Given the 40ns access times of the FIFOs, the delay through the FSM PAL, and the bus timing constraints, accessing the FIFOs requires at least two (60ns) DSP cycles.

4.1.5 The Alphanumeric Display

Testability is an important feature of any system. This is particularly true of prototypes, such as the NuMesh system. One standard way of testing systems is to run some diagnostics, and have the system report it's progress through some form of I/O stream that a human supervisor can observe and interpret.

Up until now, the only real I/O specified is via the LPI. Unfortunately, this interface would require a fair amount of additional hardware to be of any use to a person, especially if the primary function of LPI is to be preserved during testing. For this reason, it was decided to put an alphanumeric display device on the PE board.

The device used is Hewlett-Packard's HPDL-2416 4-character LED display. The display portion of the part is four red LED bubble-lens type characters, each capable of displaying any numeral or capital letter, and various other symbols. The control logic of the part includes ASCII-subset character decoding, and storage for each character.

In the configuration used, the user presents the device with a 2-bit address to select the character position, and a 6-bit ASCII-subset character word on the data pins. The character is then strobed in, whence it will remain on display despite any other bus activity, until the same character is rewritten.

One small problem with the HPDL-2416 is that it is a rather slow part, requiring more than the usual set-up and hold times, as well as requiring a long write pulse. For this purpose, a 74F194 shift register is used as a simple display controller.

The shift register uses $\overline{IOSTRB}$ without any additional decoding as an input to initiate a write to the alphanumeric display. One of the two outputs of the shift register is used as the $\overline{WR}$ (*write*) input of the display. The other output of the shift register informs the PAL of the end of a write cycle. The PAL uses the highest bit of the expansion bus address to select between its two ready signals (the one internally generated one by the FIFO control FSM, or the shift register output) to produce the actual $\overline{XRDY}$ that goes back to the TMS320C30.

The display device is connected to the expansion bus, meaning that from a programming standpoint, the user simply writes an ASCII character to the appropriate expansion bus address i.e. $\overline{IOSTRB}$ is active, highest address bit selects the display (according to the PAL firmware) and lowest two address bits select character position.

Please refer to section A.2.3 for the timing diagram and details of the alphanumeric display control logic.

Originally, it was unclear if the emulation facility described below would be supported by the TMS320C30 devices that were at our disposal, and therefore the role of the alphanumeric display would have been much more important in the task of debugging and testing. However, even with the emulation port fully functional, the display can still be used to good effect for debugging [2] once multiple NuMesh modules start computing and communicating concurrently.

4.1.6 Emulation Port

Another excellent feature of the TMS320C30 that greatly simplifies the task of testing and debugging, is the on-chip emulation support. The emulation support system consists of a card that plugs into the backplane bus of an IBM PC-compatible microcomputer, and some associated application software. A cable connects the card to the TMS320C30 by plugging into a twelve-pin header socket on the PE board. The socket is permanently connected to a set of dedicated emulation pins on the TMS320C30. These pins allow access to an internal serial scan path that includes the entire processor state.

Once connected to the emulation port, the developer can use the emulator software to examine or modify the processor state. This facility extends to anything else on the PE board that is normally accessible to the TMS320C30, including the external memory, alphanumeric display, and FIFO controller. The user can also single step through on-board code, or execute it at full speed. The emulator includes other standard debugging features, such as the ability to set breakpoints.

Using the cross-compiler, cross-assembler, and cross-linker, TMS320C30 object code can be be developed and generated on the IBM-compatible PC. With the aid of the supplied simulator, the object code can first be simulated on the PC, or using the emulator, directly downloaded onto the PE board (i.e. into internal or external memory), for on-board execution.

[2] Besides, it is almost as cute as the Connection Machine's^{tm} LED arrays!

4.2 External Bus Configuration

In distributing the above-mentioned devices onto the TMS320C30's two busses, one must consider the following issues: bootstrapping, required wait states and $\overline{RDY}$ / $\overline{XRDY}$ generation, bank switching, and optimizing transparent DMA transfer to / from the FIFO. This section addresses these issues and proposes a suitable configuration that is implemented in the PE board.

4.2.1 Booting

How to bootstrap a system from power-up into running the users application is usually a multi-level process. In the prototype NuMesh system, it may also be relevant to ask whether each PE boots its local CP or visa versa.

Booting the network, namely loading all the CFSM transition tables, is a fairly complicated process whose discussion rightly exists in a document dedicated to the communications processor. The network booting process is outlined in section 3.2.2. The important point to note is that the the CP will be booted before the PE, and thus the PE receives a high-priority reset signal from the CP.

A low-priority $\overline{CFSMReset}$ signal from PE to CP is also provided as a hook for network failure related exceptions. Associated with the last signal is $\overline{CFSMRun}$, a signal from the CP back to the PE that informs the PE that the network is still running. If the polarity of this signal were to be inverted by the CP, it would be suitable for interrupting the processor on a network failure. Currently, the TMS320C30 would have to poll the interrupt pin connected to this signal.

As far as the PE is concerned, then, it receives a reset signal at the some point in time. What the TMS320C30 does on reset is to start execution by vectoring off of location zero. Thus, the first important question should be *"what should location zero map to?"*. This question is deferred until later. The second question is where the vector should point. Obviously, before reset, none of the PE's RAM could have been initialized, which leaves us with two options as to what the vector should point to:

- **ROM** — The DSPs lowest level boot code could reside in ROM somewhere on the PE. The code would initialize the PE so that it may start uploading the application (or the next level of boot / operating system) code from the CP.

- **The CP Port** — Since the CP buffer is a memory-mapped port, the reset vector could point to that location in the memory map. The DSP could then execute boot code supplied directly by the CP.

It is truly unfortunate that the TMS320C30's 1K of internal ROM is not specifiable, necessitating the addition of external ROM chips to implement the first method. The main appeal of booting out of ROM is that the PE board can be debugged independently of the CP. A similar feature is that on power-up, the PE can perform self-diagnostics without relying on the rest of the system being functional (a likely situation for an experimental machine!).

4.2.2 Booting Method: Bus Requirements

A major benefit benefit of having the ROM, is that it presents an obvious answer to the question of what should be mapped to location zero. The ROM could be mapped starting at location zero, and the reset vector could be a small offset pointing to an initialization routine elsewhere in ROM.

The second method would require that the instruction pointer continuously point to the CP buffer throughout the booting process. Also, given that the next instruction is always the next incoming word, branches in the supplied code are meaningless.

The instruction pointer requirement can be satisfied by mapping the CP port to a large contiguous block in the memory map, which can easily be done if only the high order bits are used to select the buffer.

The main reason that the second method might be preferable, is if the chip count on the PE board can be reduced by eliminating the external ROM. However, if the external ROM were to be completely eliminated, the zero vector would also have to come from the CP buffer. That constraint would in turn require that the buffer be mapped starting at location zero.

The problem with mapping the CP buffer to location zero is that the remaining TMS320C30 interrupt and exception vectors are located in the in the successive locations. That would imply that, for normal (non-booting) execution, one should map those locations to RAM (or ROM).

Having two things map to the lowest locations does not present a contradiction, since they would only be required to do so at different points in time, namely, before and after booting. One could thus provide a “boot bit” in a flip-flop that is used by the decoding hardware to provide some dynamic control over how the memory space is mapped. The cost of implementing a boot bit is extra hardware and the potential increase in address decoding time.

A final constraint is a strong preference for putting the external RAM and CP buffer on separate external busses. Doing so increases the bandwidth for DMA block transfers between buffer and RAM, as well as reducing competition with the processor for the bus during DMA extended access cycles to the buffer.

4.2.3 The Need for ROM

Having already established the requirement of an external RAM bank and a memory-mapped CP buffer, the question is whether we can get away without adding external ROM to the board.

Taking the above constraints into account, and attempting to get by without ROMs leads us to the following contradiction.

- The CP buffer must be mapped to address zero, which is part of the Primary Buss' address space.
- The RAM must also map to the lowest addresses so that interrupts can be used.
- The CP buffer and the RAM must be on different external busses to optimize transfer rates between the two devices, and to eliminate contention between processor RAM access and DMA extended access to the CP buffer.

Clearly, if we wish to satisfy all of the above constraints which were established for the sake of performance and simplification of hardware, we would have to include an external ROM bank on the board. Even though the restriction that the buffer and RAM be on separate busses may yield negligible improvement in performance, there remains a compelling argument for including on-board ROM. The main appeal is the testability / diagnostic features mentioned in section 4.1.3. As the whole system becomes more bug-free, it is quite conceivable that future revisions of the PE will have the ROMs eliminated from them. If that happens, both the FIFO and RAM would be mapped to the primary bus as described in section 4.2.2, using a "boot bit" to select which device is actually mapped to location zero and up.

4.2.4 Chosen Bus Configuration

Having established the presence of an external ROM bank on the board, the following configuration of the TMS320C30's external busses was chosen.

- Mapping the ROM on the primary bus, starting at location zero, simplifying booting and interrupt handling. [3]

- Mapping the RAM also on the primary bus, benefiting from the bank switching feature of that bus. Also, since both ROM and RAM access cycles are zero-wait-state for the devices chosen, placing them alone on that bus greatly simplifies address decoding and ready generation, the latter of which can be done entirely within the TMS320C30.

- Since the only devices on the primary bus are one bank each of RAM and ROM, only one address bit is needed to decode primary bus addresses. The consequence of only using single-bit decoding is that the RAM and ROM banks can be accessed at multiple *sets* of addresses, since there exists many different addresses that cannot be distinguished by the hardware. Given the chip-select signals for the chosen devices, decoding is accomplished using a simple inverter on a higher order address bit.

  Although in the first version of the PE board, $A22$ is used to select between RAM and ROM, future revisions will use $A13$. By using decoding $A13$ instead of $A22$, as originally planned, two things are accomplished. The first accomplishment pertains to TMS320C30 addressing modes: both ROM and RAM can be addressed in immediate mode without changing the data page pointer (DP), and also making it easier to use short immediate constants as addresses. The second reason is that the external RAM reappears directly after the internal RAM, giving the programmer a single contiguous RAM address space.

- Mapping the CP buffer and the alphanumeric display onto the expansion bus. Both of these devices have access cycles that differ from each other and from the memory banks on the primary bus. For that reason, they might as well be lumped together. Expansion bus address decoding and ready generation can be implemented in the PAL that must be dedicated to handling the CP buffer's extended access, and thus already has some of the appropriate inputs and outputs.

Figure 4-3 shows the TMS320C30 memory map, with the final mapping of external devices superimposed on it. The map on the left represents the old (current) version of the board using

[3] We can still "pretend" to have programmable interrupt vectors by making the hard interrupt vectors point to a set of trivial interrupt preambles in ROM that use a second set of vectors that reside in predetermined area of RAM.

