27

Implementation of the Intel 486 SX Microprocessor in Verilog Hardware Description Language

by

Adam Y. Chen

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in Electrical Engineering at the Massachusetts Institute of Technology

May 1993

ARCHIVES
MASSACHUSETTS INSTITUTE
JUL 26 1993

Copyright Adam Y. Chen 1993. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute copies of this thesis document in whole or in part, and to grant others the right to do so.

Author____________________________________
Department of Electrical Engineering and Computer Science
May 17, 1993

Certified by____________________________ May 13, 1993
Gregory M. Papadopoulos
Thesis Supervisor

Accepted by____________________________________
Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

Implementation of the Intel 486 SX
Microprocessor in Verilog Hardware Description Language
by
Adam Y. Chen

Submitted to the
Department of Electrical Engineering and Computer Science

May 17, 1993

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Electrical Engineering

ABSTRACT

This thesis simulates most of the characteristics of the Intel 486 SX Microprocessor operating in Real Mode. Real Mode operation is the default condition of the 486, where it simply acts as a very fast 8086 Processor. The implementation contains the Cache Unit, the Bus Interface Unit, the Segmentation Unit, the Prefetch Unit, the Instruction Decode Unit, and the Integer Unit all working in conjunction. It runs actual 'x86 assembly code, responds to external chip control lines, and replicates the detailed physical structure of the 486 itself. Interrupts, I/O, and Power-up Debugging features of the 486 are not supported. Tools are provided to extensively test and verify the operation of this simulation. All code is written in Verilog Hardware Description Language at a structural level. Verilog is used in industry to simulate complex circuits, as well as generating gate level designs of its descriptions using logic synthesis tools.

Thesis Supervisor: Gregory M. Papadopoulos
Title: Professor, Electrical Engineering and Computer Science

Table of Contents

I. Introduction 1

II. The Microprocessor
1. Overview 3
2. External Interfaces 4
3. The Datapath 9
4. The Control Path 15
5. Instruction Execution 17
6. Differences with the Intel 486 SX 20

III. The Verilog Model
1. 'x86 Assembly Code 22
2. Model Functionality 26
3. Possible Changes and Modifications35

IV. Operation Verification
1. The System Tests 39
2. Block-Level Testing 40

V. Conclusions 42

VI. Appendices
1. Simulation Code 44
2. Datapath and ALU Controls 126
3. Handshaking Signals 128
4. Verification Programs 130
5. References 165

List of Figures

1. A General Microprocessor	2
2. The Microprocessor	4
3. RAM Flowchart	6
4. Parity Check Flowchart	7
5. Bus Interface Unit Flowchart	8
6. External Interfaces	9
7. The Datapath	10
8. Cache Structure	12
9. The Cache Algorithm	13
10. Segmentation Algorithm	14
11. The Control Path	16
12. Prefetch Flowchart	17
13. Decode Flowchart	18
14. Flag Register	19
15. Instruction Execution Process	20

List of Tables

1. External Control Signals to the Microprocessor	5
2. The MODEM Byte	23
3. Supported Instructions and Their Binary Encodings	24
4. Module i486	28
5. Module biu	28
6. Module cache	29
7. Module pchk	30
8. Module tristate	30
9. Module shifter	30
10. Module segmentation	30
11. Module paging	31
12. Module address	31
13. Module flag	31
14. Module alu	32
15. Module prefetch	32
16. Decode Subroutines	33
17. Module decode	34
18. System Test Errors	40
19. Block-Level Tests	41

I. Introduction

This thesis simulates most of the characteristics of the Intel 486 SX Microprocessor operating in Real Mode. That is to say, this project is a software package, written in Verilog Hardware Description Language, that acts like a microprocessor. It runs assembly code, responds to external control signals, and replicates the internal structure and operation of the actual processor.

The 486 SX is a chip in the 80x86 family of microprocessors from Intel that is widely used in top of the line, IBM compatible personal computers today. It features backward compatibility with the rest of the 80x86 line, an 8K cache, on-chip memory management, and a RISC integer core. The processor is broken down into eight distinct functional units working in parallel. In Real Mode, the 486's default operating mode, it's base architecture acts like an extremely fast Intel 8086 Microprocessor. The SX, unlike the 486 DX, does not contain a floating point unit [1]. While effort has been made to make this model adhere as closely as possible to the actual processor, several functions, such as interrupts, I/O, and Power-up Debugging features of the 486 are not supported.

In principle, the operation of the a microprocessor is simple. It first takes an instruction from memory. This is called the "fetch" operation. It then looks at this instruction and figures out what it should do in response. This is the "decode" phase. The requested function is then "executed", usually through an Arithmetic Logic Unit made from gate combinations. "Writeback" is the final stage where the results of the execution stage is written to memory. The cycle then repeats again with the next instruction fetched from memory (see figure 1). The power of modern microprocessors comes from their ability to repeat these simple steps over and over again, quickly, cheaply, and reliably [2].

