Chapter 6

An Automatic Compiler

The translation procedure presented in Chapter 5 is a straightforward translation from an OCCAM program to an asynchronous circuit. Using this procedure a designer can specify a system by writing a program and then follow the steps to produce a circuit that has the same behavior. Along the way, however, he is likely to make simple clerical mistakes that can result in incorrect circuits, and extended debugging of supposedly correct circuits. A better method is to develop a tool similar to a software compiler to apply the translation and optimizing transformations automatically.

6.1 System Description

I have built a system to perform automatically the translation and optimizations described in previous chapters. The system is implemented in Kyoto Common LISP, or KCL [108], which is an implementation of the Common LISP language described in [93]. KCL was chosen as the implementation language because LISP is particularly well suited for building prototypes quickly. The program is a "quick-and-dirty" prototype which was written to demonstrate the feasibility of automating the entire translation process. Although the program could include more features, and could be more user-friendly, it has successfully compiled every example presented in this dissertation in addition to many others not presented, including class projects by graduate students studying asynchronous VLSI [92, 107, 31]. The output from the compiler has been converted into source code for simulation and has also been used as input to an automatic place and route system [3, 4] for automatic assembly into chip descriptions that were then fabricated and tested.

6.1.1 Program Input

The fact that the compiler is written in LISP, and that LISP is particularly good at parsing input that uses its own syntactic conventions, results in the "LISPy-OCCAM" syntax presented in Section 3.3. This allows the KCL reader to parse and print the OCCAM expressions easily. It also allows the full power of LISP to be used in the source language descriptions. For example, LISP variables, constants, and macros may be used as parameters to OCCAM processes, or for customizing the widths of OCCAM variables and channels for different situations. Many users also find LISP-style notation easier to read and understand than the indentation-based block structuring syntax used in conventional OCCAM [42].

When writing programs using this syntax it has been assumed that any arithmetical or logical operation on the data defined for OCCAM in [42] is available for use by the programmer. In practice, however, the set of circuit modules that the compiler has available for its target circuit is limited to the circuits implemented and installed in the standard cell library. Conversely, if the programmer needs to use some specialized data manipulation function that is not in the standard definition of OCCAM, and he is willing to implement that function as a new circuit module, that function should be made available in the source program. A facility for defining the interface to gates and functions used in a particular program is added to the syntax of the source language. Any data manipulation operations used in a program must be defined before use so that the compiler is aware of the conditions on their use.

For purposes of the prototype system, there are two classes of data manipulation modules: *gates* and *functions*. The distinction in this case is whether the circuit modules include a storage element. A *gate* is a combinational circuit with no storage and a *function* includes both a function on the input data, and a storage element to hold the result. Note that this means that a gate is simply inserted into the data path as a combinational circuit. A function, by virtue of its storage element, also includes a request-acknowledge control interface that becomes part of the control path. This is explained further below.

New Gates

Gates are defined using a LISP function called *Add-Gate* to add the pertinent information about the new gate to the compiler's tables. The arguments are:

(Add-Gate *string Interface-list*)

The string is the name by which the new gate is called in the program, and the interface-list is a list of the widths of the interface paths in the order in which they will be called. Gates may have any number of input data paths but are limited to a single output data path. The last number in the interface list must be the width of the output path. For example, a simple two-input NAND gate would be defined by:

(Add-Gate “NAND” ’(1 1 1))

This expression shows that the gate has two single-bit inputs and a single bit output. A gate to detect that a 16-bit data word is zero might have a 16-bit input and a single bit output to signal the zero condition and be defined as:

(Add-Gate “zero-detect” ’(16 1))

Expressions using any of the defined gates may be used wherever an expression is allowed, provided that the delay through the gates is accounted for in the bundling constraint computations. Gates to implement common logic functions of two Boolean variables such as AND, NAND, OR, NOR, XOR, XNOR, and NOT are implemented in the standard circuit library. Their definitions are built into the compiler and need not be added explicitly.

New Functions

New functions in this version of OCCAM are defined using a function similar to that used for new gates. This function is defined as:

(Add-Function *string Interface-List*)

The purpose is the same as for gates: to define a new module that can be used in the assembly of circuits translated from the source code. The difference is that while new gates are simple combinational circuits, functions include a register of some sort to store the result of the function. Like gates, the last number in the Interface-list is the width of the output data path, but in this case it is also the width of the output storage register. For example, an ALU like the one shown in Figure 4.32 with two 16-bit inputs, a 5-bit control input, and 16-bit output could be defined as:

(Add-Function “alu16” ’(16 16 5 16))

The storage register at the output data path is the means by which the result of the function is made available to the rest of the system. Other circuit modules may read the value of that register in the same way as the value of any variable stored in a register is accessed. A value may be written into this function variable, however, only by the function itself. This leads to a new class of data object called a *Funcvar*. A *Funcvar* is defined in the source code and associated with a function defined by a call to *Add-Function*. This *Funcvar* is available for read access in any situation where a normal *Var* is acceptable. However, because it can be written only by the function module of which it is a part, it cannot be written to using the standard *Set* primitive. A new primitive is added, called *SetFunc*, to allow functions to set the values of their output registers.

The circuit module corresponding to the newly defined function must have a standard transition request-acknowledge control interface. The data must obey the bundling constraint with respect to these control signals. Upon receipt of a request transition, the function computes its value and stores the result in the register. When the output of the register is stable and valid as all the receivers of that signal, the module makes a transition on its acknowledge line. The circuit module should also respond to a master clear signal by clearing its internal state and make itself ready to accept a request from the environment. The interface to the function module looks just like the interface to a Register module except that a function might have multiple input data paths.

The function definition is provided to model circuits that must include a storage register to meet the specification on the data path required by the two-phase control style. That is, the module must produce an output in response to a request transition, and hold that output stable until the next request transition. An example of a data path circuit that is implemented as a new function is the carry-completion adder described in Chapter 4. The carry-completion adder itself is a four-phase circuit: the first request initiates the addition, and the second request clears and precharges the circuit to prepare for the next addition. In order to present a two-phase interface to the rest of the system, a circuit like that in Figure 4.32 or 4.33 is used. This uses a register to capture the output of the adder during the second phase of the four-phase handshake. The external interface is a two-phase request-acknowledge handshake. When defined for use in this compiler, the carry-completion adder is defined as a function because of the storage register that is part of the implementing circuit.

The complete syntax accepted by the compiler showing the new *Funcvar* definitions is shown in Figure 6.1.

Data Types

Recall that the only data type used in this dialect of OCCAM is a simple word consisting of some number of bits. Variables and channels may be defined with any number of bits, with a *Subseq* operator used to access contiguous subparts of a variable or channel. The number of bits in a variable is specified by giving the variable width in angle brackets after its definition. If no width is given, the variable is assumed to be a single-bit Boolean value. The width of the data path of a channel is specified in the same way. If a channel is used for synchronization only with no data being passed it must have a width of zero.

Alias Definitions

In addition to adding new gates and functions, a facility is defined in the prototype compiler to use aliased names for data elements. This allows commonly used expressions to be given a more meaningful name to improve program readability. For example, it is common

```
Program        =  Process
Process        =  Primitive | Construct
Primitive      =  Assignment | Input | Output | Skip
Assignment     =  (Set  Variable Expression )
                  | (SetFunc  Funcvar Func-call )
Input          =  (?  Channel [ Variable ] )
Output         =  (!  Channel [ Expression ] )
Skip           =  (Skip)
Construct      =  (Seq  Decl {Process} ) | (Par  Decl {Process} )
                  | (Alt  {Guarded-Proc} ) | (Lex-Alt  {Guarded-Proc} )
                  | (If  {Cond-Proc} )
                  | (While  Cond Process ) | (While  Cond Implicit-Seq )
                  | (Block  Decl Process ) | (Block  Decl Implicit-Seq )
                  | ( macro-name { var-name | chan-name} )
Implicit-Seq   =  {Process}
Guarded-Proc   =  ( Guard Process) | ( Guard Implicit-Seq )
Guard          =  (  Cond Input ) | ( Cond Skip )
Cond-Proc      =  ( Cond Process) | ( Cond Implicit-Seq )
Decl           =  ( {Decl-Part} )
Decl-Part      =  (Chan  {chan-name [ < int > ]} )
                  | (Var  {var-name [ < int > ] } )
                  | (Funcvar  { ( funcvar-name function-name ) } )
Variable       =  var-name | (Subseq var-name int int )
Channel        =  chan-name | (Subseq chan-name int int )
Funcvar        =  funcvar-name
Expression     =  Variable | Funcvar | ( gate-name { Expression })
Func-call      =  ( function-name { Expression } )
Cond           =  Boolean-expression
Macro-Def      =  (Process macro-name Decl Process )
                  | (Process macro-name Decl Implicit-Seq )
```