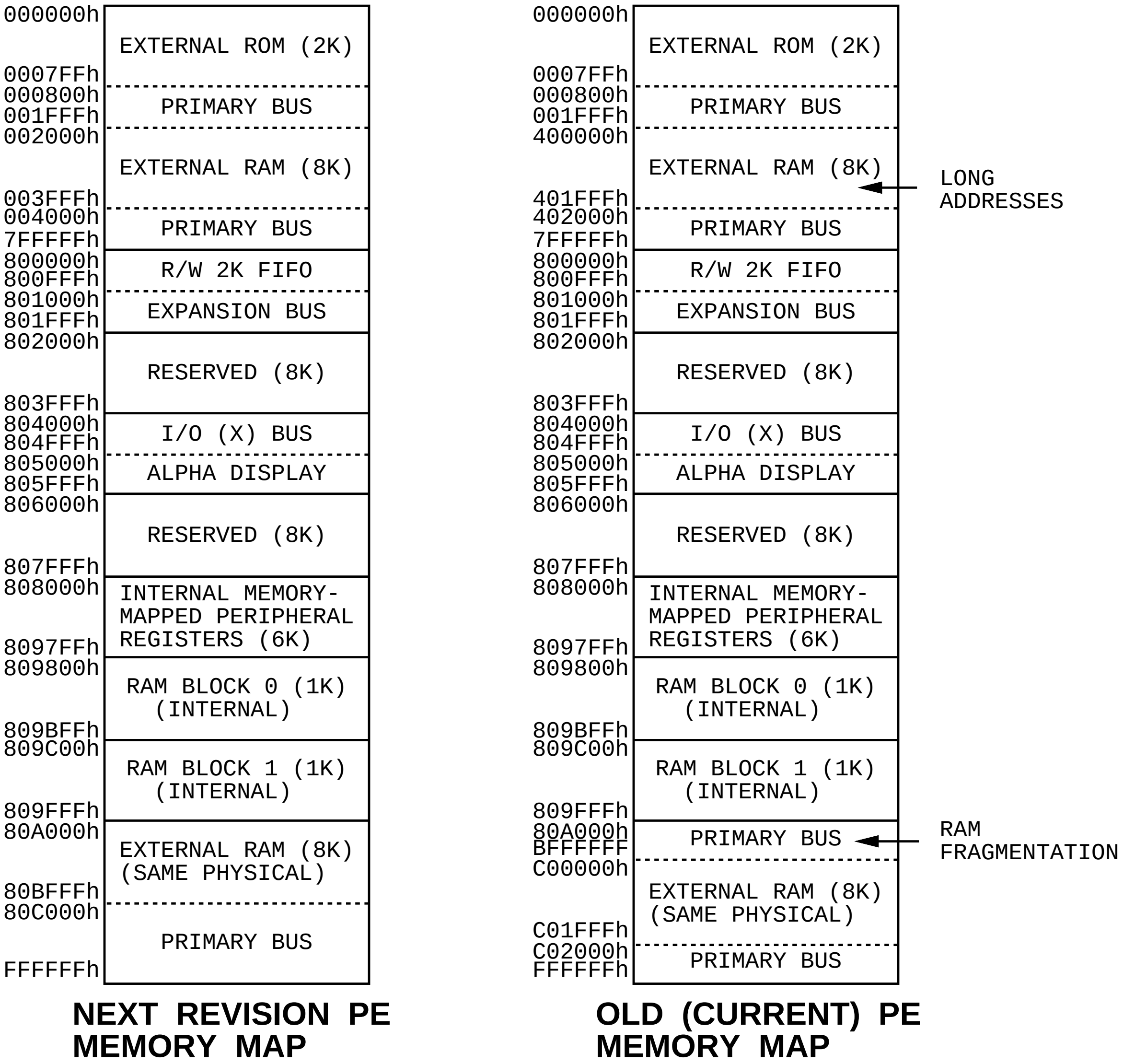

Figure 4-3: Processing Element Memory Map

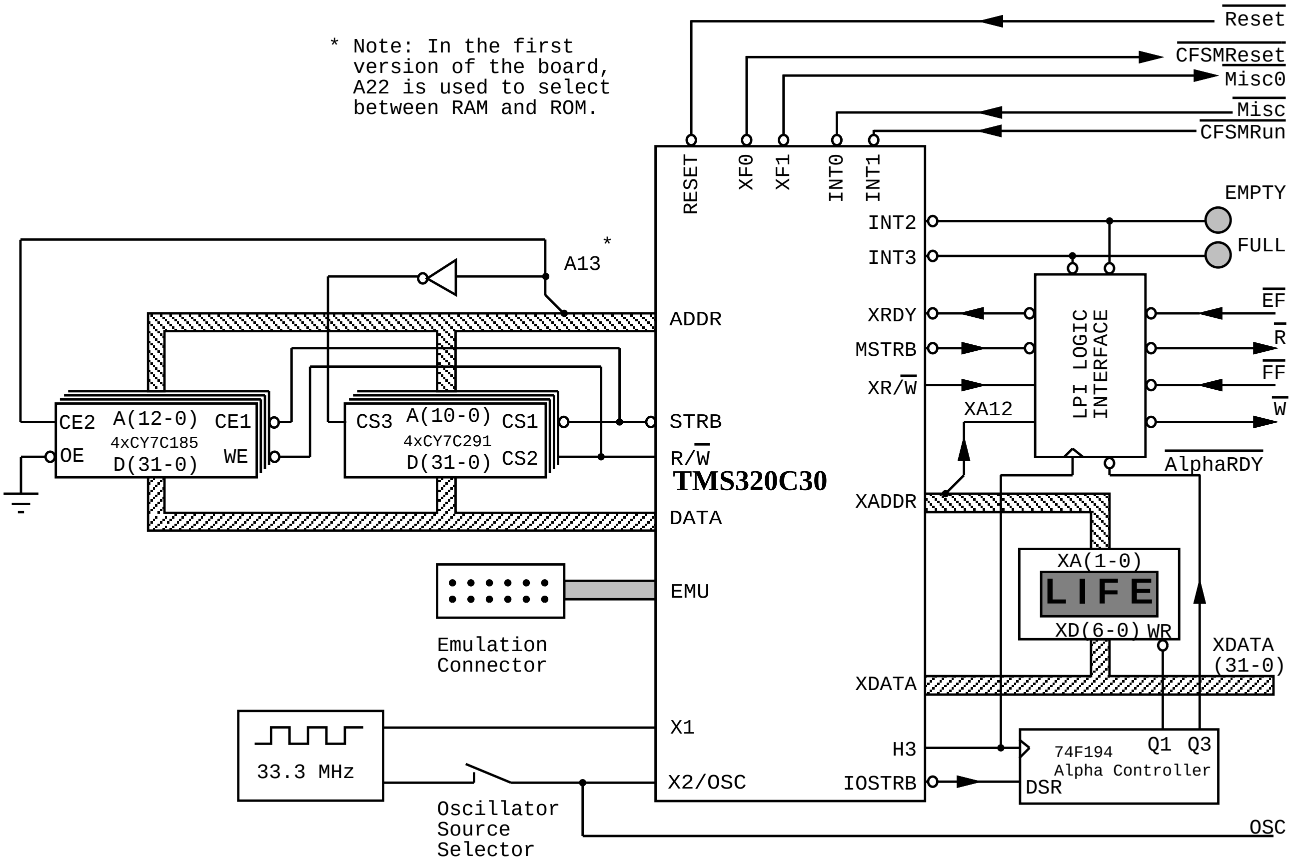

Figure 4-4: Processing Element Block Diagram

$A22$, while the one on the right represents the memory map of the next revision of the PE board.

4.3 Summary of Chapter

Figure 4-4 summarizes what subsystems the PE comprises, and how these subsystems are arranged about the TMS320C30 central processing unit. Note that most of the "LPI LOGIC INTERFACE" black box is detailed in figure 4-2, in section 4.1.4. The two additions to the logic shown in figure 4-2 are the $\overline{AlphaRDY}$ and XA12 inputs to the PAL, which are used for the combined ready-generation for the FIFO and alphanumeric display.

Chapter 5

Design Process, Results, and Conclusion

5.1 CAD / CAM Tools

The circuit board for the PE was designed and manufactured with the aid of a CAD / CAM package written by Douglas Systems. The software consisted of a suite of three integrated applications that run on a Macintosh II. The three applications consist of a schematic capture program, a board layout program, and an auto-router.

5.1.1 Schematic Capture

The schematics for the design were entered using Douglas' schematic editor. A program such as this has two main functions. The first function is to serve as a drafting aid, namely to maintain a neat graphical representation of the schematics. A good editor should make it easy for the user to edit the schematic and to get updated hard-copies for closer study.

Associated with the schematic editor is a schematic librarian. The librarian program is used to create a library of parts for incorporation into schematics using the schematic editor. Although the original software is supplied with a library of parts, the library did not contain the TMS320C30, nor many of the other parts used in the PE. This meant that the fairly tedious task of creating a graphical library of parts used had to be undertaken.

The other main function of the schematic capture program is the ability to create a net-list from the schematic. The main use of the net-list is as input to the auto-router, which is described in a later section.

5.1.2 Board Layout

The schematic program described above is used to create a logical description of the circuit topology. In contrast, the board layout program generates a close representation of the actual masks that would be used for the etching the board.

Six layer Board

The masks to be generated generally consist of two main types of features: pads and traces. The final board had four signal layers, and an additional two planes for power and ground, which meant that six masks had to be created. The layout editor allowed one to work with all planes at the same time by overlaying them over each other.

Board Features

Pads are copper-clad areas, drilled to allow component pins to be place through them for soldering. There is one pad for every component pin on the board. Pads are also grouped together to form the pad "footprint" of multi-pin devices. Once again, although the supplied software included a library of standard device footprints, many (including the 180-pin grid array for the TMS320C30) had to be created specially for this board.

Traces are the copper tracks that connect pads together. No traces were manually placed on this board. Instead, the traces were automatically placed by the third program in the suite, the auto-router.

Generally, pads go on all planes at once, but with an insulating ring around them when on a power or ground plane. Traces go only on a particular plane. Finally shorting bars are inserted to connect power or ground to a pad in the respective power or ground plane. Pad holes are plated through with metal, so that a pad is electrically connected at all planes.

Footprint Layout

The process for creating the masks involved first using the layout editor to place all the pad footprints to create an unrouted board. Second, the unrouted board and the schematic net-list are both passed to the auto-router, which places traces between the pads in accordance with the connectivity specified in the schematic.

5.1.3 Auto-Router

Although the process outlined above sounds simple enough, it turned out to be the hardest part of this thesis project. The problem stems from the limited intelligence of the auto-router; without any human help, the auto-router would only be able to route about 70% of the net-list.

Since the net-list produced by the schematic was frequently being updated, the complete routing process needed to be automated, otherwise, every minor schematic modification would involve hours of tedious human work in modifying the (partially) routed layout file. Some degree of automation can be achieved by making up a set of files that can help the auto router to route the entire net-list.

Command File

There are essentially to two types of files that can help the router. The first type is the obligatory router command file. Although a default command file is supplied with the software, it was naturally inadequate for our purposes. The text-only command file specifies various constraint, tolerance, and cost parameters of the routing process. Three examples of these parameters are:

trace width, minimum trace separation, and relative cost of putting a vertical trace versus a horizontal trace on a particular plane.

Barrier Files

The other type of file that can be used to help the router is an optional barrier file. This is a graphical file created using the board layout editor. The barrier file specifies geographical constraints on where the router may place traces. By designing an appropriate set of barrier files, the router can be prevented from routing early traces in areas that have to be used by later traces, effectively painting itself into a corner, to use a popular metaphor.

5.1.4 Iterative Routing Process

The board was finally routed through an iterative process of cycles consisting of footprint placement (to make certain trace-connected pads closer together, to allow more space between devices, or to absorb unused space), and command and/or barrier file modifications. A cycle is ended by invoking the auto-router on the new set of files. Occasionally a routing cycle would start with a new net-list resulting from a modification of the schematic.

Once again, I would like to thank Ricardo Jenez for undertaking the extremely time consuming task of routing the board, made non the easier by often inadequate documentation of the tools, by software bugs that resulted in frequent machine crashes, and by auto-router runs that were so long that they had to be left running overnight.

The final routed file was then mailed to Douglas Systems who manufactured a few instances of the board, and mailed them back to us along with the etch masks.

5.2 Testing

After the board came back from Douglas Systems, it's trace connectivity was first verified against the schematic, using an ohm-meter. This time, credit is once again due to Jim Olsen for debugging the board's traces (see section 5.2.1 below), and for soldering the parts onto the board.

5.2.1 Ground Bug

The first error was discovered in the board's trace connectivity. The problem was that the ground supply pins on all four CY7C185 SRAM devices were not connected to the ground plane via shorting bars. The error was traced to the schematic, where there was discovered a signal name mismatch between the ground net and the ground pins of these particular devices.

The schematic was modified so that future boards would not suffer this problem. The problem was remedied on the board by using some very heavy gauge wire to connect the floating device pins to some correctly grounded pins.

Once this phase was tested and debugged , the board was populated by socketed integrated circuit packages and discrete components.

5.2.2 Power-Up Test

The populated board was powered up and connected to the emulation environment for the next phase of testing. This test was in some sense the true test of whether the board was working or not. Fortunately, on power-up, the emulator screen on the PC indicated that the board was indeed connected, and that the displayed first few words of the ROMs did indeed contain what they were programmed with.