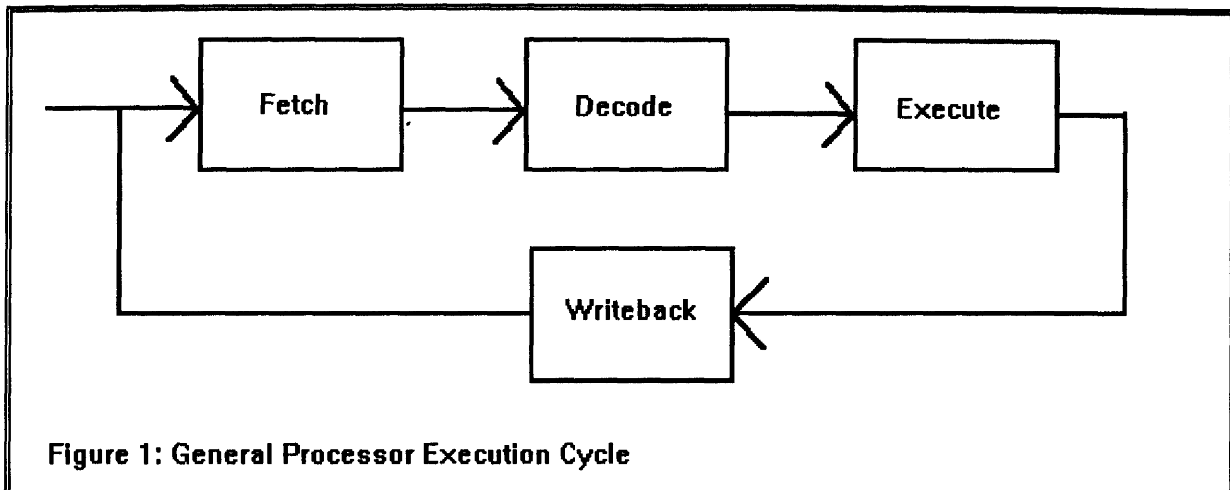


Figure 1: General Processor Execution Cycle

This project is written in Verilog HDL at a structural level. Verilog is widely used in industry to simulate complex circuits. As electronic designs become larger and more involved, gate level schematics become unmanageable and incomprehensible. Verilog helps to abstract away that mess, leaving the behavior of the circuit unobstructed. Yet Verilog is more than just a description tool, it acts also as a design tool. A well written, structurally based Verilog program can be compiled to a gate level implementation through the use of logic synthesis tools. So a high level Verilog program is equivalent to a circuit diagram, without the mess, and retaining the functionality. In addition, that same Verilog description can be simulated to verify the operation of the design [3]. This thesis presents exactly such a simulation.

The purpose behind creating this model is to verify the operation of a Verilog simulator being ported to a massively parallel architecture. Since Verilog is designed for hardware simulations, many of its operations occur in parallel. It is hoped that a version of Verilog running on a parallel architecture would be able to execute simulations at much higher speeds. This 486 model will serve as a complicated benchmark to thoroughly test out this parallel version of Verilog. To this end, a set of testing and verification tools are also included in this project.

II. The Microprocessor

1. Overview

This microprocessor model can be separated into eight distinct units in three functional blocks[4]. While these units do not correspond exactly to the units in the 486, they do exhibit glaring similarities. The external interface consists of two of these units, the parity check unit and the bus interface unit. The parity check unit does not interface with the internals of the processor, but rather just checks to see if a read or write to memory is occurring and responds appropriately. The bus interface unit acts as the go between for the processor and the outside world.

The datapath of the processor is the trail on which the data being processed travels. This includes the cache unit, the memory segmentation unit, and the integer unit. The integer unit is where the arithmetic unit, the barrel shifter, and the processor registers exist. The datapath also describes the connections of the wires and the tristate flow controls between these separate blocks. This is where the execution and writeback phases in a processor occur.

The fetch and decode phases of a processor take place in the control path. The control path encompasses the prefetch, paging, and decode units. The prefetch unit, with help from the paging unit, gets instructions from memory and pipes them to the decode unit for processing. The only connection between the data and control paths occur here. The decode unit will issue instructions that directly affect the tristate flow controls, as well as pushing immediate values contained in the assembly code to the datapath.

In order for the microprocessor to function, two critical functional blocks external to the processor are necessary. The first is the clock. Almost every operation in the microprocessor is pegged to the clock. Without it, no functions can occur. The other block is memory. Memory must exist to hold the instructions for the microprocessor as well as provide a place for the results of the computations to be returned to. In addition to external memory and the clock, external control signals to the microprocessor must also be set to

define the proper state of the chip. These signals define operations such as reset, cache invalidation, bus size, etc. (see figure 2).