Figure 6.1: Source Language Syntax for Prototype Compiler

to base control decisions on individual bits extracted from a command word. Rather than repeat the *Subseq* expression each time the control bit is tested, it can be given a name that is more descriptive of the purpose of that bit. Syntax for these definitions is:

(define *string expression*)

For example, if bit 3 of a variable called "control" is tested to determine if output should be generated, this bit can be aliased to a more meaningful name using:

```
(define "gen-output"  '(subseq control 3 3))
```

Thereafter, the third bit of a variable named "control" may be referred to as "gen-output."

6.1.2 Compiling the Circuit

Once the OCCAM code has been written, it can be compiled into an initial circuit by calling the *Compile-Occam* function. This function takes one argument which may be either an OCCAM process, or the name of a file that contains the source code to be compiled. The result is that the initial translation from code to circuit presented in Section 5.1 is carried out on the source code. The result is an initial circuit represented internally as a graph. The nodes of this graph are the circuit modules and the arcs are the wires that connect them. This initial circuit, although possibly containing many superfluous circuit modules, will correctly implement the behavior of the source code.

The compiler reports any errors that are encountered in the translation process, and, if the compilation is successful, reports the wire index of the final acknowledge wire for the compiled process. This circuit can be used as is, but application of the optimizing transformations will almost always result in significant improvement of the circuit.

6.1.3 Circuit Optimization

Once the compiler has made the initial translation from OCCAM source code into circuits, the optimizations described in Section 5.2 can be applied automatically to that circuit. The algorithm for applying the optimizations is particularly simple in this prototype compiler. The compiler searches the graph to find appropriate situations, and then replaces the circuit by modifying the circuit graph. If it finds a subcircuit that matches the template for an optimization, that piece of the graph is replaced with a new piece. The optimization process repeats until it traverses the graph completely and makes no changes. The prototype compiler reports its progress by reporting each of the optimizations as it is applied. All of the optimizations described in Section 5.2, except channel splitting, are performed automatically by calling the *Optimize-Circuit* function. This function takes no arguments and applies the optimizations to whatever circuit is currently represented internally in the compiler.

Each optimization is applied by its own LISP function. Each optimization function traverses the graph looking for a particular situation. When it finds such a situation, it modifies the graph and returns. Additional optimizations may be added to the system by writing a new function for the desired optimization and including it on a list of functions to be called during the optimization phase. The current set of optimizations is not sensitive to order of application. New optimizations might, of course, require that they be applied in a particular order to be effective and the optimization phase of the compiler could be adjusted accordingly.

6.1.4 Output Formats

Once the circuit has been compiled and optimized by the system, it must be printed in a form suitable for simulation or fabrication of the circuit. The internal representation of the circuit is as a graph where the nodes of the graph are circuit modules and the arcs are the wires that connect them. The output driver of the prototype simply prints out this graph, with its syntax suitably modified for whatever purpose the output file is intended. For this prototype compiler there are separate output formats for simulation and fabrication. Adding a new output driver for some different output syntax involves writing a LISP function to traverse the circuit graph and massage it into whatever form is desired.

Netlist Output Format

The output format suitable for simulating the circuit is based on Terman's Net language [96]. This language is a simple network description language for VLSI circuits that uses a LISP-like syntax and allows macro-based hierarchical description of networks of transistors. A function called *Print-Netlist* will print the circuit generated by the translation procedure using this Net notation to describe how the circuit modules are connected. Each circuit module that is mentioned must also have a Net macro description in a library which describes the circuit in terms of other macros, and ultimately, transistors.

Using a standard tool, the Net description can be converted to a SIM-format file which is a flat, non-hierarchical list of transistors and their connections. This format can be used to simulate the circuit at the switch level using any of a number of popular simulators such as RNL [97], IRSIM, or COSMOS [11].

Fusion Output Format

In addition to netlist output, the prototype can also print the circuit in a form that the Fusion place and route system can use as input [3, 4]. This allows the generated circuit to be assembled automatically into a form suitable for fabrication as an integrated circuit. The function that produces Fusion-compatible output is called *Print-Fusion*. The Fusion service will place and route the modules that make up the system, generate a pad ring, and connect external signals to the pads. The output format assumes that each of the target circuit modules is instantiated as a layout file and has a CIF description. In addition, each module must have a Fusion description that identifies the ports of the module, their location, and their type. This description can be generated automatically from the CIF description [3]. Given the layout of each module, and the Fusion specification file, the Fusion place and route system returns a CIF file of the entire circuit, including connections to pads.

6.2 Examples

6.2.1 FIFO

The first example is the familiar 8-bit wide, two-word deep, FIFO buffer used as an example of the compilation technique in Chapter 5. The code from Figure 5.30 was compiled using the prototype compiler. The compiler output generated when performing the initial translation followed by the optimization stage is shown in Figure 6.2. The initial translation to a circuit with the *Compile-Occam* function was successful as seen in the upper part of the figure. The optimizations are then applied using the *Optimize-Circuit* function. This function reports its progress by listing each of the optimizations that are applied. The result is a circuit using only six circuit modules. Notice that because the buffer macros are written as non-terminating loops, the final acknowledgment is eliminated. Once started with a *Start* signal, this buffer will pass data from input to output continuously and never terminate and acknowledge.

After the circuit has been compiled and optimized, a netlist of modules and their connections can be printed using the *Print-Netlist* function. An output trace from the compiler is shown in Figure 6.3. The *include* statement points to a library file that contains Net macros for all of the circuit modules known to the compiler. This file must be included if the netlist file is to be used to generate a transistor-based SIM file.

The *simulate-modules* variable is a flag that allows some circuit modules to have separate simulation and fabrication versions. A Q-flop, for example, contains what is essentially an analog circuit to sample the input and cannot be simulated using a switch level simulator. One solution is to use conditional compilation in the netlist file to allow for a simpler circuit that will simulate the behavior of the Q-flop, and a different circuit that is actually fabricated. Careful transistor level simulation is required of the second version to have confidence that the circuit will work when fabricated. The switch level version need not be concerned with the more exotic functionality of the Q-flop as long as it performs the basic flip-flop function.

```
>(compile-occam "fifo.oc")

Compiling code from fifo.oc.....
once through compile-occam - final-ack 6
....done.
NIL
>(optimize-circuit)

before optimization..... 23 modules
removing lone enable enable.69
removing lone enable enable.50
removing lone enable enable.38
removing lone enable enable.19
removing single input call call.70....
removing single input call call.62....
removing single input call call.54....
removing single input call call.51....
removing single input call call.42....
removing single input call call.23....
removing xor.63, it looks just like a wire
removing xor.55, it looks just like a wire
removing c-element.39, it looks just like a wire
removing c-element.20, it looks just like a wire
Removing select.45 because of TRUE select input
Removing select.26 because of TRUE select input
Last input is eliminated, removing gate c-element.57
Warning: Deleting Final acknowledge, 6

...after optimization 6 modules
NIL
>
```