Next, the first diagnostic TMS320 code [1] was download from the PC into the TMS320C30 internal RAM space. The program was designed to test four aspects of the hardware:

- Primary bus speed — Set memory wait-states in software to ensure that the bus timing requirements are satisfied by the hardware.
- Display check — Display the four characters "LIFE" on the alphanumeric display.
- RAM check — For all external RAM locations, starting with the lowest order data bit, write a 1 into that bit position and zeros in the remaining 31 bit positions. Next, execute a checking pass on all locations to ensure the correct value was written and read correctly. Repeat for the remaining 31 bit position, reporting an error at the first incorrect bit, otherwise reporting success.
- Interrupt and I/O pins check — For each of the four interrupts, a handler was written to display the interrupt number and output the two-bit number on $< XF1;XF0 >$ I/O pins. The final section of the diagnostics code enables interrupts and then spins in an idle loop forever, except for servicing the occasional interrupt.

  To test the test code for the interrupt handlers themselves, just before enabling the interrupts, the internal interrupt flags were set in software. The idea was that the four handlers would display their corresponding messages at least once, right after interrupts were enabled.

The above four tests were all completed with varying degrees of success, as described below.

5.2.3 Clock Bug

The bus did not seem to be running reliably with zero wait states. The most obvious indicator that something was wrong, was that while single-stepping the emulator through the test code, the stack pointer would be updated to a fairly random value for each instruction, regardless if it was a stack operation at all.

The problem was finally relieved by temporarily disconnecting the on-board oscillator circuit, and instead feeding in a 25 MHz clock from a signal generator. There were two theories as to the nature of the problem: either the emulator system was interfering with TMS320C30 operation at the manufacturers specification, or the TMS320C30 part that we had was defective.

Much later (after almost everything had been debugged at 25 MHz) the problem was finally traced to the oscillator circuitry. The crystal circuit's output was so bad that it was affecting the TMS320C30 internal clocks, $H1$ and $H3$ which are out-of-phase divided-down-by-two versions of the crystal input. By removing the crystal and soldering in a new one, everything was again tested with the result that the system worked correctly at full speed.

[1]The full diagnostics code as downloaded into RAM is given in Appendix C

5.2.4 Display Bug

At the time that the diagnostic code was executed for the first time, the PAL had not been programmed with the correct code, and the display did not work at all. The success of the RAM check test was determined by inspecting the appropriate register after a breakpoint at the end of the test.

After modifying the PAL, the display did work, but unreliably. Sometimes the correct symbol would be displayed, and sometimes the display would not be updated at all when it should have been.

Using the Hewlett-Packard logic analyzer to monitor and display the inputs and outputs of the display controller shift register, a bug was discovered in the logic design of the display controller which resulted in a timing requirement of the display not being met. The controller was fixed by changing the some of the wiring to the shift register, and updating the schematic. After that fix, the display worked flawlessly.

5.2.5 Interrupts Mystery

One aspect of the hardware whose correctness was never fully determined by the diagnostics program, is the TMS320C30 interrupt subsystem. The code never did produce the desired effect. This was due either to the probability that the emulator system uses / bypasses some of the interrupt subsystem, or to a software error. I believe that the problem stems from a software error in the part that relies on servicing the four internally generated interrupts.

The external hardware was successfully tested by writing a patch of code that read or wrote the IF and IOF registers and confirming that they corresponded to the logic values of the appropriate interrupt or I/O pins.

5.2.6 Testing the FIFO Controller

Since a full test of the FIFO controller requires the CP board (housing the FIFO devices), such a test was impossible to conduct, since the CP board was not ready at the time of testing. For the purpose of testing the controller, a minimal amount of the CP's functionality was emulated by wiring a single CY7C429 FIFO device on a proto-board.

The inputs and outputs of the makeshift FIFO were tied together and to the lowest eight bits of the TMS320C30 expansion bus at the LPI connector. The CY7C429's reset pin was connected to $XF0$ configured as an output, and both $\overline{EF}$ and $\overline{FF}$ of the one device were connected to the appropriate pins on the LPI connector. Finally the $\overline{W}$ and $\overline{R}$ outputs on the LPI connector were tied to the corresponding pins on the CY7C429 FIFO device.

Next, some test code [2] was written and downloaded for execution. The program keeps a loop counter which can be reloaded from a maximum count register. The code was written in this way to allow easy changes in the number of iterations to be performed, by loading different constants into the one register.

After configuring the wait state generation and resetting the FIFO, the code loads a constant into the maximum count register. Next the program loads the loop counter and then repeatedly writes the value of the loop counter into the FIFO, decrementing the counter until zero. Finally the program reloads the loop counter, and repeatedly dumps the output of the FIFO into

[2] The code used to test the FIFO is given in Appendix C

consecutive locations of a reserved area of memory. There normally was a breakpoint set at the end of this section

To evaluate the results, the emulator's inspect memory feature was used to examine what was read in from the FIFO.

5.2.7 The $\overline{MSTRBS}$ Bug

In testing the FIFO interface, it was discovered that the $H1$ clock used to delay $\overline{MSTRB}$ by half a clock cycle was not connected to the appropriate pin on the flip-flop. Once again, this was due to a difference in the names assigned to the same signal on different pages of the schematic. The hardware was patched with a wire and the schematic was updated.

5.2.8 The Glitchy Read Signal

Originally, the FIFO finite state machine controller was designed with 30 nanosecond FIFO devices in mind, instead of the 40 nanosecond parts that were available. Faster FIFO devices would have allowed the FIFO $\overline{R}$ and $\overline{W}$ signals to be generated directly from controller state bits within the PAL implementation.

Unfortunately, given the slower devices, a FIFO-read did not have enough time to complete if $\overline{R}$ was asserted for only one cycle. Instead, a last-minute fix was implemented by starting a FIFO-read cycle in the same cycle that the read request came from the TMS320C30, and not in the next cycle when the controller would go to the read state. This meant that the $\overline{R}$ signal had to be a potentially hazardous combinational output of the PAL.

In the first few versions of the controller, the hazard resulted in a glitchy $\overline{R}$ signal, which would have been disastrous if left untreated. The hazard was finally eliminated using a somewhat ungraceful solution that relies on the PAL delaying the clock by a minimum amount. The delayed clock is used to gate the $\overline{R}$ signal as soon as a TMS320C30 FIFO-read request comes in i.e. while the FSM is still in the IDLE state.

The entire problem that warranted such an ungraceful solution in the first place can be completely eliminated as soon as faster FIFO devices become available. All that is involved is a slight modification to the PAL source code, and programming a new device for a more robust design.

It should be noted that a reprogrammable 15 nanosecond 16v8 PAL device is currently being used to emulate the 16r4. When the FSM is changed to control the faster FIFOs, the 10 nanosecond speed of the 16r4 will be a necessity.

5.3 Summary and Conclusion

The limitations of the backplane as a communications substrate were exposed and the NuMesh was introduced as a possible alternative. A project to build a prototype NuMesh based system was embarked upon in order to evaluate the potential of the ideas behind NuMesh. The test system consists of an array of modules connected to each other in a two-dimensional grid.

Each NuMesh module consists of a processing element (PE) that can communicate to a communications processor (CP). The communications processor has the ability to communicate to its neighboring modules' communications processors, thereby providing a mechanism that allows all processors to communicate with each other.

The requirements of a TMS320C30-based digital signal processing element for the prototype NuMesh module were determined, and were followed by a logic design for the element. Using some CAD tools, the circuit board layout for a compact processing element was made, and a board was manufactured by a third party.

The manufactured board was populated and tested with some diagnostics code written to test different aspects of the hardware. Some hardware bugs were found, and were all eliminated by various patches to the board. The CAD files were updated so that future boards would not suffer the same problems.

Complete testing of the asynchronous interface to the communications processor is impossible until the latter is ready. As of this date, the communications processor board is almost ready, and a more complete diagnostic of the bipartite module will soon be made.

Appendix A

TMS320C30 Bus Timing

A.1 Primary Bus

The primary bus accesses the external ROM and RAM banks. The bank-switching feature of the TMS320C30 can be programmed to insert an extra cycle when switching from ROM to RAM on consecutive cycles. Slow decoding hardware can benefit from the extra delay. In this case, decoding between RAM and ROM is done by an inverter on the PAL.

In this implementation, the primary bus is used exclusively for zero-wait-state access to the external RAM and ROM banks. With the memory devices currently on the board, this presents no problem. However, the current ROMs are only once-programmable, and it may be desirable to replace them by some EPROMs, making it possible to change the contents without needing a new set of chips for each revision.

Unfortunately, fast reprogrammable EPROMs are harder to find, since EPROM technology tends to be slower. For this reason, a minor feature built into the primary bus interface allows programmable wait state generation for ROM access while maintaining zero wait state access to the RAM. The appropriate mode is selected in hardware completely transparently to the user.

This is accomplished by connecting the external RAM select line to the external $\overline{RDY}$ pin. The internal $\overline{RDY}$ can be programmed to be the logical-OR (i.e. the first) of the internal, counter-generated (for ROM) or the external $\overline{RDY}$ (for RAM).

A.1.1 Memory Timing

Memory reads are very straight forward. The address and $\overline{STRB}$ are asserted at the beginning of the cycle, and data appears at the end of the cycle. Consecutive reads can be chained without any overhead by asserting $\overline{STRB}$ throughout the sequence, and changing only the address between reads.

Memory writes have a minimum cycle time of two clock periods. This stems from the fact that a clean $\overline{STRB}$ edge is necessary to initiate a read cycle within the static RAM device. Thus an extra half cycle is inserted before and after the assertion of $\overline{STRB}$.

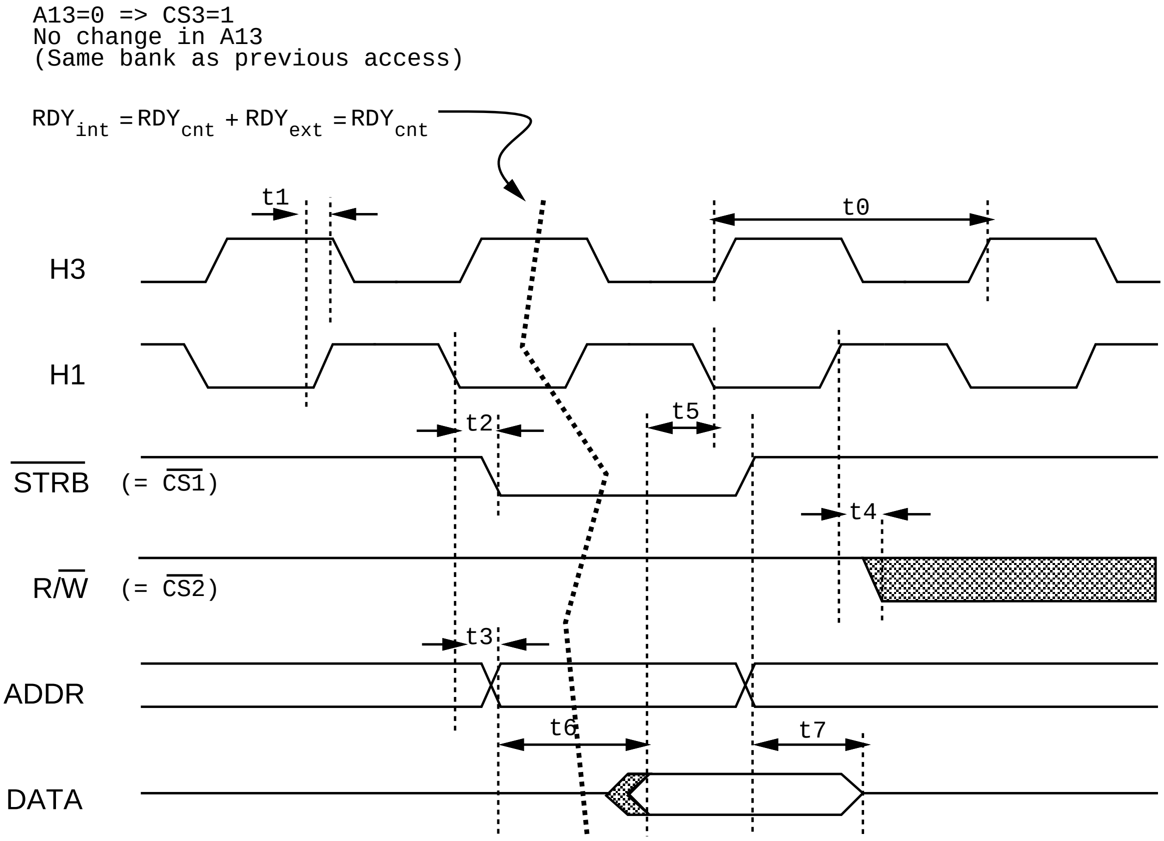

Figure A-1: ROM Read Cycle

Table A.1: Clock Parameters

Symbol	Parameter	min(ns)	max(ns)
t0	$H1/H3$ cycle time	60	
t1	$H1$ to $H3$ skew		5