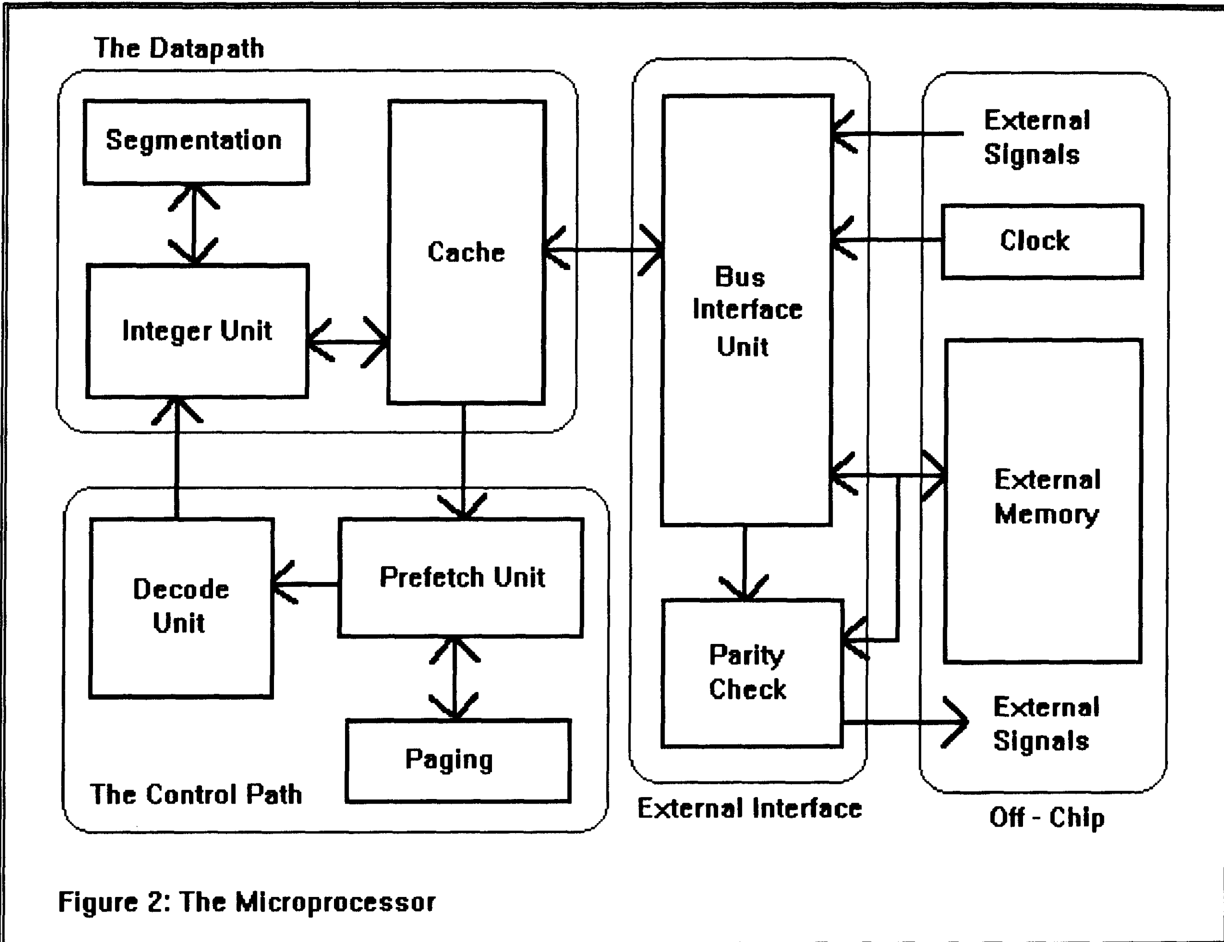


Figure 2: The Microprocessor

2. External Interfaces

The only functional units of the microprocessor that have contact with the external world are the bus interface unit and the parity checking unit. In this simulation, the external world is defined as everything off-chip. This would be the clock, the external memory, and the external signals that control the processor. The model supports cache control signals, bus arbitration signals, bus size control signals, bus cycle definition signals, the reset signal, and the burst, read/ write control signals (see table 1).

Table 1: External control signals to the microprocessor

BENABLE	Controls external memory size (only 32-bit mode is supported)
BCDEF	A read from memory is initialed at BCDEF = 25, 26, or 27, Write at BCDEF = 30 or 31
HOLD	Forces processor to release control of the external bus
HLDA	Acknowledges the receipt of the Bus HOLD signal
BOFF	Forces processor to release control of the external bus (similar to HOLD)
ADS	Request for memory operation (see appendix 3 for handshaking details)
BRDY	Data ready signal from memory (see appendix 3 for handshaking details)
BLAST	Indicates last byte during a burst read or write (see appendix 3 for handshaking details)
PLOCK	Indicates that the processor is using the external bus
BREQ	Similar function to ADS
BSIZE	Defines the size of the external bus (only 32-bit memory is supported in this model)
AHOLD	Stops the processor from driving the bus and waits for input onto the external bus
EADS	Performs a cache invalidation on the address specified by the external bus
RESET	Performs a soft reset and reinitializes variables used in the simulation

The simulation uses a simple clock, one that oscillates between 1 and 0 at each time unit in Verilog. The other off-chip functional unit is the RAM memory. It acts as 32-bit memory, which means it is unable to operate on single bytes, but only on double words (32-bits, 4 bytes). It also contains circuitry to decode the BCDEF signal from the bus interface unit (BIU) to determine if a read or write has been requested. The RAM works independently of the clock, simply waiting around for an activation signal from the BIU (see figure 3). Assembly code for the microprocessor must be entered into memory through the Verilog code starting at "data[0]" (refer to appendix 1, ram.v).

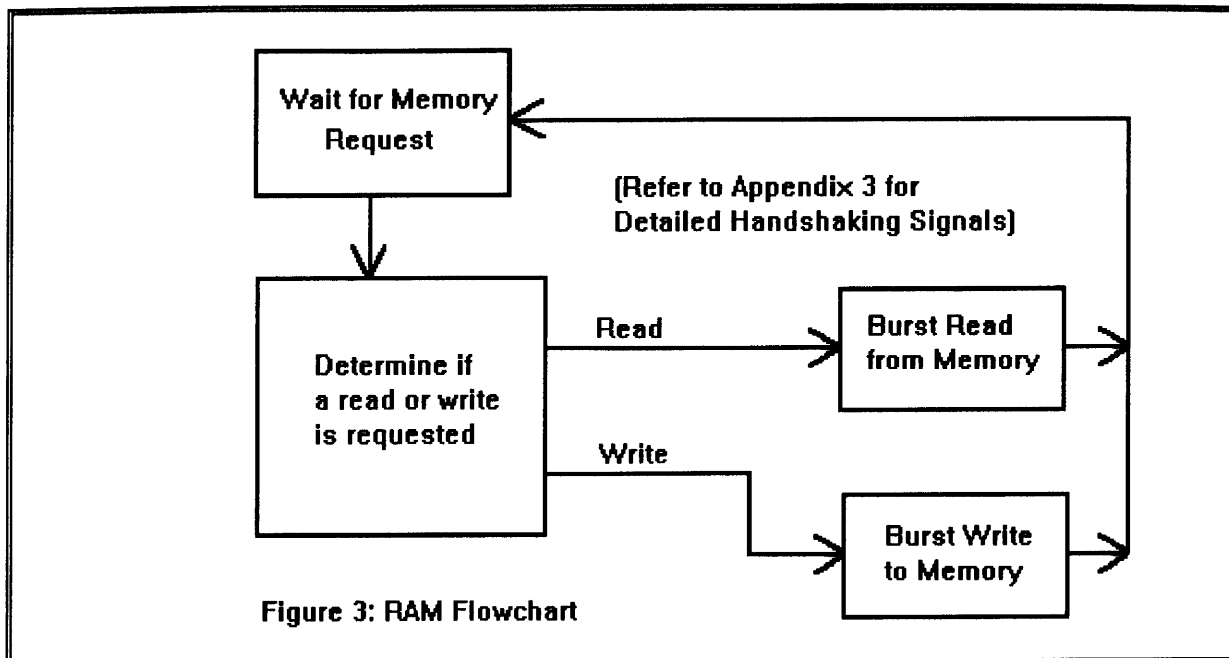


Figure 3: RAM Flowchart

The on-chip interface with the external world consist of the parity and bus interface units. On a memory read, the parity unit will examine the four inputs of the data parity pins. Each pin corresponds to a byte on the external bus. A parity error will be generated if that byte and its corresponding data parity input do not have an even number of high bits. No action is taken by the microprocessor on a parity error other than the parity status pin going low. This status output can thus be completely ignored if parity errors are not of interest. On a memory write, the data parity outputs will insure even parity for each output byte (see figure 4). It is then up to the system to check for any parity errors which may develop. This information is currently ignored by the simulation.