Figure 6.2: Execution Trace from Compiling the FIFO Example

```
>(print-netlist)
;; netlist file....
(include "/u/elb/thesis/layout/misc/net/netstuff.net")
(setq simulate-modules t)
(node coreq reqdis varreq TEMP while INPUT
      init FALSE TRUE pad )
;;;Wires with special descriptions are:
;;Incoming control wire from channel OUTPUT.5 - 68
;;Outgoing control wire to channel OUTPUT.5 - 36
;;Output data (width 8) to channel OUTPUT.5 - 30
;;Outgoing control signal to channel INPUT.4 - 17
;;Incoming control signal from channel INPUT.4 - 58
;;Input data (width 8) from channel INPUT.4 - 4
;;Initial Request - 3

;;;Making channel OUTPUT.5
;;;Finished with channel OUTPUT.5

;;;Making channel INPUT.4
(c_c-element coreq.58 while.8 varreq.13 )
;;;Finished with channel INPUT.4

;;;Making channel MID.7
(c_c-element reqdis.17 while.27 varreq.32 )
;;;Finished with channel MID.7

;;;Making variable TEMP.30
(c_lat2 TEMP.11 varreq.32 TEMP.30 reqdis.36 0 7 )
;;;Finished with variable TEMP.30

;;;Making variable TEMP.11
(c_lat2 INPUT.4 varreq.13 TEMP.11 reqdis.17 0 7 )
;;;Finished with variable TEMP.11

;;;The rest of the circuit....
(c_xor init.3 reqdis.36 while.8 )
(c_xor init.3 reqdis.68 while.27 )
done.
NIL
```

Figure 6.3: Netlist Output from FIFO Example

A careful examination of the netlist in Figure 6.3 will show that the resulting circuit is exactly the same as the circuit compiled by hand in Chapter 5. The six circuit modules listed in the netlist correspond exactly to the modules shown in Figure 5.33.

If this circuit is satisfactory, the next step is to print a Fusion-format file to send to the place-and-route software. This file is shown in Figure 6.4. This file describes the same six modules as the netlist file but with a different syntax. Also, because this file is intended to describe a fabricated chip, the external interface signals have been assigned to pads. This file can be submitted to Fusion along with the CIF descriptions of the circuit modules to yield a CIF description of the entire chip, ready for fabrication.

6.2.2 Torus Routing Chip

The next example is a switch for cut-through packet routing in a multiprocessor interconnection network similar to the Torus Routing Chip (TRC) described by Dally and Seitz [20]. Each processor in the system has an associated routing circuit, as illustrated in Figure 6.5. The processor communicates to the routing circuit through the *Pin* and *Pout* channels. Each routing circuit routes packets in two dimensions through the *Xin, Xout, Yin,* and *Yout* channels. The circuit must accept packets on any of the three input ports and route them to the appropriate output port depending on address information in the packet header.

This version of the router uses byte-wide data paths on all channels extended with a single tag bit to indicate the end of a packet. As shown in Figure 6.6, address information is contained in the first two bytes of a data packet and specifies relative addresses in the X and Y directions. Packets are routed first in the X direction and then in Y until the destination is reached. The packet is then routed to the processor attached to that router. Because relative addresses are represented using a single byte, a packet sent through this version of the router is limited to 256 hops in either direction.

The code for this simple torus routing process is shown in Figure 6.7 with the top-level macro called *mesh-element*. The mesh-element macro is actually made up of two *router* macros operating in parallel as shown in the block diagram in Figure 6.5. One router accepts packets from the processor and from the X direction and routes that packet along the X direction, or to the other router. The second router takes packets from the first router or from the Y direction, and routes them along the Y direction or to the attached processor. Note that a single mesh-element can be routing a packet both along the X direction and a different packet along the Y direction at the same time.

Packets are routed first in X and then in Y. The first byte is used as the current address byte and decremented as the process reads it. If the result is non-zero, the packet is forwarded in the same direction it came from; if the result is zero, the byte is dropped and the packet changes direction. When the Y address is decremented to zero, the packet has arrived at its destination and is routed to the attached processor.

The initial translation of this mesh-element uses 120 modules. Optimization reduces

```
>(print-fusion)
doit ([ LAMBDA: 1.0
FIRST_FREE_CIF_SYMBOL_NUMBER: 1000
PROJECT: {
'lat2_8_fus' \AT {COLLECT 'INPUT.4'#I FOR I FROM 0 to 7 ; ;
   'varreq.13';
   COLLECT 'TEMP.11'#I FOR I FROM 0 to 7 ; ; 'reqdis.17'};
'xorh_fus' \AT {'init.3'; 'reqdis.36'; 'while.8'; 'mcbar'};
'lat2_8_fus' \AT {COLLECT 'TEMP.11'#I FOR I FROM 0 to 7 ; ;
   'varreq.32';
   COLLECT 'TEMP.30'#I FOR I FROM 0 to 7 ; ; 'reqdis.36'};
'xorh_fus' \AT {'init.3'; 'reqdis.68'; 'while.27'; 'mcbar'};
'celmmc_fus' \AT {'reqdis.17'; 'while.27'; 'varreq.32'; 'mcbar'};
'celmmc_fus' \AT {'coreq.58'; 'while.8'; 'varreq.13'; 'mcbar'};
}
TECHNOLOGY: 'SCMOS'
        MANHATTAN: true
        PAD_LIBRARY: 'TINY_SCN2U_PADS'
        PAD_FRAME: '40PC22X22,40P'
        PADS: {
'1:IN(mcbar)';
'IN(init)';
'IN(reqdis.68)';
'OUT(reqdis.36)';
'OUT(TEMP.30#0)';
'OUT(TEMP.30#1)';
'OUT(TEMP.30#2)';
'OUT(TEMP.30#3)';
'OUT(TEMP.30#4)';
'OUT(TEMP.30#5)';
'OUT(TEMP.30#6)';
'OUT(TEMP.30#7)';
'OUT(reqdis.17)';
'IN(coreq.58)';
'IN(INPUT.4#0)';
'IN(INPUT.4#1)';
'IN(INPUT.4#2)';
'IN(INPUT.4#3)';
'IN(INPUT.4#4)';
'IN(INPUT.4#5)';
'IN(INPUT.4#6)';
'IN(INPUT.4#7)';
}]);
done.
NIL
```