Table A.2: Primary Bus Parameters

Symbol	Parameter	min(ns)	max(ns)
t2	$H1$ low to $\overline{(M)STRB}$		10
t3	$H1$ low to $ADDR$		10
t4	$H1$ to $R/\overline{W}$		10
t5	Data setup from $H1$ low	15	
t10	Bank Decoding Inverter		10

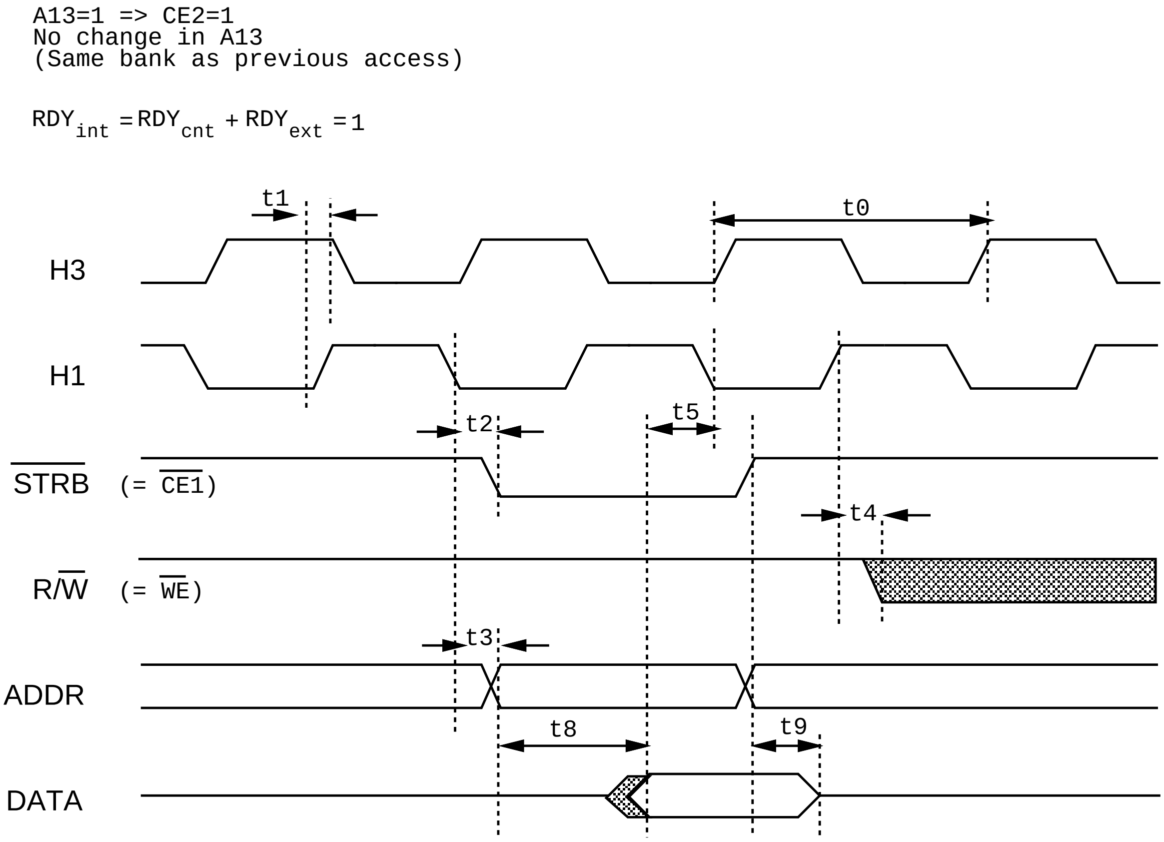

Figure A-2: RAM Read Cycle

Table A.3: ROM Parameters

Symbol	Parameter	min(ns)	max(ns)
t6	ROM access time		35
t7	ROM deselect time		25

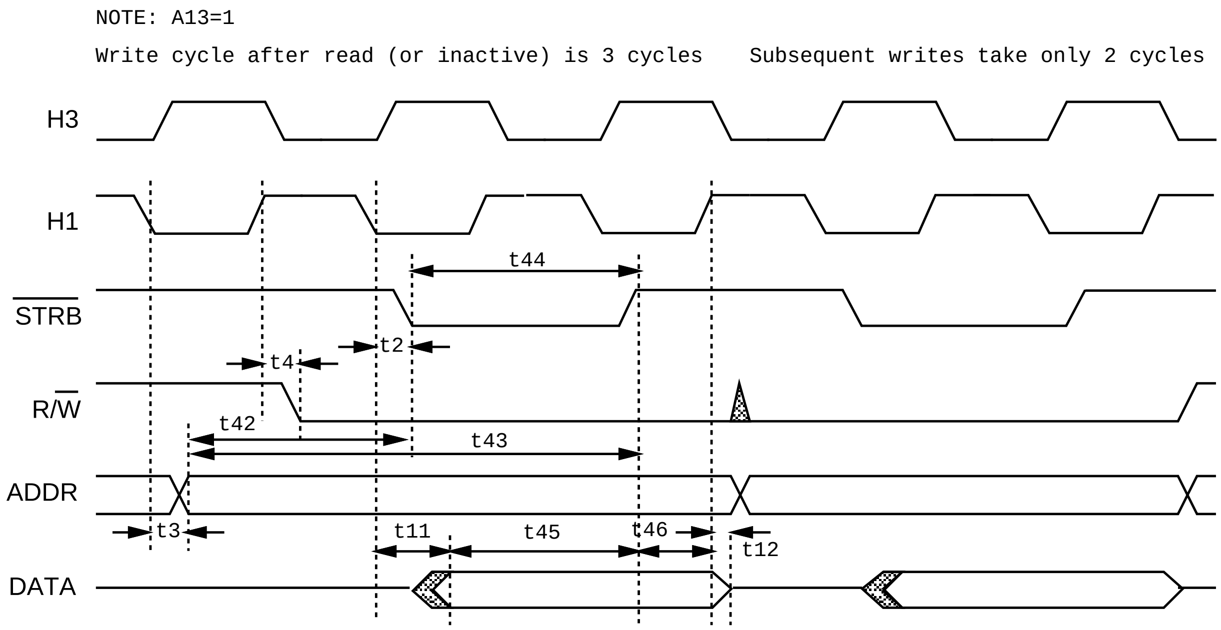

Figure A-3: RAM Write Cycle

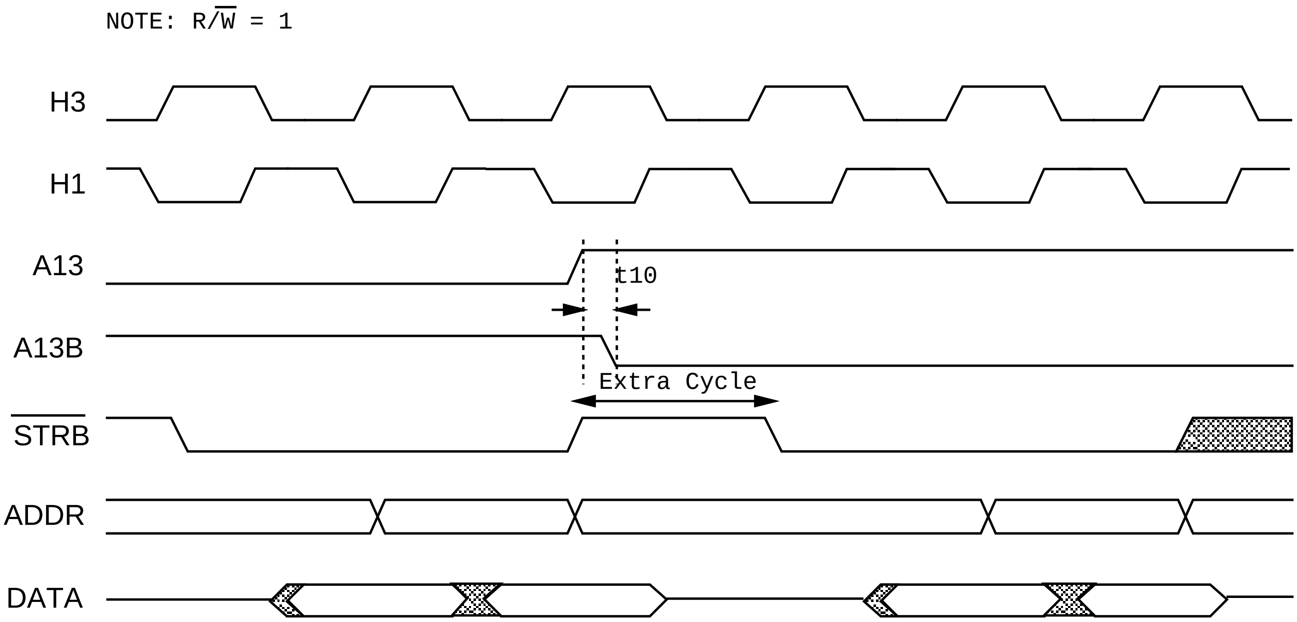

Figure A-4: ROM / RAM Bank Switch Cycle

Table A.4: RAM Parameters

Symbol	Parameter	min(ns)	max(ns)
t8	RAM access time		35
t9	RAM turnoff time		15
t42	RAM address setup to write start	0	
t43	RAM address setup to write end	25	
t44	RAM write pulse width	20	
t45	RAM data setup to write end	15	
t46	RAM data hold from write end	0	

A.2 Expansion Bus

The expansion bus services the FIFO, which is active on $\overline{MSTRB}$ and normally takes two $H3$ cycles for either a read or write access. [1] The expansion bus also services the alphanumeric display, which is active on $\overline{IOSTRB}$, and takes four cycles per write.

A.2.1 FIFO FSM Controller

The state transition diagram for the finite state machine is shown in figure A-5. The inputs on the arcs are in fact made up of macros that have been defined with the intent of making the diagram less cluttered and easier to understand. The macros are fully defined by the PAL source file written in the ABEL language. The source file and the ABEL-generated documentation file showing the reduced equation can both be found in Appendix C.

Furthermore, macros have been defined so that all inputs and outputs of the finite state machine are written in positive logic. The senses of the signals in the ABEL source state diagram are the same as those in figure A-5.

For quick reference, here is a description of the macro inputs:

- **FREAD** — signifies a FIFO read *attempt*, without further assumption about if it is empty or not.
- **FWRITE** — signifies a FIFO write *attempt*, without further assumption about if it is full or not.
- **VFWRITE** — signifies a *valid* FIFO write attempt, i.e. one that will succeed because the FIFO is not full.

The state machine is fairly straight-forward. The controller is usually in the IDLE state, if there are no read or write requests to the FIFO, and the FIFO is not empty.

On a read attempt while in the IDLE state, R (positive logic signal) is asserted as soon as possible while still in the IDLE state [2] and the controller goes to the READ state on the next clock edge, where R is asserted for another full cycle. While in the READ state, RDY (actually $\overline{XRDY}$) is asserted so that the transaction can complete as far as the TMS320C30 is concerned.

At the end of the READ state, the FIFO could have just been emptied if the datum read was the last word in the PE-bound FIFO. If that is the case, the controller goes to the EMPTY state, otherwise it goes back to the IDLE state as before.