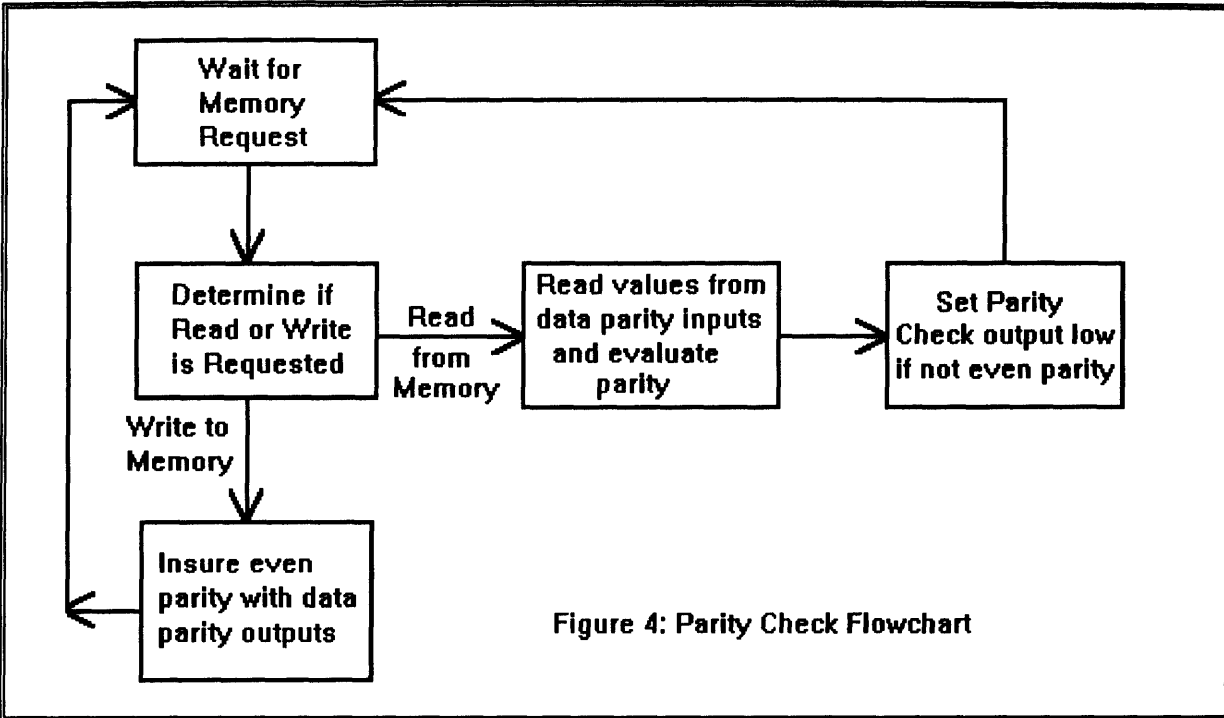


Figure 4: Parity Check Flowchart

The bus interface unit is the essential block that allows the microprocessor to communicate with the outside world. It handles the external control signals as well as all accesses to RAM. Internally, it is controlled directly by the cache unit. Memory requests by any other units must first pass through the cache, to the BIU, then to external memory.

The BIU reads data from memory sixteen bytes (four double words) at a time. This is known as a burst read. Sixteen bytes also happens to be the exact number to fill a cache line (more on this later). For writes to memory, the BIU contains a write buffer capable of holding four double words. In a burst write, anywhere from one to four double words can be written to memory at a time. Single byte operations are not possible since only 32-bit memory is supported by the simulation. Currently, all memory accesses must be properly aligned (bits 0 and 1 of the address have to be zero).

The BIU is responsible for several functions. It can float the external bus, burst read from memory, write to its internal buffers, or undertake "idle writes" to RAM. Idle writes take data from the internal buffer and burst writes them to memory (appendix 3 shows

detailed handshaking signals). They occur when the buffer is full and an additional write is pending, cache misses exist in the buffer and a read is pending, or when the BIU is idle. Generally, memory reads take precedence over memory writes, so reads can take place even if there is still data in the write buffers that have not yet been written to RAM. However, this reordering is undermined if cache misses exist in the buffer or if the buffer contents have already been reordered once before (see figure 5).