Figure 6.4: Fusion-Format Output from FIFO Example

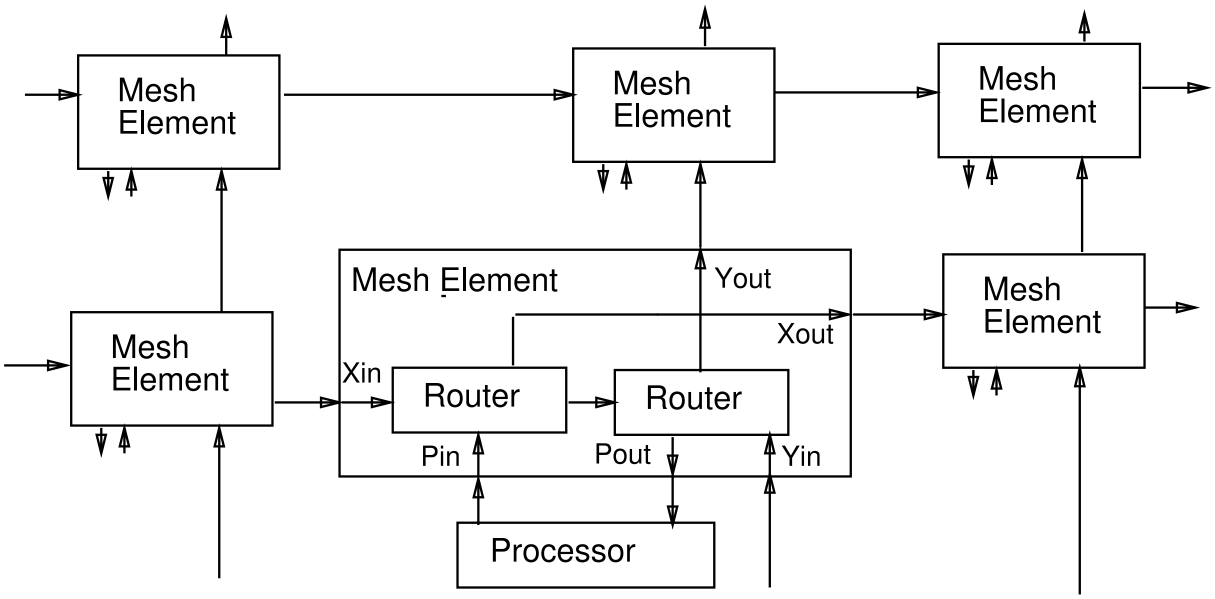

Figure 6.5: Routing Circuit Block Diagram

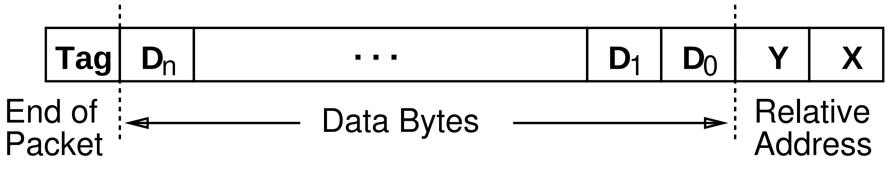

Figure 6.6: Packet Format for the Routing Process

```
;; Start with some definitions
(Defvar *width* 9)                                   ;Define the data path width
(Define "eop" '(subseq data (1- *width*) (1- *width*)))   ;Alias for the tag bit
(Add-Gate "notzero"  '(*width* 1))                   ;Gate to sense non-zero address
(Add-Gate "decr"  '(*width* *width*))                ;A decrement-by-one gate

;; Send data from in to out while data are not tagged
(Process send-data ((Chan in out))
  (While (not eop)                                   ;while data are not tagged
    (! out data)                                     ;send data out
    (? in data))                                     ;and get the next byte
  (! out data))                                      ;send last byte

;; Switch data from input to one of two outputs depending on first byte.
(Process switch-data ((Chan in Aout Bout))
  (If ((notzero data)                                ;if data are non-zero
    (send-data in Aout))                             ;send it out on Aout
  (True                                              ;otherwise
    (? in data)                                      ;drop it, get new byte
    (send-data in Bout))))                           ;and send out on Bout

;; Take data from whichever input channel is ready and route it to
;; either of the two output channels.
(Process router ((Chan Ain Bin Aout Bout))
  (Block ((Var data⟨*width*⟩))                       ;define a local variable
    (While True                                      ;do forever
      (Fair-alt                                      ;choose fairly
        ((True (? Ain data))                         ;data on the A input
          (Set data (decr data))                     ;decrement address
          (switch-data Ain Aout Bout))               ;send out data
        ((True (? Bin data))                         ;data on the B input
          (Set data (decr data))                     ;decrement address
          (switch-data Bin Aout Bout))))))           ;send it on out

;; The router element. Route data streams first in the X and then
;; in the Y direction based on information in header bytes
(Process mesh-element ((Chan Xin Yin Pin Xout Yout Pout))
  (Block ((Chan mid⟨*width*⟩))
    (Par
      (router Pin Xin Xout mid)                      ;route in X direction
      (router mid Yin Yout Pout))))                  ;route in Y direction
```

Figure 6.7: A Simple Routing Process

this number to 64 modules. The schematic for the control part of a single *router* macro is shown in Figure 6.8, and the data path for a single *router* is shown in Figure 6.9. Recall that it takes two *router* macros to make a single *mesh-element* macro as shown in Figure 6.5, so the full mesh-element requires two copies of each schematic.

Because of the large number of I/O pins required to implement the six data channels required of a mesh-element, a version of the mesh-element using a five bit data path consisting of four data bits and a tag bit was compiled for fabrication. This allowed a mesh-element to be implemented on a standard 40-pin DIP. The only change from Figure 6.7 was to change the **width** parameter at the top of the file from 9 to 5. The output from the compiler was sent to Fusion for placement and routing using the MOSIS two-micron SCMOS process and consisted of $\approx$11,000 transistors using the current version of the cell library. The resulting layout from Fusion measured 4164 by 2104 microns and fit into a MOSIS standard 40-pin 4600 by 6800 micron pad frame. The chip was then fabricated through MOSIS and tested. Unfortunately, due to a design rule violation in the Fusion-provided routing, the chip was not fully functional. Luckily, the design rule violation resulted only in the inadvertent connection of three data path wires from different parts of the circuit so although the data path was not correct, the control path could be tested and was found to be fully functional. A photomicrograph of the five-bit version of the mesh-element is reproduced in Figure 6.10.

.

6.2.3 Memory Controller

The last system is an example of using this compiler on a larger OCCAM program. Readers not interested in the particulars of a larger example may safely skip to the last chapter. The example is a memory controller for interfacing asynchronous parts to standard dynamic random access memory (DRAM). This circuit is intended to operate with several DRAM parts to make an asynchronous memory system. This example is based on a memory control circuit designed and built previously at Carnegie Mellon University[90].

The memory controller is a chip that allows asynchronous system parts that use a two-phase bundled data protocol to interface with standard DRAM parts. The block diagram is shown in Figure 6.11. The bundled data paths at the right and left of the figure are eight bits wide. The memory chips are shown connected to the bottom of the block diagram. Timing for the memory chips is implemented using timing models external to the memory controller chip used to model the delay required by the DRAM parts for RAS, CAS and WE signals. These timing models are connected to channel interfaces of the memory controller. For example, the memory controller sends a request to the RAS channel. This sends a signal through a delay to model the RAS duration required by the DRAM. The delayed signal is sent back to the memory contrller as the acknowledge of the RAS channel. This timing model interface is shown in Figure 6.12.