On a write attempt while in the IDLE state, the controller either spins in the IDLE state or goes to the WRITE state depending on if the the FIFO is too full to be written or not. If the FIFO is too full, the controller spins until the FIFO is no longer full, whence it goes to the WRITE state. In the WRITE state, W and RDY are asserted to complete the TMS320C30 transaction.

The EMPTY state is somewhat interesting in that it mimics the behavior of the IDLE state for write attempts, but it stays in that state throughout the time that the FIFO is empty and

[1] A "normal" access refers to reading a non-empty FIFO or writing a non-full FIFO.

[2] The duration of R's assertion within the READ state would have been sufficient for a 30ns CY7C429 FIFO device, but only 40ns parts were available at the time. For further discussion see section 5.2.8.

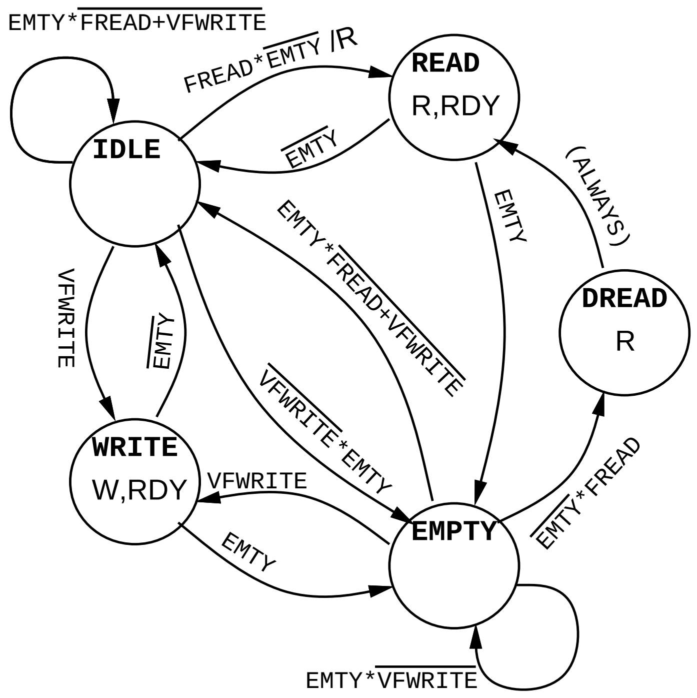

Figure A-5: FIFO Controller Finite State Machine

disallows FIFO reads. Writes originating from the EMPTY state, return to the EMPTY state, while writes originating from the IDLE state end return to the IDLE state. This is ensured by the two arrows that come out of the WRITE state.

Finally, while in the EMPTY state and a word comes into the PE-bound buffer there are two possible outcomes. If there already was a pending read request, the controller reads the FIFO by going to the READ state via the DREAD state, which is used to extend the length of the R pulse. Although the R pulse could have been started while in the EMPTY state (as from the IDLE state), this could cause a hazard since the read request could come in at the same time as $EMTY$ gets deasserted. Such a two input change of a combinational output could (and *did*, during testing) cause a race condition that results in a glitch. The DREAD state is inserted to eliminate that hazard by allowing only a single bit change (namely in the highest order state bit, $Q3$) to affect R. The DREAD state can be completely bypassed when using 30ns CY7429 FIFO devices, since the duration of R would be sufficient within the READ state.

The second alternative of a word coming in while in the EMPTY state is when there is no pending read request, in which case the controller goes back to the IDLE state.

A.2.2 FIFO Timing

Table A.5: Expansion Bus Parameters

Symbol	Parameter	min(ns)	max(ns)
t11	$H1$ low to $XDATA$ valid (writes)		20
t12	$XDATA$ hold time from $H1$ high	0	
t13	$H1$ high to $\overline{IOSTRB}$ valid		10
t14	$H1$ low to $XR/\overline{W}$		10
t15	$H1$ low to $XADDR$ valid		10
t16	$H1$ low to $XDATA$ valid (I/O writes)		15
t19	$\overline{XRDY}$ to $H1$ high setup	8	
t20	$H1$ high to $\overline{XRDY}$ hold	0	

Table A.6: FIFO Read Parameters

Symbol	Parameter	min(ns)	max(ns)
t26	$H1$ high to $\overline{MSTRBS}$ valid		9.2
t27	$H3$ to $H3D$ delay		10
t28	$H3$ low to $\overline{R}$ low		20
t29	$H1$ low to $\overline{R}$ high		20
t30	$\overline{R}$ pulse width	40	
t31	$\overline{R}$ low to data output valid		40
t32	$\overline{R}$ high to data output high Z		25
t33	$\overline{R}$ high to data output hold	3	
t34	$H3$ high $\overline{XRDY}$		18
t35	$H3$ high to FSM state valid		8

Table A.7: FIFO Write parameters

Symbol	Parameter	min(ns)	max(ns)
t36	$\overline{W}$ pulse width	40	
t37	FIFO data setup time	20	
t38	FIFO data hold time	0	

Table A.8: FIFO Flag Parameters

Symbol	Parameter	min(ns)	max(ns)
t39	$\overline{R}$ low to $\overline{EF}$ low		35
t40	$H3$ high to $\overline{EFS}$ valid		9.2
t41	$\overline{EFS}$ setup to $H3$ high	3	
twff	$\overline{W}$ low to $\overline{FF}$ low		35