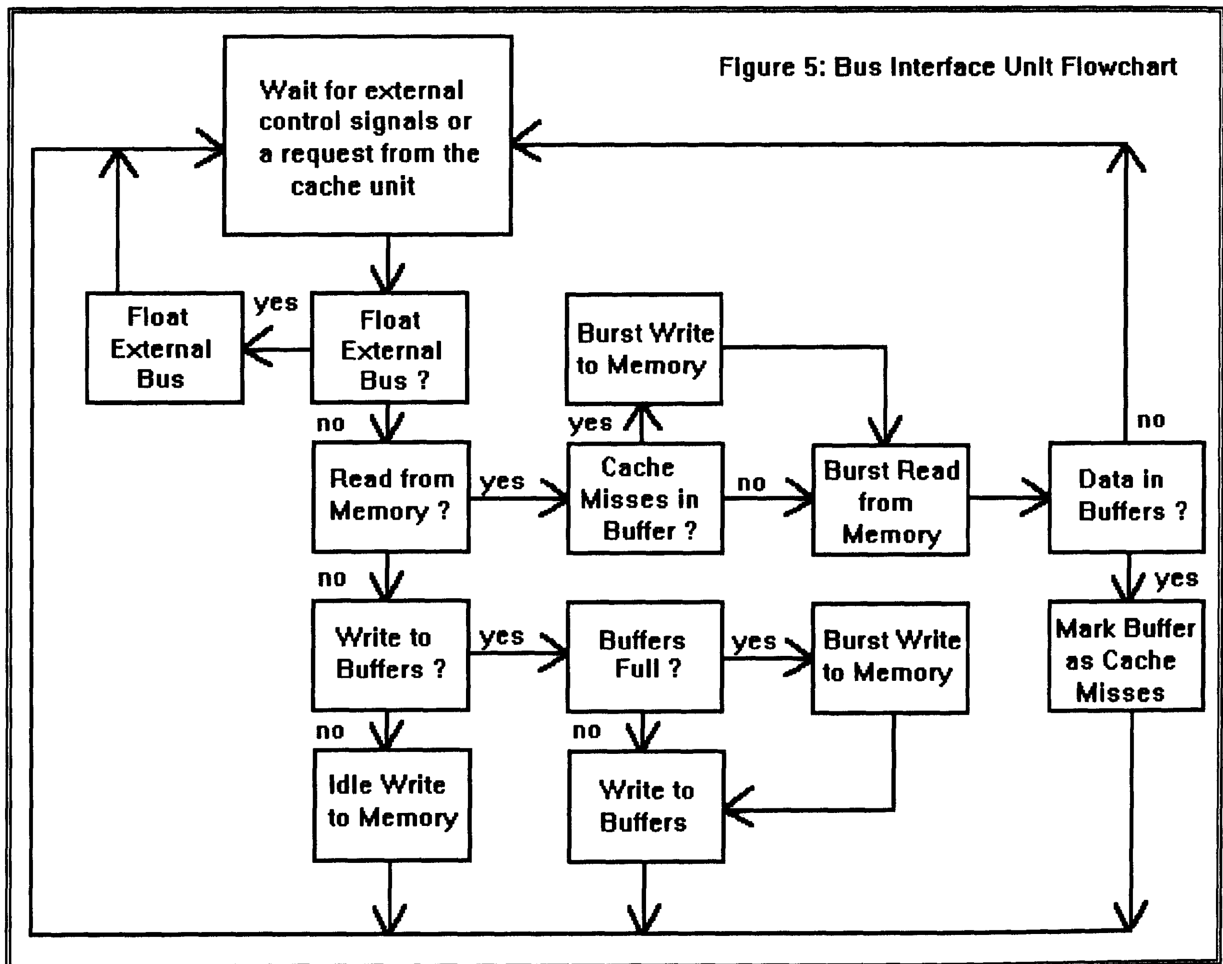


Figure 5: Bus Interface Unit Flowchart

The Bus Interface Unit is the external traffic cop for the microprocessor. It allows data to be read from and written to the RAM. It can prioritize that sequence, as well as, take orders from the system and provide bus information to the system. The external interfaces is the buffer between the internal on-chip memory, the cache, and the external memory, the

RAM. Figure 6 shows the exact inputs and outputs to the external interfaces used in the simulation.

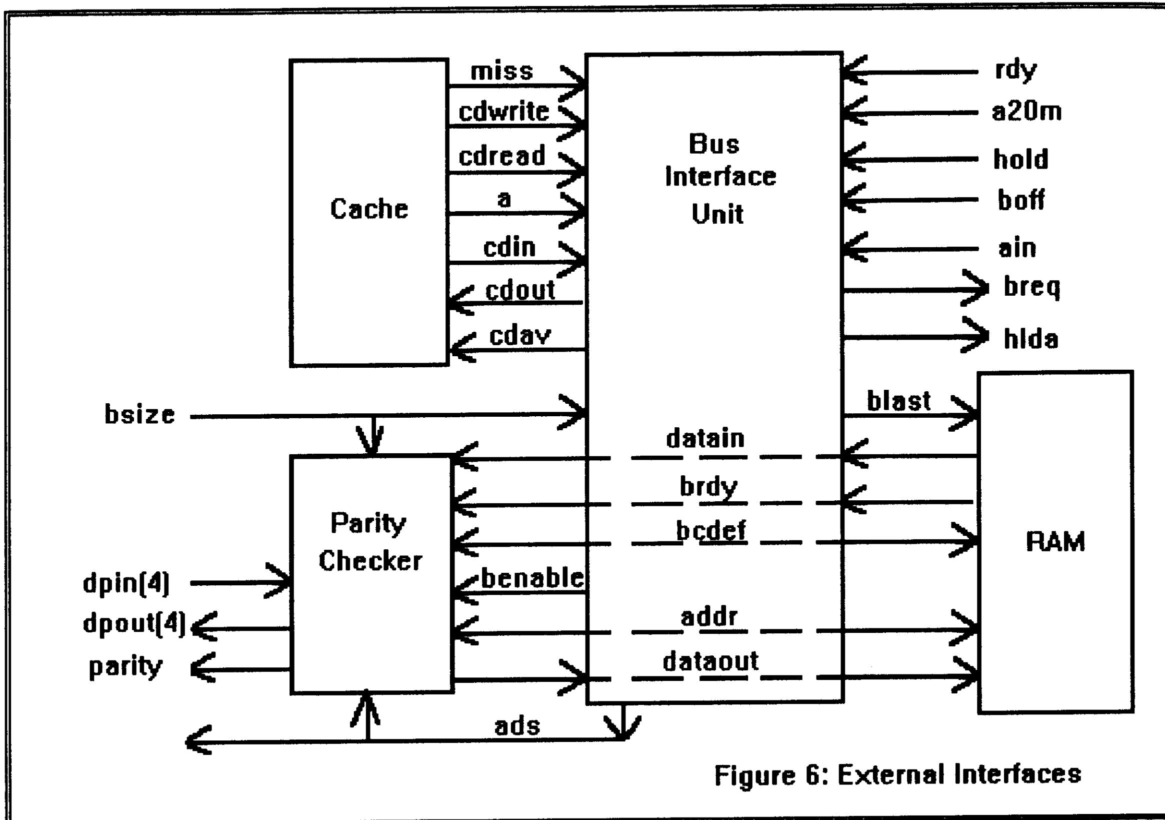


Figure 6: External Interfaces

3. The Datapath

In the datapath lies the "guts" of the microprocessor. It is where the actual instructions are executed. It includes the cache, the segmentation unit, the arithmetic logic unit (ALU), the barrel shifter, and all the processor's registers. These functional units are connected by internal buses, with flow on them dictated by tristate buffers (see figure 7). The tristates, as their name implies, can take on one of three states, high, low, or a high impedance (open circuit) state. They act as gates to either pass data at their inputs, or block data from getting to the buses. The control circuitry in the microprocessor exerts its authority by turning these tristates on and off, in addition to giving explicit instructions to the individual units.