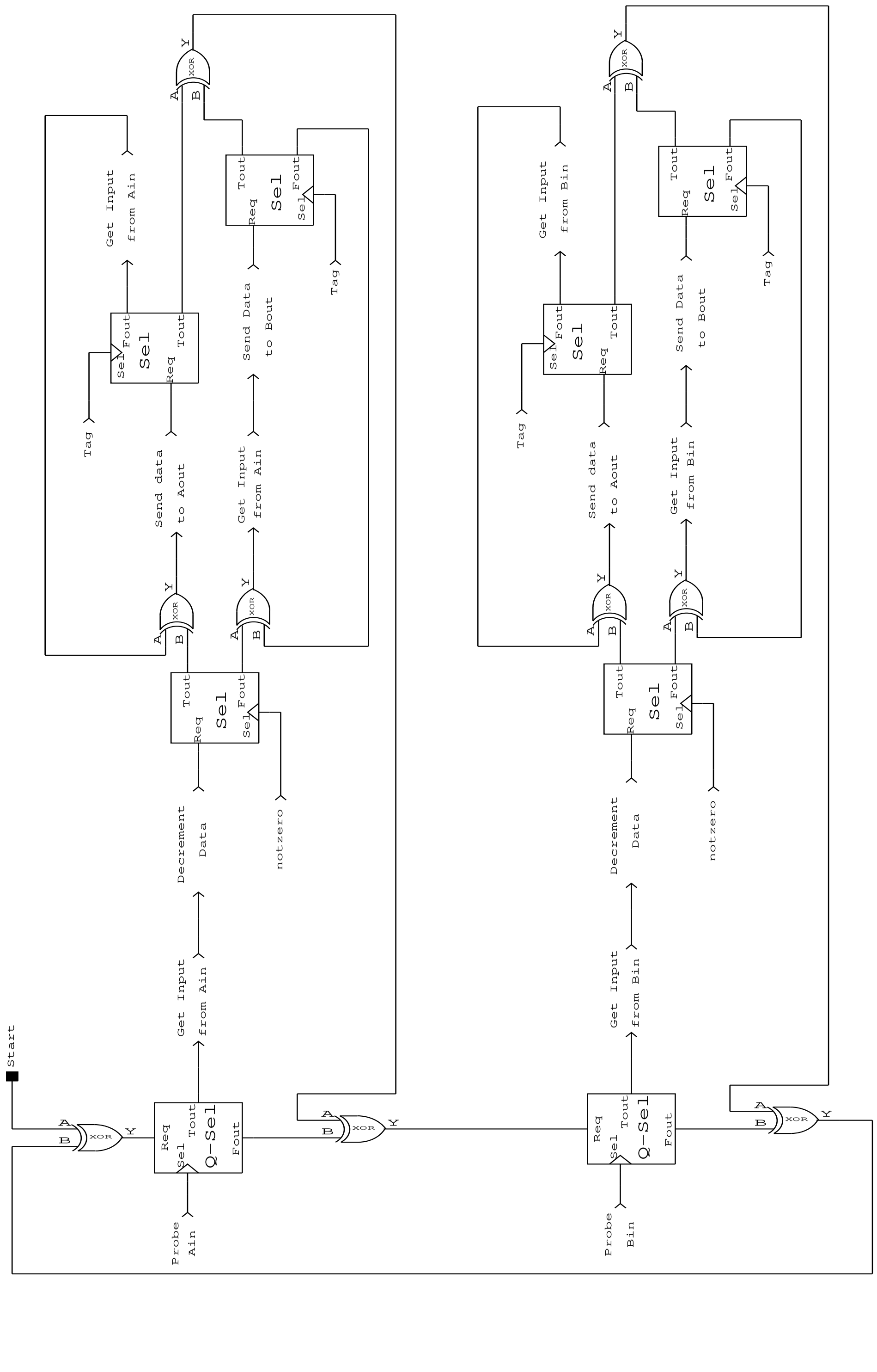

Figure 6.8: Control Paths for a Router Macro

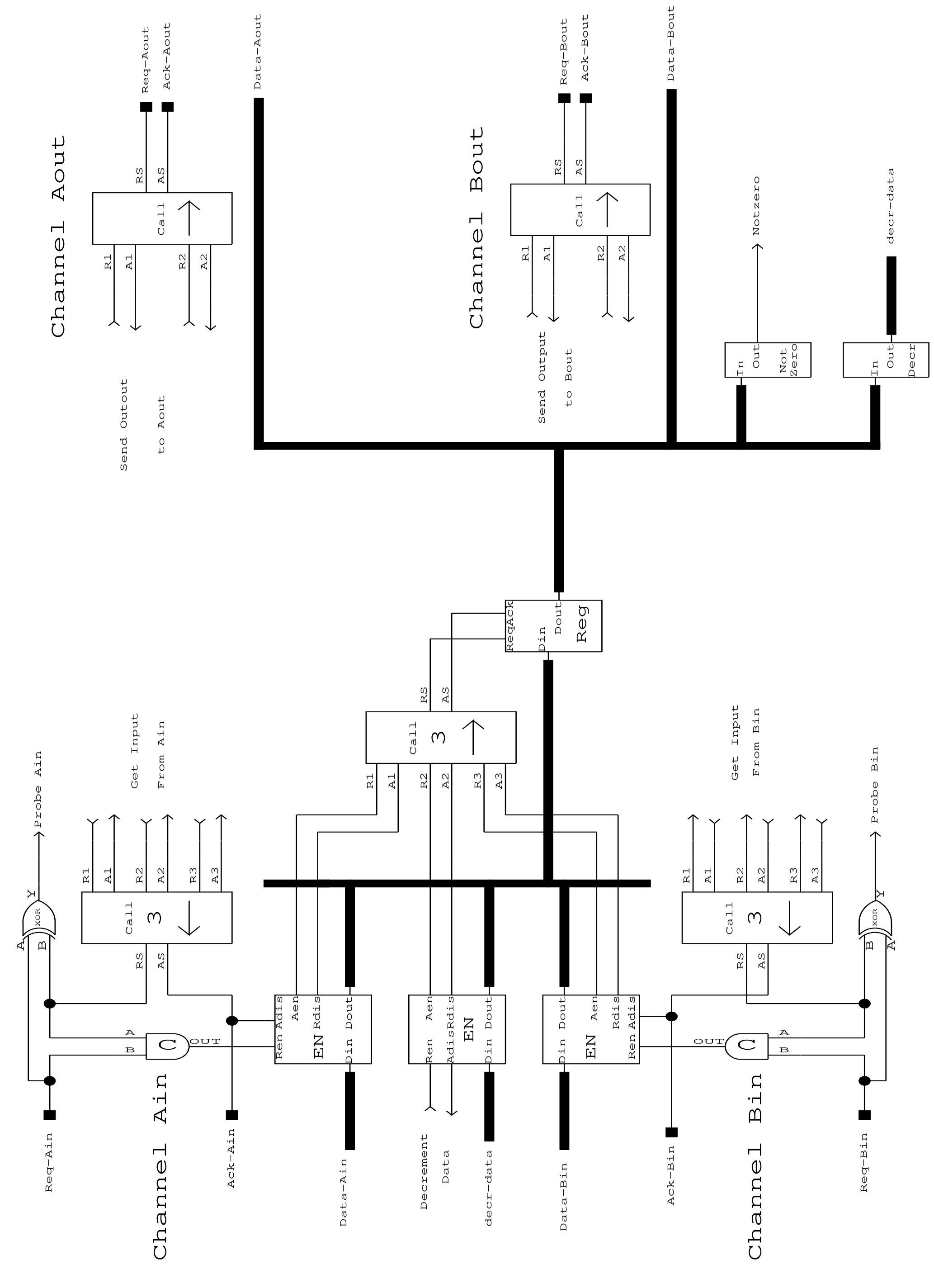

Figure 6.9: Data Path for a Router Macro

Figure 6.10: A Photomicrograph of the Fabricated Mesh-Element

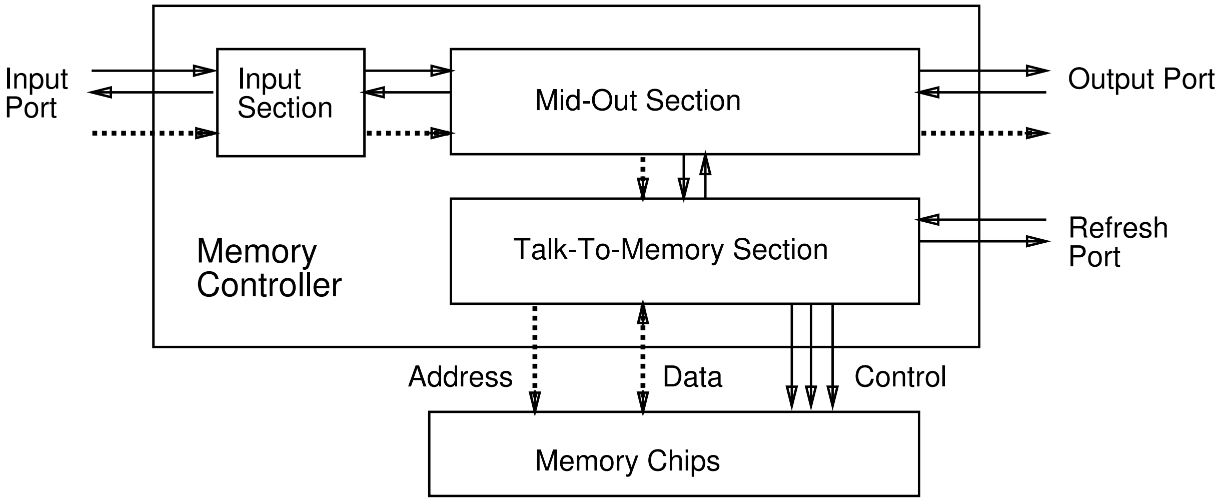

Figure 6.11: Memory Controller Block Diagram

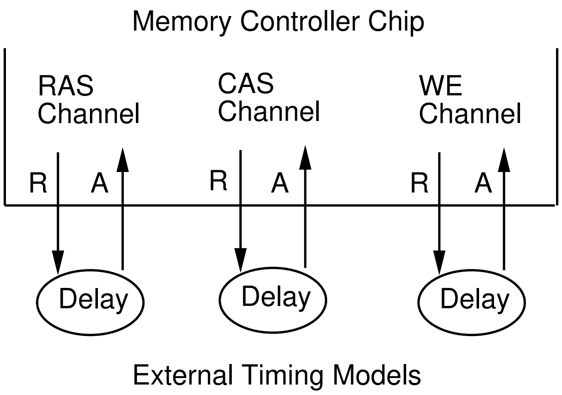

Figure 6.12: DRAM Timing Models

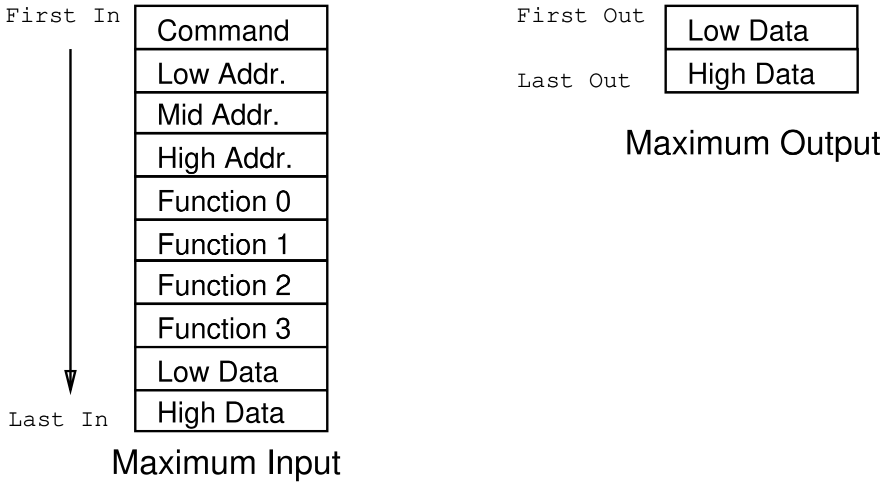

Figure 6.13: Maximum Input and Output Packets

In a single command the memory controller can do either a read cycle, a write cycle or both to implement a read-modify-write cycle. Data can be delivered to the output port on any cycle, even a write cycle, and the data written to the chip can be combined with data read from the chip through a full ALU. The data path to the memory is 16-bits wide, and the current controller uses 18-bit addresses (although three bytes of address are consumed). There is also a control channel used to initiate refresh events to the memory chips. No data are passed across this channel.

Command Packets

Commands to the memory controller are sent as a series of bytes to the input port. The first byte is the command byte which tells, among other things, how many more bytes the chip can expect in this packet. The maximum possible input and output packets are shown in Figure 6.13. From this figure you can see that after the command byte come 0, 1, 2, or 3 address bytes, 0, 1, 2, or 4 function bytes, and possibly two data bytes. The output is either nothing, or two data bytes depending on which function is executed.

The content of the input packet is heralded by a command byte, the layout of which is shown in Figure 6.14. The low-order two bits determine how many address bytes will follow. The bits are encoded as follows:

0 → No address bytes. Previous address is incremented by one and used as the new address.
1 → One address byte will follow
2 → Two address bytes will follow
3 → Three address bytes will follow

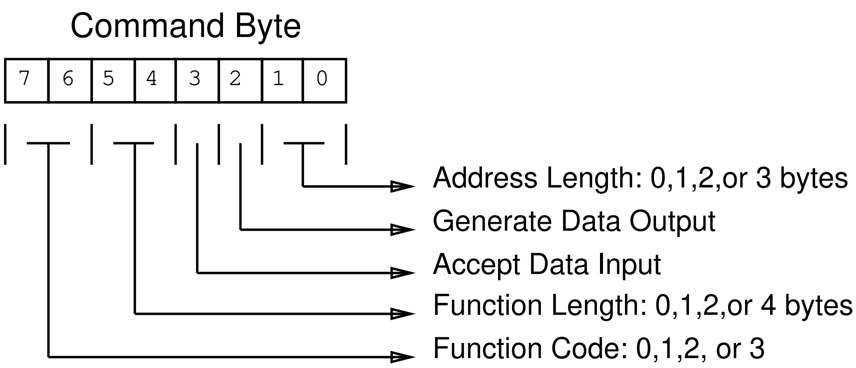

Figure 6.14: Command Byte Format

Bit 2 determines whether data will be delivered to the output port. Note that because this is specified in the command byte it is independent of the function being executed. Even a write cycle may generate output. The next bit warns the chip whether to expect input or not. If so, two data bytes will be expected in the input packet.

The function length field tells how many new function bytes will follow in the packet. This is how the function registers on the chip are loaded. There is a separate field to determine which of the function registers will be used. Note that there is no way to load only three function bytes. The coding is:

0 $\rightarrow$ No new function bytes will follow
1 $\rightarrow$ One function byte will follow. Load it into register 0.
2 $\rightarrow$ Two function bytes will follow. Load the first into register 0 and the next into register 1.
3 $\rightarrow$ Four function bytes follow. Load all the function registers with new information

The function bytes encode parameters for various chip functions. There are four registers in the input section of the memory controller that store these bytes. Each may be set to indicate a different set of operations. For example, function[0] may be set up to do a simple read cycle, function[2] may do a simple write cycle, function[3] might be set up for a read-modify-write cycle where the data from the memory is "anded" with a mask provided by the user, and function[4] could be set up to add the input data to the contents of the memory and write the value back. The contents of these function registers can be changed with each command packet received by the memory controller.