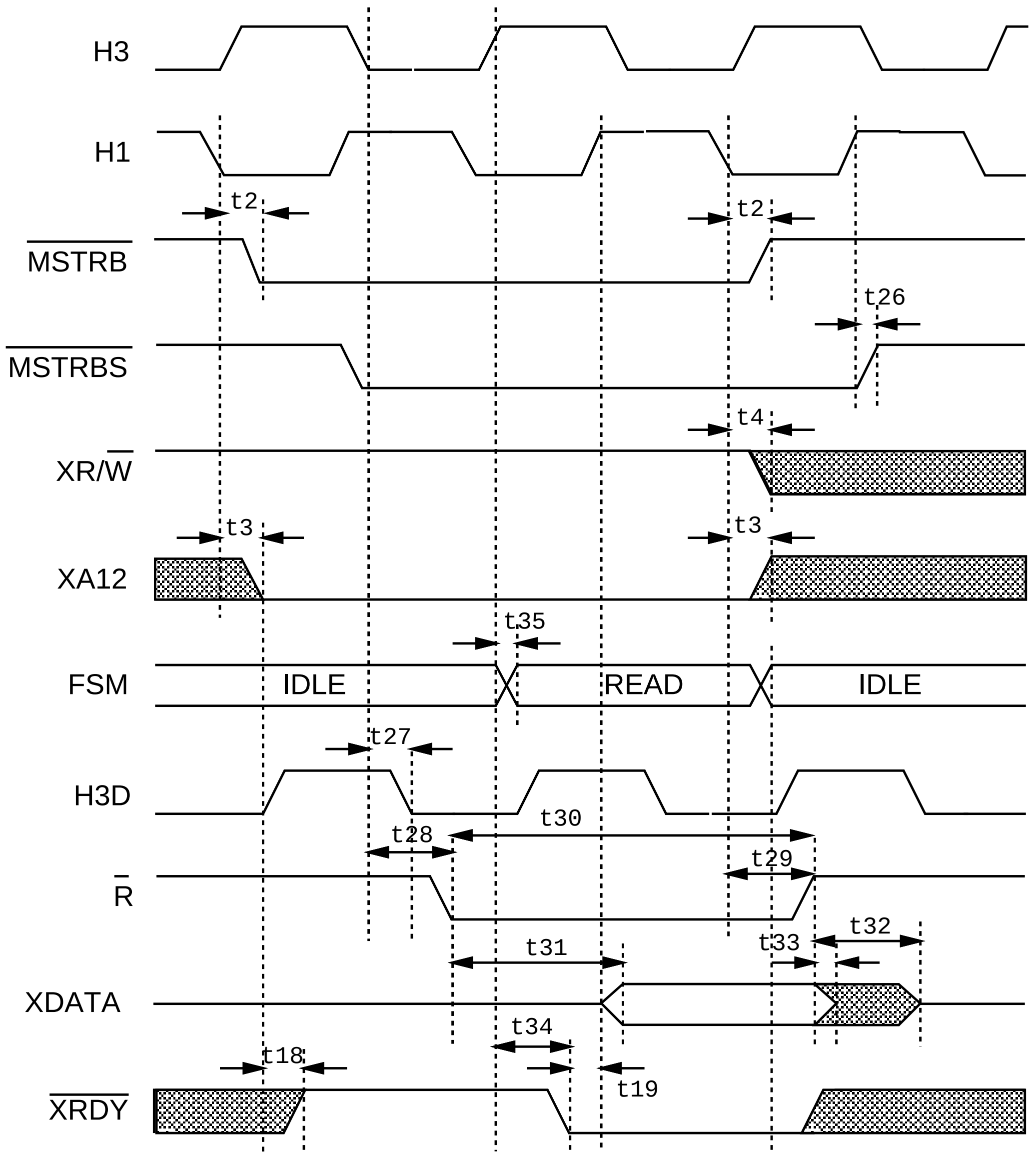

Figure A-6: Standard FIFO Read Cycle

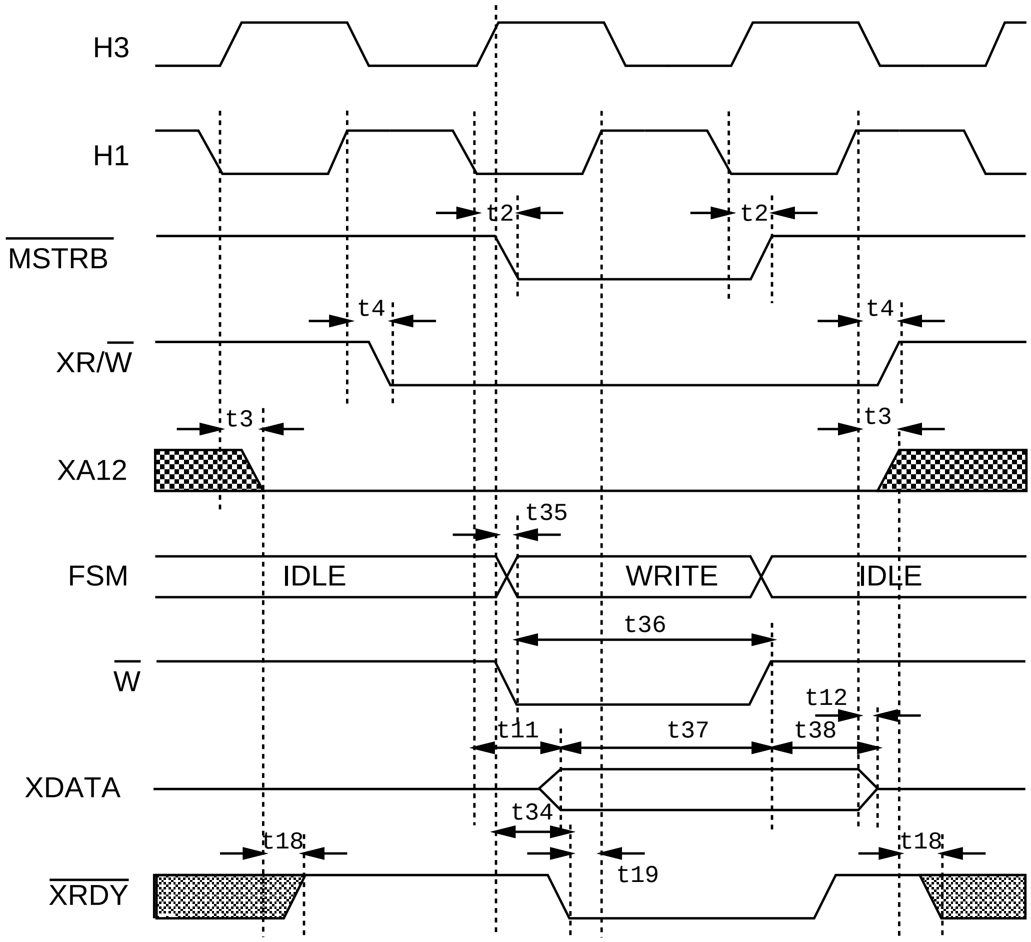

Figure A-7: Standard FIFO Write Cycle

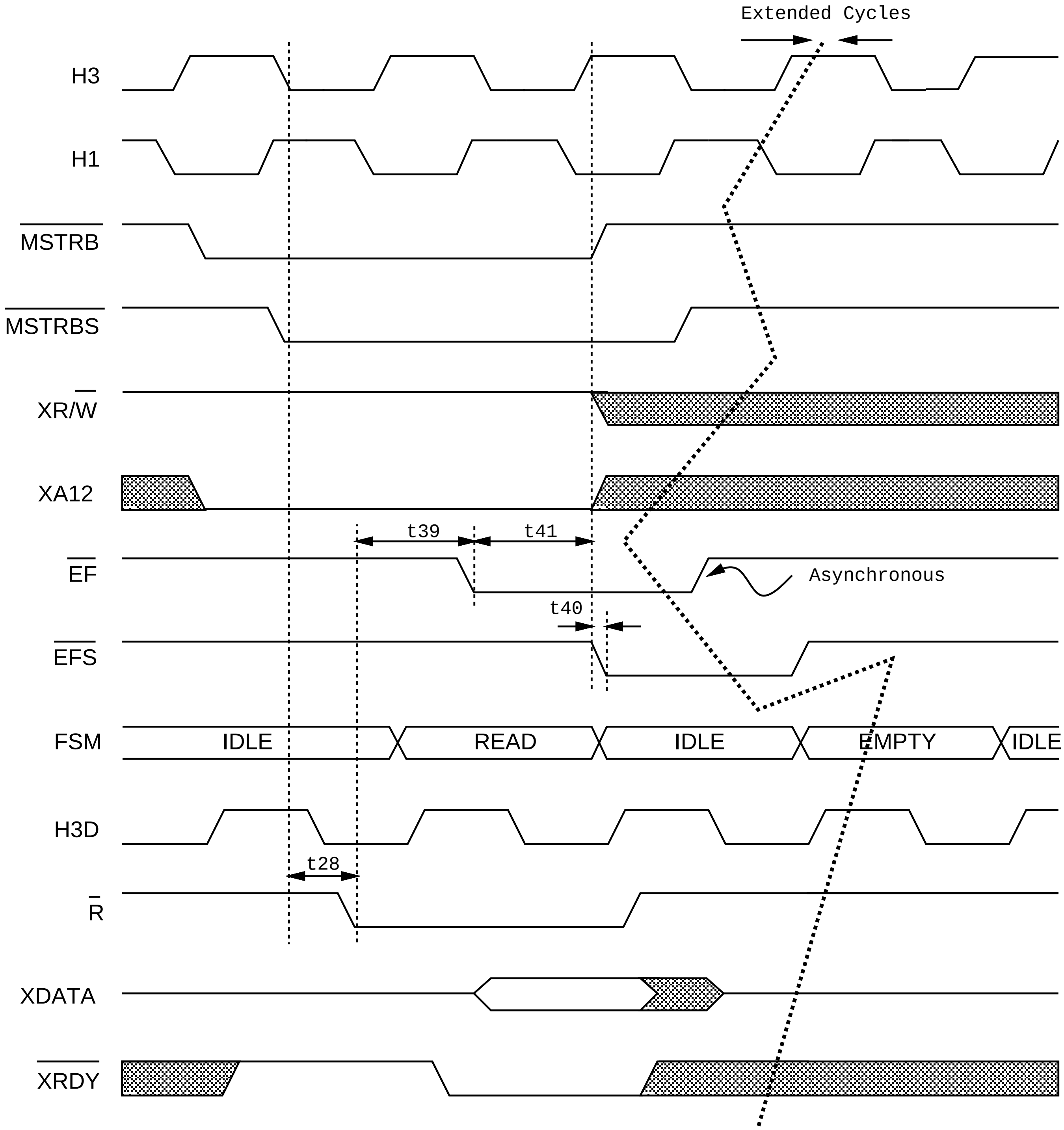

Figure A-8: Emptying FIFO Read Cycle

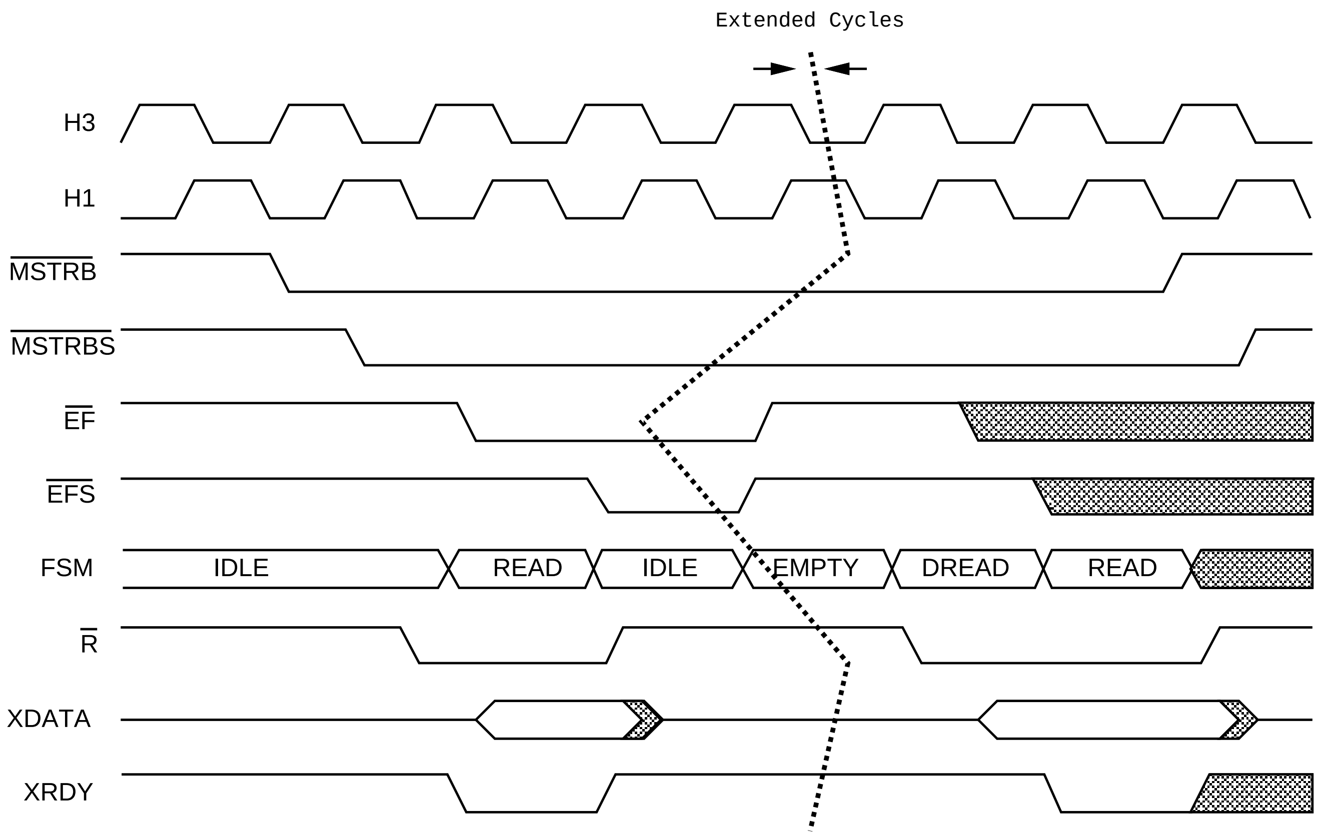

Figure A-9: Pending Read FIFO Cycle

Table A.9: Display Subsystem Parameters

Symbol	Parameter	min(ns)	max(ns)
t17	$H3$ high to shift register Q		8
t18	$XA12$,$XR/\overline{W}$,$\overline{ALPHRDY}$ to $\overline{XRDY}$		10
t21	Alpha write pulse width	100	
t22	Alpha address setup time	115	
t23	Alpha address hold time	45	
t24	Alpha data setup time	45	
t25	Alpha data hold time	45	

A.2.3 Alphanumeric Display Timing

The alphanumeric display is mapped to the I/O Bus which is physically on the expansion bus, and is activated by $\overline{IOSTRB}$. This fact is used to simplify the external decoding hardware by using the $\overline{IOSTRB}$ signal directly to initiate a write to the alphanumeric display, the only device on said "virtual" bus.

The $\overline{IOSTRB}$ signal is connected to the shift register's $S1$ input. Normally, when $\overline{IOSTRB}$ is not asserted, the shift register executes a parallel load that results in the output connected to the $\overline{WR}$ of the display being inactive.

Once $\overline{IOSTRB}$ is asserted, the shift register starts a self-terminating (by using feedback to the serial input) shift sequence that results in the appropriate assertion of $\overline{WR}$ and $\overline{ALFRDY}$, the latter used by the PAL to generate $\overline{XRDY}$ at the end of the display write cycle.

Although only $\overline{IOSTRB}$ has to be active to initiate a write to the alphanumeric display, the highest bit of the expansion bus address has to be correct to ensure that the $\overline{XRDY}$ is generated correctly so that the transaction may complete. The ABEL PAL source file in Appendix C shows how $\overline{XRDY}$ is produced as a result of either a display-write cycle or a FIFO transaction.