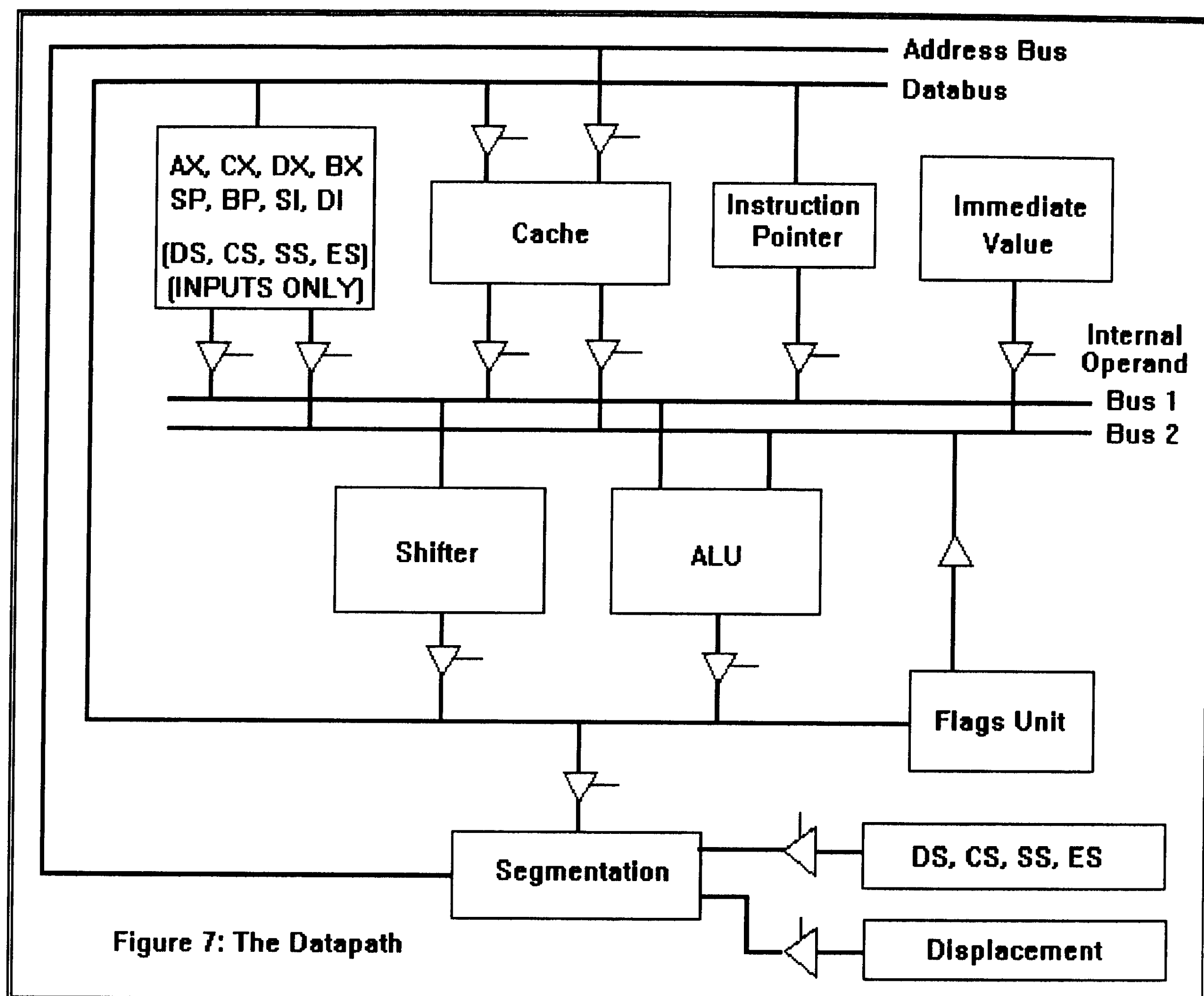


Figure 7: The Datapath

The cache is the fast internal memory of the processor. Generally, a cache contains tags which holds addresses of the data being held in the cache. As a read request comes in, the address of the requested data is compared to the addresses in the tags. If a match occurs, the data is simply read directly from the cache, rather than external memory. Otherwise, the cache will get the data from external memory, place it into cache memory, and then send it to the requesting device. A write request operates similarly, except that cache hits are also written through to external memory. This insures that should the particular cache line be discarded, a subsequent read from external memory will still provide valid data [5].

While newer designs usually contain separate data and instruction caches to further separate the data and control paths, the 486 has one unified 8 kilobyte cache. The existence

of an on-chip cache speeds up instructions, but with only one cache, conflicts may develop when the processor needs data from memory but the cache is already in use by the control path. A semaphore (a device which indicates when a resource is being used) is utilized by the control and data paths to allocate authority over the cache.

The microprocessor uses a 4-way set associative cache, with Least-Recently-Used replacement. An ideal cache would be fully associative. This means that each byte of data would have a unique tag with a unique address corresponding to it. This works really fast and really well. But to determine a cache hit, the number of comparators (to compare the requested address to the tagged addresses) would have to equal the number of memory locations. Hardware limitations force a compromise, resulting in a 4-way set associative cache.

This cache divides a requested address into three parts, the tag field (bits 31 to 11), the index field (bits 10 to 4), and the byte select (bits 3 to 0). Since data is stored into 16 byte cache lines, equivalent to a single BIU burst read, the byte select is simply used to pick the exact byte in the cache line. The cache lines are divided up into 4 "ways", with 128 cache lines (called sets) in each way. So a set contains 4 cache lines, one in each way. The index field determines which of the 128 sets has been requested. The tag field concludes which of the four ways data can be found in, if any (see figure 8).