Finally, the function code tells which function register to use for this memory operation. Thus, the maximum input packet is 10 bytes as shown in Figure 6.13, and the minimum input packet is a single byte. Imagine that function register 2 is set up to be a simple read

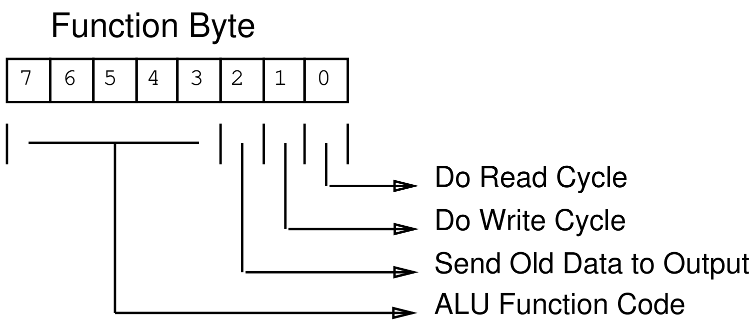

Figure 6.15: Function Byte Format

cycle. In this case, if the user wants to read from a location one away from the last location (remember that if no new address bytes are received the old address is incremented by one) a single command byte can execute that read cycle and generate output. The command byte in this case would be #b10000100.

Function Bytes

Chip function is stored in four function registers in the input section. The format of the function bytes is shown in Figure 6.15. There are separate bits to enable a read cycle and a write cycle. If both bits are asserted the chip does a read cycle followed by a write cycle. For any write cycle, the data to be written are combined with previous data read from the chip through an ALU. Because data always go through the ALU, the user must make sure to set the ALU into the proper mode if he wants the data written without modification. When output is requested in the command byte, the function byte allows a choice of which data are delivered to the output port. Either the data from the input port is sent, or the modified data at the output of the ALU is sent. The remaining bits in the function byte determine the ALU function to be performed.

The ALU provides all 16 logic operations of two variables. The two variables in this case are the input data, and the data that have been read from the memory. The result of the logical operation is sent to an adder. In the table that follows, Ain represents the input data, and Bin represents the data from the memory chip. The function of the ALU is (If addsel Then Ain + F(Ain, Bin) Else F(Ain, Bin))

The ALU function is encoded in bits 3–7 of the function byte. Bit 3 is the adder select bit. If it is high, the add function is selected, if low, the logic unit is selected for output. The other four bits encode the logic function computed by the ALU. These functions are encoded as shown in Figure 6.16.

bits 7654	Logic Function
0000	0
0001	$A \wedge \overline{B}$
0010	$\overline{A \vee B}$
0011	$\overline{B}$
0100	$A \wedge B$
0101	A
0110	$\overline{A \oplus B}$
0111	$A \vee \overline{B}$
1000	$\overline{A} \wedge B$
1001	$A \oplus B$
1010	$\overline{A}$
1011	$\overline{A \wedge B}$
1100	B
1101	$A \vee B$
1110	$\overline{A} \vee B$
1111	1

Figure 6.16: ALU Function Codes

Operation

The sequence of signals observed at the pins of the memory controller chip depend on the function being executed. At the input and output ports, the bundled data interfaces behave just like standard two-phase bundled data paths. At the input side, the sender sets the value of the data, waits for the bundling delay, and then causes an event on the *Req* line. When an event is seen on the *Ack* line the sender can update the data to new values and cause another event on the *Req* line. Likewise, at the output port, when an event occurs on the output *Req* line the data are stable. When the receiver has finished with the data it causes an event on the output *Ack* line and waits for another event on the *Req* line. These input and output ports correspond to the input and output channels described in the OCCAM code for the memory controller.