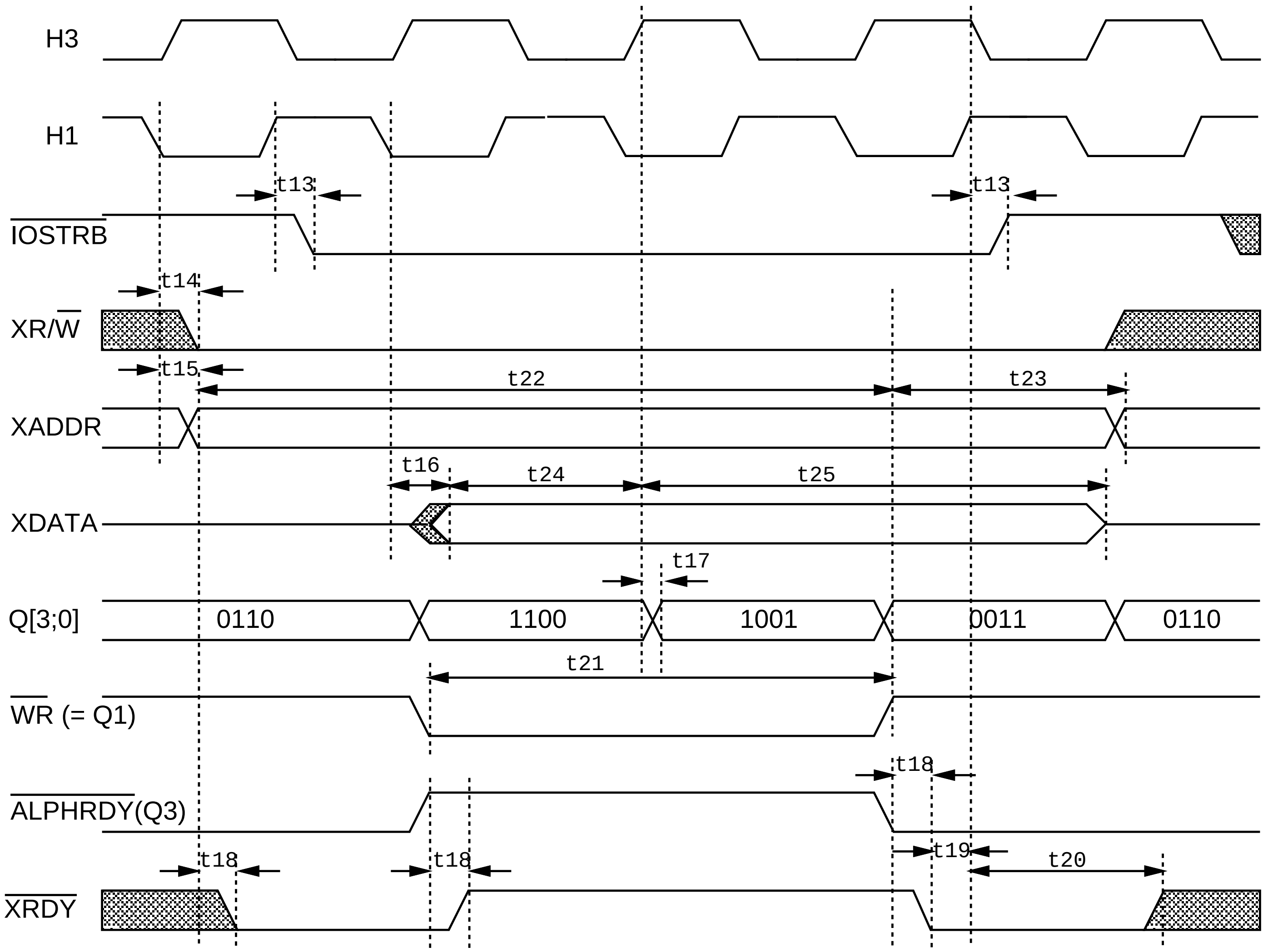

Figure A-10: Alphanumeric Display Write Cycle

Appendix B

Schematics

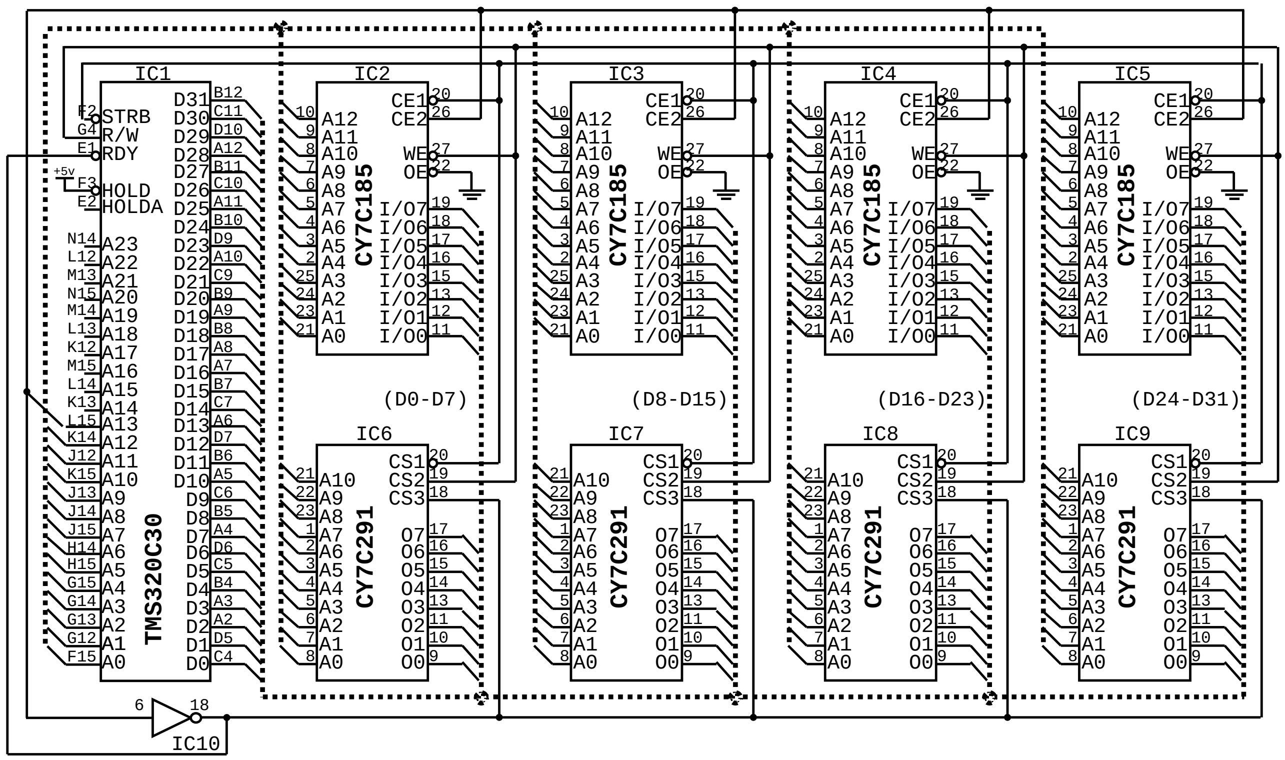

Figure B-1: Primary Bus Schematics

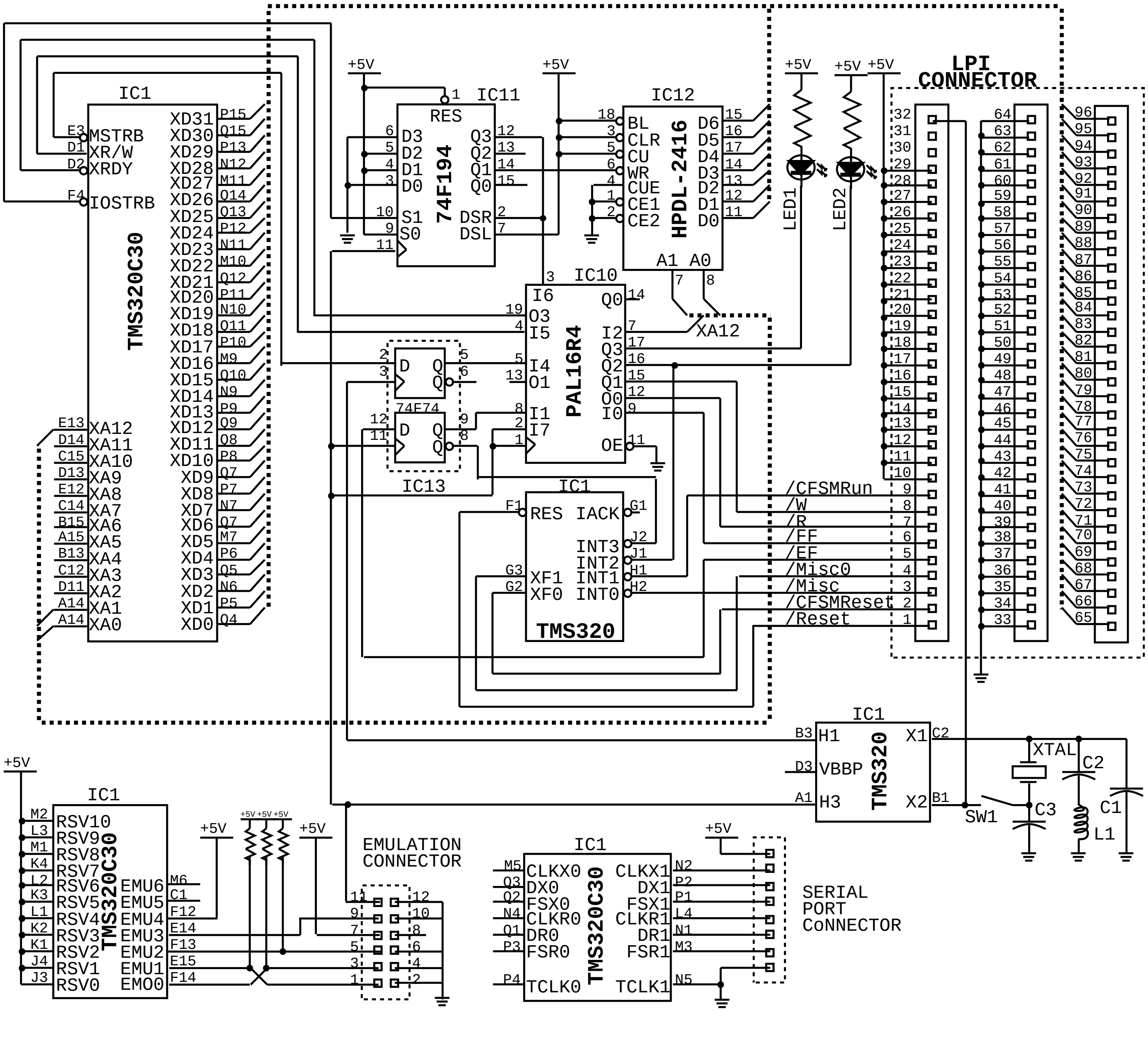

Figure B-2: Expansion Bus and Miscellaneous Circuitry Schematics

Appendix C

Source Code Files

C.1 Source Abel PAL File

```
module NuPal(dev)
FLAG    '-R2';
title 'Miscellaneous function pal for numesh processor board.
Karim Abdalla   Massachusetts Institute of Technology   2/17/90'

        IC10    device  'P16v8r';
"@IFB(?dev)     {'p16r4';};"
"@IFNB(?dev)    {'?dev';};"


@ALTERNATE      "Alternate operators symbols"

"Input pins"
        CLK     pin 1;          "State clock is H3 of TMS320"
        H3      pin 2;          "Same as pin 1 but useable in equations"
        _ALFRDY pin 3;          "Alpha Display ready signal"
        XRnW    pin 4;          "Expansion Read not Write"
        _MSTRBS pin 5;          "MSTRB synchronized to H1 of TMS320"
        A22     pin 6;          "Highest order address bit of primary bus"
        XA12    pin 7;          "Expansion address selects FIFO/ALPHA"
        _MTS    pin 8;          "FIFO empty flag, synchronized to H3"
        _FULL   pin 9;          "FIFO full flag, unsynchronized"

"Output Enable"
        _OE     pin 11;

"Output pins"
        RD_     pin 12 = 'neg'; "Read FIFO"
        H3D     pin 13 = 'neg'; "Combinationally delayed H3"
        Q1      pin 14 = 'neg'; "State bit"
        Q0      pin 15 = 'neg'; "State LSB also used as WR, Write FIFO"
        FULLS_  pin 16 = 'neg'; "FIFO full, internally synchronized to H3"
        Q2      pin 17 = 'neg'; "FIFO empty and state MSB"
        A22B    pin 18 = 'neg'; "A22 inverted - selects beteween ROM/RAM"
        XRDY_   pin 19 = 'neg'; "Expansion bus ready signal back to TMS320"


"Expansion bus address map"
        ALPHA   MACRO   {(XA12 == 1)};
        FIFO    MACRO   {(XA12 == 0)};

"Make positive logic symbols"
        EMTY    MACRO   {/_MTS};
        FULL    MACRO   {/FULLS_};
        RD      MACRO   {/RD_};
        WR      MACRO   {/WR_};
        XRDY    MACRO   {/XRDY_};
        ALFRDY  MACRO   {/_ALFRDY};
        MSTRBS  MACRO   {/_MSTRBS};
```

```
"Make up symbols for expressions"
        FREAD   MACRO   {(MSTRBS * XRnW * FIFO)};       "FIFO read attempt"
        FWRITE  MACRO   {(/XRnW * FIFO)};              "FIFO write attempt"
        VFWRITE MACRO   {(FWRITE * /FULL)};     "Valid FIFO write attempt"

        H,L,C,Z = 1,0,.C.,.Z.;
        TRUE    = 1;

"FSM states"
        IDLE    = ^b111;        "7"
        READ    = ^b101;        "5"
        DREAD   = ^b001;        "1"
        WRITE   = ^b110;        "6"
        EMPTY   = ^b011;        "3"

        TRAP0   = 0;
        TRAP2   = 2;
        TRAP4   = 4;

EQUATIONS
        H3D     = H3;
        A22B    = /A22;
        XRDY    = ALFRDY * ALPHA;
        FULLS_  := _FULL;

STATE_DIAGRAM [Q2,Q1,Q0];

        STATE IDLE:
                RD =    FREAD * /EMTY */H3D;
                CASE    FREAD * /EMTY: READ;
                        VFWRITE: WRITE;
                        /VFWRITE * EMTY: EMPTY;
                        /EMTY * /(FREAD + VFWRITE): IDLE;
                ENDCASE;

        STATE READ:
                RD      = TRUE;
                XRDY    = TRUE;
                CASE     EMTY: EMPTY;
                        /EMTY: IDLE;
                ENDCASE;

        STATE WRITE:
               "WR      = TRUE; Implicit since WR_ is the state bit"
                XRDY    = TRUE;
                CASE     EMTY: EMPTY;
                        /EMTY: IDLE;
                ENDCASE;

        STATE EMPTY:
```

```
                CASE    VFWRITE: WRITE;
                         EMTY * /VFWRITE: EMPTY;
                        /EMTY * FREAD: DREAD;
                        /EMTY * /(FREAD + VFWRITE): IDLE;
                ENDCASE;

        STATE DREAD:
                RD      = TRUE;
                GOTO READ;

        STATE TRAP0:
                GOTO IDLE;
        STATE TRAP2:
                GOTO IDLE;
        STATE TRAP4:
                GOTO IDLE;


end NuPal;
```