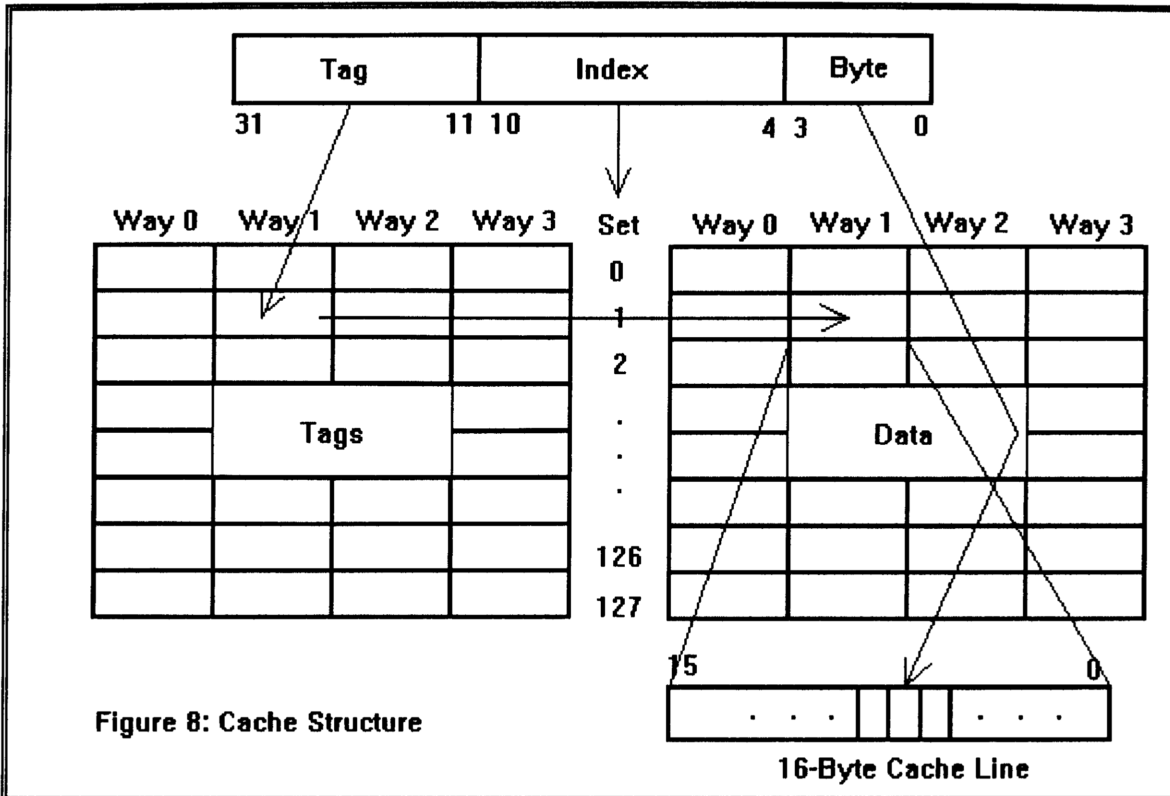


Figure 8: Cache Structure

So only four comparators are needed. The index field decides which set the data will be contained in, if it exists. The cache will then compare the tag field with the tags in that set. There will be four tags per set, one for each way that the data can be stored in. If the four comparisons turn up empty, then a cache miss has occurred. In that case, the request goes out to external memory. The new data will then replace the least recently used cache line in the set, putting it in one of the four ways. In the event of a reset or an external cache line invalidation, the appropriate cache lines will be flagged as invalid and any new cache requests will become cache misses (see figure 9).