Because the memory chips do not use a bundled two-phase interface, the interface to the memory chips is a little different. The control signals to the DRAMs are *RAS*, *CAS*, and *WE.* These are implemented as synchronization channels with no data being passed. The memory controller code generates a signal on these control wires by synchronizing on these channels. This causes a transition on the output *Req* wire of the channel and changes its value. A second synchronization on that channel will change that wire's value back. A pulse on a control wire to the memory chips is produced by a series of two synchronizations on the corresponding channel.

These synchronization channels are also used to model the delays required by the DRAMs. A delay external to the memory controller chip is connected between the *Req* and *Ack* of each control channel to model the necessary delay in these signals. For example, the user of the memory controller is responsible for adding a delay between the *RAS-req* and *RAS-ack* to model the time that the DRAM requires *RAS* to be asserted. These delay models may be simple RC delays, or more complicated models if, for example, the DRAMs have different requirements for high and low durations of control signals.

Chip operation consists of sending sequences of bytes to the input channel and receiving data from the output channel. Periodically, requests are made at the refresh channel to initiate part of a refresh cycle. Because there is an arbiter included in the circuitry of the memory controller, a refresh cycle will begin sometime soon after the request is issued. A memory operation in progress will not be interrupted, but the refresh will occur before the next memory cycle can begin. Because refreshes must be made at periodic intervals, a low frequency square wave input to the refresh request should be adequate to make sure the chip is refreshed.

Refresh cycles use a technique known as "RAS-only refresh" where a row address is presented to the DRAM and the *RAS* control line is cycled. This is a simple way of refreshing DRAMs but one that requires that the memory contoller maintain a separate refresh address counter. More modern DRAMs may have more efficient refreshing schemes, possibly including their own refresh counters as part of their design, but *RAS-only* refresh was used to maintain compatibility with a previous version of this chip.

The function bytes determine the sequence of events at the interface to the memory chips. In general, addresses are presented along with the control signals *RAS*, *CAS*, and *WE*. Data are read and written on a single I/O data bus to the memory chips. The precise sequence of these events can be determined from the source code listing that describes the memory controller.

Source Code for the Memory Controller

Definitions The OCCAM code for the memory controller is shown in Figures 6.17 to 6.22. The first piece, shown in Figure 6.17 describes the alias definitions and new functions used in the memory controller. Each field of the command and function bytes are given aliases for later use in the code. Note that because these definitions are simply macro substitutions they must be used inside the scope of a variable named "command" or "function" respectively.

Three new functions are defined for use in the memory controller. The two functions that increment a variable are built using a carry completion adder like that described in Section 4.5.6 modified to always add one to the other input. Because these are defined as functions, there will be a register, and thus a *Funcvar*, associated with each one.

The third function is an ALU. The ALU is also defined as a function and will also

```
;;
;; Definitions for the memory controller
;;

;; Identify the bits in a byte named “command”
(Define   “Asel0”        ’(subseq command 0 0))     ;number of address bytes
(Define   “Asel1”        ’(subseq command 1 1))     ;0, 1, 2, or 3 bytes
(Define   “gendata”      ’(subseq command 2 2))     ;generate output data
(Define   “getdata”      ’(subseq command 3 3))     ;get input data
(Define   “funlen0”      ’(subseq command 4 4))     ;function length
(Define   “funlen1”      ’(subseq command 5 5))     ;0, 1, 2 or 4 bytes
(Define   “funsel0”      ’(subseq command 6 6))     ;select a function
(Define   “funsel1”      ’(subseq command 7 7))     ;for this memory cycle

;; Identify the bits for a byte named “function”
(Define   “funcread”     ’(subseq function 0 0))    ;read cycle flag
(Define   “funcwrite”    ’(subseq function 1 1))    ;write cycle flag
(Define   “olddataflag”  ’(subseq function 2 2))    ;send old data flag
(Define   “funcbits”     ’(subseq function 3 7))    ;alu function bits

;; Define functions to increment different sized quantities,
;; and a 16 bit alu function.
(Add-function “incr”     ’(24 24))                  ;increment main address
(Add-function “refincr”  ’(9 9))                    ;increment refresh address
(Add-function “alu”      ’(16 16 5 16))             ;ALU: Ain Bin Function and output
```

Figure 6.17: Definitions for Memory Controller Source Code

include a *Funcvar*. The ALU is implemented like the one described in Section 4.6.2 and shown in Figure 4.32.

Input Section The input section, shown in Figures 6.18 and 6.19, receives packets from the input channel a byte at a time and operates on those bytes accordingly. The first byte in each packet is the command byte for that packet and will determine how many more bytes the input section will expect from this packet. Auxiliary processes are called to read the address, function, and data bytes as required. Once an entire packet has been assembled, the appropriate information is sent in parallel to the middle section and a new packet is awaited.

There is pipelining between each of the sections of the memory controller. While the other sections are performing a memory cycle, the input section can be assembling a new packet from the input port.

```
;;
;; The input process. Get data from the input and pass it along to the middle.
;;
(Process input-section ((Chan input mid0 mid1 mid2 mid3))
  (Block ((Var command⟨8⟩ address⟨24⟩ data⟨16⟩           ;command, address and data words
          func0⟨8⟩ func1⟨8⟩ func2⟨8⟩ func3⟨8⟩)            ;four function bytes
          (Funcvar (addr⟨24⟩ incr)))                       ;the var in the incr function
  (While True                                              ;do forever
    (? input command)                                      ;get command byte
    (get-address input address)                            ;get other bytes as needed
    (get-functions input func0 func1 func2 func3)
    (get-data input data)
    (Par                                                   ;when you have the data, send it all out
      (! mid0 gendata)                                     ;just send the gendata bit
      (! mid1 address)                                     ;send the address (24 bits)
      (! mid3 data)                                        ;send the data (16 bits)
      ;; Decode which function to send out on the mid2 channel
      (If ((and funsel1 (not funsel0)) (! mid2 func2))     ;10 → func 2
        ((and funsel1 funsel0) (! mid2 func3))             ;11 → func 3
        ((and (not funsel1) (not funsel0)) (! mid2 func0)) ;00 → func 0
        ((and (not funsel1) funsel0) (! mid2 func1)))))))  ;01 → func 1
```

Figure 6.18: Memory Controller Input Section

The auxiliary processes called by the input section use the information in the control byte to gather the other pieces of the input packet. Note that if no new address bytes are expected the old address is incremented.

Middle Section The middle section of the memory controller is essentially just a pipeline stage between the input and output sections, and the memory control stage. The input packet is accepted by the middle section from the input section and then sent to the memory control section for processing. If the particular function requires output from the memory section, the middle section waits for that output to be produced from the memory and then sends it to the output port before accepting a new input packet.

Memory Control Section The memory control section, shown in Figure 6.21, is where the control signals to the actual DRAMs are generated. The main body of the memory control section is an *Alt* construct that chooses between new memory requests and refresh requests. Because the *Alt* is enclosed in a non-terminating loop the choice of alternative will be fair in the sense that a pending refresh request will be serviced after at most one memory cycle.

If a refresh request is chosen for execution, the first step is to present the *RAS* address on the address lines, and then assert *RAS-req*. After some *RAS* delay, this request is acknowledged and *RAS* may be set low with another communication on the *RAS* channel and the refresh address incremented.

If the chosen alternative is a memory cycle, then the rest of the information from the middle section must be assembled. Now the *RAS* address is presented to the address lines and communication requested on the *RAS* channel. When that communication, and the corresponding *RAS* delay, have completed the *CAS* address may be presented to the memory. Now, if the function byte indicates a read, *CAS* is asserted and the result is read from the data lines, after which *CAS* may be lowered. The data read from the memory is now operated on by the ALU according to the function requested.

If output is requested of this function, it is done in parallel with the possible write cycle of the chip. The output data are sent to the middle section for delivery to the output channel.