C.2 Abel Generated Document File

```
                                                                          Page 1
ABEL(tm) 3.00a  -  Document Generator             17-Feb-90 02:38 PM
Miscellaneous function pal for numesh processor board.
Karim Abdalla   Massachusetts Institute of Technology   2/17/90
Equations for Module NuPal

Device IC10


- Reduced Equations:

    H3D = !(!H3);

    A22B = !(A22);

    XRDY_ = !(!Q0 & Q1 & Q2 # Q0 & !Q1 & Q2 # XA12 & !_ALFRDY);

    FULLS_ := !(!_FULL);

    RD_ = !(Q0 & !Q1 & Q2
           # Q0 & !Q1 & !Q2
           # !H3D & Q0 & Q2 & !XA12 & XRnW & !_MSTRBS & _MTS);


    Q2 := !(Q0 & Q1 & !Q2 & !XA12 & XRnW & !_MSTRBS
           # !FULLS_ & Q0 & Q1 & !_MTS
           # Q0 & Q1 & XA12 & !_MTS
           # Q0 & Q1 & XRnW & !_MTS
           # !Q0 & Q1 & Q2 & !_MTS
           # Q0 & !Q1 & Q2 & !_MTS);


    Q1 := !(Q0 & !Q1 & !Q2 # Q0 & Q1 & !XA12 & XRnW & !_MSTRBS & _MTS);

    Q0 := !(FULLS_ & Q0 & Q1 & !XA12 & !XRnW);
```

```
ABEL(tm) 3.00a  -  Document Generator            17-Feb-90 02:38 PM
Miscellaneous function pal for numesh processor board.
Karim Abdalla   Massachusetts Institute of Technology   1/18/90
Chip diagram for Module NuPal

Device IC10


                           P16V8R

                 ----------\       /----------
                 |          \     /          |
                 |           -----           |
             CLK |  1                    20  | Vcc
                 |                           |
              H3 |  2                    19  | XRDY_
                 |                           |
         _ALFRDY |  3                    18  | A22B
                 |                           |
            XRnW |  4                    17  | Q2
                 |                           |
         _MSTRBS |  5                    16  | FULLS_
                 |                           |
             A22 |  6                    15  | Q0
                 |                           |
            XA12 |  7                    14  | Q1
                 |                           |
            _MTS |  8                    13  | H3D
                 |                           |
           _FULL |  9                    12  | RD_
                 |                           |
             GND | 10                    11  | _OE
                 |                           |
                 |                           |
                 -----------------------------


end of module NuPal
```

C.3 Source Code for Main Diagnostics

```
DISP_S  $MACRO  S
        LDI     :S:,AR0
        LSH     16,AR0
        LSH     -16,AR0
        ADDI    R7,AR0
        CALL    DISP_STRING
        $ENDM

        .data
DATA:
TXT     .word   RESET
        .text
        .global RESET

RESET:  LDP     DATA            ;data page register is data section.
        LDI     DP,R7
        ASH     16,R7
;       LDI     0Fh,IE          ;Enable INT0 to INT3 (not yet globally)
*       Set up wait state generation for both busses

        LDI     @BUSCTRL,R0     ;See fields in data section
        LDI     @PBUSREG,AR0
        STI     R0,*AR0
        XOR     1000b,R0        ;RDYint=RDY for XBUS
        STI     R0,*-AR0(4)

        .data
BUSCTRL .field  0,1             ;(Read only field)
        .field  1,1             ;NO HOLD
        .field  0,1             ;no HIZ
        .field  01b,2           ;RDYint=RDYwtcnt
        .field  1,3             ;1 wait state (software only)
        .field  00010b,5        ;MSBs defining a bank: A23;A22

PBUSREG .word   808064h
XBUSREG .word   808060h
RAM1    .word   809C00h

        .text
*       Set up system stack (AR7) in internal RAM block 1
        LDI     @RAM1,SP

        DISP_S  LIFE$           ;Print herald on display
        CALL    TESTRAM         ;Test RAM
        DISP_S  S4$
;       LDI     2000h,ST        ;Enable global interrupts
;       LDI     0Fh,IF          ;Test the interrupt routines thru software
LOOP:   IDLE                    ;Allow real interrupts to test XF0,XF1
        B       LOOP            ;Ad nauseum
```

```
INT0:
        LDI     00100010b,IOF   ;Sets XF1;XF0 to 00b
        DISP_S  INT0$           ;Display "INT0"
        RETI
INT1:
        LDI     00100110b,IOF   ;Sets XF1;XF0 to 01b
        DISP_S  INT1$           ;Display "INT1"
        RETI
INT2:
        LDI     01100010b,IOF   ;Sets XF1;XF0 to 10b
        DISP_S  INT2$           ;Display "INT2"
        RETI
INT3:
        LDI     01100110b,IOF   ;Sets XF1;XF0 to 11b
        DISP_S  INT3$           ;Display "INT3"
        RETI

ERROR:                          ;remaining TRAP/INT vectors point here
        DISP_S  ERROR$
        RETI


        .data
INT0$   .string "INT0",0
INT1$   .string "INT1",0
INT2$   .string "INT2",0
INT3$   .string "INT3",0
ERROR$  .string "ERR!",0

        .text
***     R0,R1 , AR0,AR1 are scratch for all subroutines to use



ramstrt .set    400000h
ramend  .set    401fffh
alpha   .set    805000h

*       TEST ALPHANUMERIC DISPLAY

        .data
LIFE$   .string "LIFE",0
ALPHA   .word   alpha
SPACES  .word   '    '

        .bss    BSS,0
        .bss    ALPHA_STATE,1

        .text
DISP_C:
```

```
*       scrolls display one character to the left and
*       shifting in R0 as rightmost character.
*       then pauses for a fraction of a second.
        LDI     @ALPHA,AR1
        LDP     BSS
        LDI     @ALPHA_STATE,R1
        LSH     8,R1
        OR      R0,R1
        STI     R1,@ALPHA_STATE
        LDP     DATA
        STI     R1,*AR1++
        LSH     -8,R1
        STI     R1,*AR1++
        LSH     -8,R1
        STI     R1,*AR1++
        LSH     -8,R1
        STI     R1,*AR1++
PAUSE:
*       Pause for a fraction of a sec ...
        LDI     @PAUSECOUNT,RC
        RPTB    RTURN
        NOP
        NOP
        NOP
RTURN:  NOP
        RETS

        .data
PAUSECOUNT
        .word   00080000h
        .word   08000000h       ;max_pause*8*2(cycles/instr) =:= 2E12
        .word   1000000000000          ;2E12 cycles = approx .25 secs

        .text
D_CLEAR $MACRO
        LDI     @ALPHA,AR1
        LDI     @SPACES,R0
        LDP     BSS
        STI     R0,@ALPHA_STATE
        LDP     DATA
        LDI     ' ',R0
        STI     R0,*AR1++
        STI     R0,*AR1++
        STI     R0,*AR1++
        STI     R0,*AR1++
        $ENDM

DISP_STRING:

*       display null terminated packed strings pointed to by AR0.
*       by repeatedly calling DISP_C for each character
```

```
        PUSH    R2              ;R2 holds current word from packed string
        PUSH    R3              ;R3 is character counter within string word
        D_CLEAR

DISP_WORD:
        LDI     *AR0++,R2       ;Load unshifted word of string
        LDI     4,R3            ;Initialize # characters in word to 4
DISP_CHAR:
        LDI     R2,R0           ;copy current (shifted) word into R0
        AND     0FFh,R0         ;Current character is lowest byte of word
        BZ      STRING_DONE     ;If null then end of string
        CALL    DISP_C          ;otherwise display the character
        LSH     -8,R2           ;and shift the word to get next character
        SUBI    1,R3            ;decrement the character counter
        BNZ     DISP_CHAR       ;repeat for 4 characters with a word

        B       DISP_WORD       ;Repeat for the next word of string

STRING_DONE:
        POP     R3
        POP     R2
        RETS

TESTRAM:
        .data
RAMSTRT .word   ramstrt
RAMEND  .word   ramend

        .text
*       R2 is error flag for a single pass of all RAM
*       R3 is mask for a single bit of primary data bus
*       R4 is bit counter
*       R5 is error flag for all passes

        LDI     1,R3            ;initially test lowest bit
        LDI     32,R4           ;initialize bit counter
        LDI     0,R5            ;clear global error

TESTBIT:
        LDI     0,R2            ;clear single pass error
        LDI     @RAMSTRT,AR0    ;AR0 points to start of RAM
        RPTS    ramend-ramstrt  ;for all ram locations ...
        STI     R3,*AR0++       ;fill the ram with contents of R3

        LDI     @RAMSTRT,AR0    ;AR0 points to start of RAM
        LDI     ramend-ramstrt,RC
        RPTB    ENDBIT          ;for all ram locations ...
        SUBI3   *AR0++,R3,R0    ;If RAM location != R5, R0 becomes nonzero
ENDBIT: OR      R0,R2           ;OR errors for all locations into R2
        BZ      OVER            ;kludge - simulator screws up CALLNZ
        CALL    RAMERROR        ;if R2 is non zero we have data path error
```

```
OVER:   LSH     1,R3            ;test the next most significant bit
        SUBI    1,R4            ;decrement the bit counter
        BNZ     TESTBIT         ;if not done, test the next bit

        LDI     S1$,AR0         ;anticipate "success" message
        CMPI    0,R5            ;check if global error
        LDINZ   S3$,AR0         ;if so, change to "no success" message
        LSH     16,AR0
        LSH     -16,AR0
        ADDI    R7,AR0
        B       DISP_STRING     ;display exit condition and (tail) return

RAMERROR:
        LDI     1,R5            ;flag global error
        DISP_S  S2$             ;display error message
        LDI     R4,R0           ;get bitcount
        SUBRI   32,R0           ;subtract from 32 for so that LSB = 0
        LSH     -4,R0           ;get higher nybble first
        ADDI    '0',R0          ;get ASCII code
        CALL    DISP_C          ;and display
        LDI     R4,R0           ;get bitcount again
        SUBRI   32,R0           ;subtract from 32
        AND     0Fh,R0          ;get lower nybble this time
        ADDI    '0',R0          ;get ASCII code
        B       DISP_C          ;display and return (tail call)



        .data
S1$     .string "RAM OK!!",0
S2$     .string "RAM ERROR AT D",0
S3$     .string "RAM CHECK FAILURE!",0
S4$     .string "HELLO STEVE, SHALL I SING YOU A SONG?...
        .string "CHECK OUT MY CHARACTER SET! ""#$%&'<>*+,-./0123456789",
        .string 3AH,3BH,3CH,3DH,3EH,"?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_",0

        .end
```

C.4 Source Code for FIFO Diagnostic

```
        .text
        .global START

START   ldp     DATA
        ldi     @XBUS,ar0
        ldi     222h,r0
        sti     r0,*ar0
        ldi     22ah,r0
        sti     r0,*+ar0(4)
        ldi     @STKADD,sp

        ldi     010b,iof        ;xf0 = 0
        nop
        ldi     111b,iof        ;xf0 = 1

        ldi     @FIFO,ar1
        ldi     5,r5

        ldi     r5,r1
l0      sti     r1,*ar1
        addi    -1,r1
        cmpi    0,r1
        bnz     l0

        ldi     @FIFBUF,ar2
        ldi     r5,r1

l1      ldi     *ar1,r0
        sti     r0,*ar2++
        addi    -1,r1
        cmpi    0,r1
        bnz     l1

l2      br      l2


        .data
DATA
XBUS    .word   808060h
FIFO    .word   800000h
FIFBUF  .word   400400h
STKADD  .word   809c00h
```