If a write function is requested, the direction of the data I/O path is changed to output and the data to be written, along with a *WE* signal, are presented to the memory chips. *CAS* is asserted to complete the write operation. After the *CAS* delay, the *CAS* and *WE* signals are lowered, and the direction of the data I/O path is changed back to input.

Finally, when all possible parts of the memory cycle have completed, *RAS* is lowered to finish the cycle.

Top Level Memory Controller Code Once each of the sections has been defined, they are combined to form the complete memory controller. First the middle and memory

```
;;
;; Either get 1,2, or 3 new address bytes or increment the current address by 1.
;;
(Process get-address ((Chan input)(Var address))
   (If ((or Asel0 Asel1)                                    ;get at least one byte
         (? input (subseq address 0 7))
         (If (Asel1                                         ;get at least two bytes
              (? input (subseq address 8 15))
              (If (Asel0                                    ;get all three bytes
                   (? input (subseq address 16 23)))))))
      (True                                                 ;get no new bytes
         (Setfunc addr (incr address))                      ;increment old address
         (Set address addr))))                              ;and set the address to that value

;;
;; Get 0,1,2, or 4 function bytes.
;;
(Process get-functions ((Chan input) (Var func0 func1 func2 func3))
   (If ((or funlen0 funlen1)                                ;at least one byte
         (? input func0)
         (If (funlen1                                       ;at least two bytes
              (? input func1)
              (If (funlen0                                  ;all four bytes
                   (? input func2)
                   (? input func3))))))))

;;
;; If the input-data bit is set get 16 bits of data from the input port.
;;
(Process get-data ((Chan input)(Var data))
   (If (getdata   (? input (subseq data 0 7))               ;get data a byte at a time
                  (? input (subseq data 8 15)))))
```

Figure 6.19: Memory Controller Input Section Auxiliary Code

```
;;
;; The middle stage - Get all the data from the input stage and
;; assemble it for sending to the talk-to-mem process. Get data
;; back from the memory process and send it to the chip output.
;;
(Process mid-out-section ((Chan mid0 mid1 mid2 mid3
                               output funcchan addrchan
                               ctlchan dataoutchan datainchan ))
  (Block ((Var outputflag                          ;flag to generate output
         function⟨8⟩ address⟨24⟩                   ;function and address words
         datain⟨16⟩ memdata⟨16⟩))                  ;data from input and from memory
    (While True                                    ;repeat forever
      (Par                                         ;get all the info
        (? mid0 outputflag)                        ;from the input stage
        (? mid1 address)
        (? mid2 function)
        (? mid3 datain))
      ;; Then send the data to the process that talks to memory.
      (Par                                         ;send the data in parallel
        (! funcchan function)                      ;send function byte
        (! addrchan address)                       ;send addresss
        (! dataoutchan datain)                     ;send data
        (! ctlchan outputflag))                    ;the generate-data bit
      (If (outputflag                              ;now, if you are expecting data
          (? datainchan memdata)                   ;get data from memory process
          (! output (subseq memdata 0 7))          ;send it out a byte
          (! output (subseq memdata 8 15)))))))    ;at a time
```

Figure 6.20: Memory Controller Middle Section Code

```
;;
;; This is the process that talks to the memory and chooses between
;; a memory cycle and a refresh cycle using an ALT statement
;;
(Process talk-to-mem ((Chan funcchan addrchan ctlchan datain dataout refresh))
  (Block ((Var address⟨24⟩                              ;whole memory address
               readdata⟨16⟩ writedata⟨16⟩               ;data to and from memory
               function⟨8⟩ outputflag)                  ;function byte and output flag
      (Funcvar (aludata⟨16⟩ alu)                        ;data from alu
               (refaddr⟨9⟩ refincr))                    ;refresh address
      (Chan ras⟨0⟩ cas⟨0⟩ we⟨0⟩ memchangedir⟨0⟩         ;memory and data path control
               tomemory⟨16⟩ frommemory⟨16⟩              ;channels to and from memory
               memaddress⟨9⟩))                          ;addresss to memory
    (While True                                         ;repeat forever
      (Lex-alt                                          ;check in lexical order
      ((True (? refresh))                               ;refresh request
        (! memaddress refaddr)                          ;send addresss to memory
        (! ras)                                         ;turn on ras, wait for delay
        (Par (! ras)                                    ;turn RAS off
             (Setfunc refaddr (refincr refaddr))))      ;increment refaddr
      ((True (? funcchan function))                     ;memory operation request
        (Par (? addrchan address)                       ;get memory address
             (? datain writedata)                       ;get data to write
             (? ctlchan outputflag))                    ;get output flag
        (! memaddress (subseq address 0 8))             ;RAS cycle with part of address
        (! ras)                                         ;turn on ras, wait for delay
        (! memaddress (subseq address 9 17))            ;switch address to CAS part
        (If (funcread                                   ;a read cycle is required
             (! cas )                                   ;turn on CAS, wait for delay
             (? frommemory readdata)                    ;latch the data from memory
             (! cas)))                                  ;turn off cas, wait for delay
        (Setfunc aludata (alu writedata readdata funcbits))   ;compute ALU output
        (Par                                            ;send output and finish cycle in parallel
          (If (outputflag                               ;flag to send output data
             (If (olddataflag (! dataout readdata))     ;choose which data to send
                (True (! dataout aludata)))))
          (Seq (If (funcwrite                           ;write data from alu to mem
                    (! memchangedir)                    ;change dir to output
                    (! we)                              ;activate write enable
                    (! tomemory aludata)                ;set data to write to memory
                    (! cas )                            ;turn cas on
                    (Par (! we)                         ;reset the signals in parallel
                         (! cas)
                         (! memchangedir))))            ;change dir back to input
               (! ras))))))))                           ;turn off ras to finish cycle
```

Figure 6.21: Memory Controller Memory Section Code

```
;;
;; now you can combine the middle and talk-to-mem sections into
;; one to hide the internal communications.
;;
(Process mid-mem-section ((Chan mid0 mid1 mid2 mid3 out refresh))
  (Block ((Chan funcchan⟨8⟩ addrchan⟨24⟩ ctlchan              ;Define the internal channels
              datafrominput⟨16⟩ datafrommem⟨16⟩))              ;between the two processes
    (Par                                                        ;Run the processes in parallel
      (mid-out-section mid0 mid1 mid2 mid3 out
                       funcchan addrchan ctlchan
                       datafrominput datafrommem)
      (talk-to-mem funcchan addrchan ctlchan
                       datafrominput datafrommem refresh))))
;;
;; The memory controller is generated by running the input and
;; mid-out processes in parallel.
;;
(Process memory-controller ((Chan in out refresh))
  (Block ((Chan mid0 mid1⟨24⟩ mid2⟨8⟩ mid3⟨16⟩))                ;Define channels from input to middle
    (Par                                                        ;Run the processes in parallel
      (input-section in mid0 mid1 mid2 mid3)
      (mid-mem-section mid0 mid1 mid2 mid3 out refresh)))
;;
;; Finally, make a call to the memory-controller process to generate the circuit
;;
(Block ((Chan in⟨8⟩ out⟨8⟩ refresh⟨0⟩))
  (memory-controller in out refresh))
```

Figure 6.22: Memory Controller Top-Level Code

sections are combined, and this composite is then combined with the input section. These operations hide all of the channels between the various sections as internal channels. The final *memory-controller* macro includes only the input, output, and refresh channels. Because the connections to the memory are not standard bundled data channels, they must be treated separately.

Implementation

The code for the memory controller was compiled by the prototype compiler. The initial circuit included 236 circuit modules, too many to include a schematic in this dissertation. After optimization this number is reduced to 114; still too large to illustrate here. The optimized version sent for fabrication included ≈22,000 transistors using the current cell library.

The netlist result from the compiler was simulated completely using the RNL simulator [97] before fabrication. The result from the fusion router is 5254 by 6056 microns and fits into the MOSIS 64PC69x68 64-pin standard pad frame. A photomicrograph of the memory controller chip is reproduced in Figure 6.23. Initial testing on the returned chips indicates that they are fully functional but additional tests are needed before claims about timing can be made.

Figure 6.23: Photomicrograph of the Memory Controller Chip