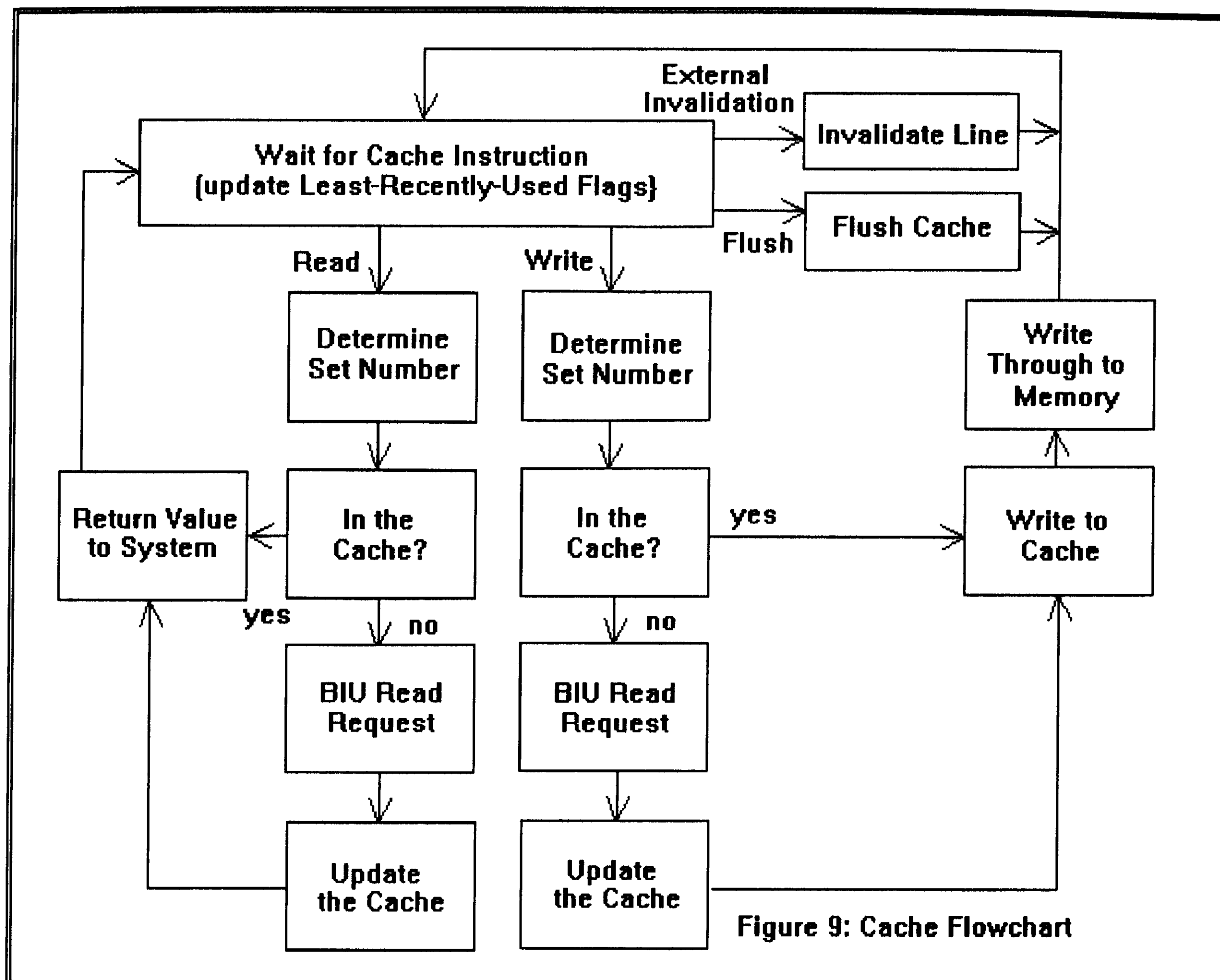


Figure 9: Cache Flowchart

The ALU handles all the integer math and logic required by the system. The actual function of the unit is determined by control signals from the decoded instructions (see appendix 2, under ALU for available functions). While in real life, the ALU would be composed of specialty designed (transistor by transistor) adders, subtractors, and the such, this simulation simply emulates their behavioral characteristics. It is composed of purely combinational logic, which is to say no clocks are involved. As soon as the inputs change, the outputs also change after a finite propagational delay. Getting that delay time down is a major challenge in microprocessor design today, whereas the simulation abstracts all that detail away. This is not cheating however, logic tools are available to easily translate such abstract behavior into a transistor level design, abet probably not quite as effectively.

The barrel shifter occupies the same role as the ALU. It can both shift input values and rotate them. As the ALU, it too is directed by control signals from the decoded instructions. The selection between the ALU and the shifter is controlled by tristates at their respective outputs. Both outputs may not be in use at the same time.

Memory management in real mode is trivial in comparison to the complicated virtual memory requirements of the protected and virtual modes of the 486. Presently, this simulation only supports real mode operation. In real mode, the microprocessor can access 1 MB (20 bit addressing). However, real mode operation uses only 16 bit registers. The addressing problem is solved by using two 16 bit values to represent the full address [6].

The physical addresses of the microprocessor is the 20 bit address used to address the cache. The two 16 bit values are known as the base and offset addresses. To generate the physical address from the segmented address, simply shift the base address left four bits and add the offset address. this would obviously generate the twenty bits we require (see figure 10).

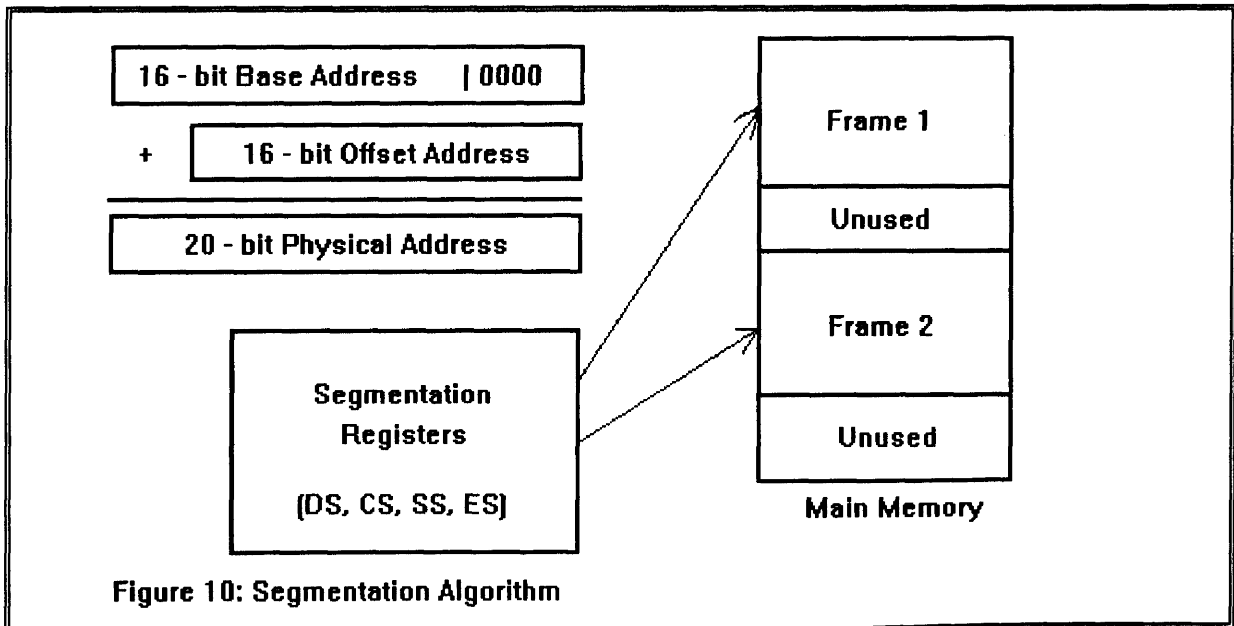


Figure 10: Segmentation Algorithm

The base address is generally stored in one of the 486's segmentation registers while the offset is carried around as data. This method allows for some powerful features. For example, to change the location of a program in memory, simply change the value of the base address, all other addresses should be just offsets to this base. This method could also allow for two completely separate frames of 64 K each to exist in memory at once by using two different base addresses. The internal stack uses exactly this method by having its base address stored in a different segmentation register than the code or data base addresses.

To execute an instruction, the processor must first determine if a memory access is required. If so, the physical memory address must be calculated by the segmentation unit, the result of which is placed on the internal address bus (connected to the address inputs of the cache). The values to be operated on will then be read and placed on the internal operand buses. The ALU or shifter will then perform the necessary functions, with the result being placed on the databus. If the results are to be stored to memory, the previous physical memory calculation would still be valid, resting at the address bus. The write back phase can then return the result to either the cache or one of the registers.

4. The Control Path

The control path is responsible for getting instructions from memory, decoding them, and manipulating the datapath to create the desired results. To this end, the prefetch unit, with the paging unit to generate physical addresses for it, periodically fetches instructions from the cache. As mentioned previously, accesses to the cache can only occur when it is not in use by the datapath. Once the instruction has been fetched, it is passed to the decode unit when the next instruction is requested. The decode unit in the simulation will directly control the datapath (see figure 11).

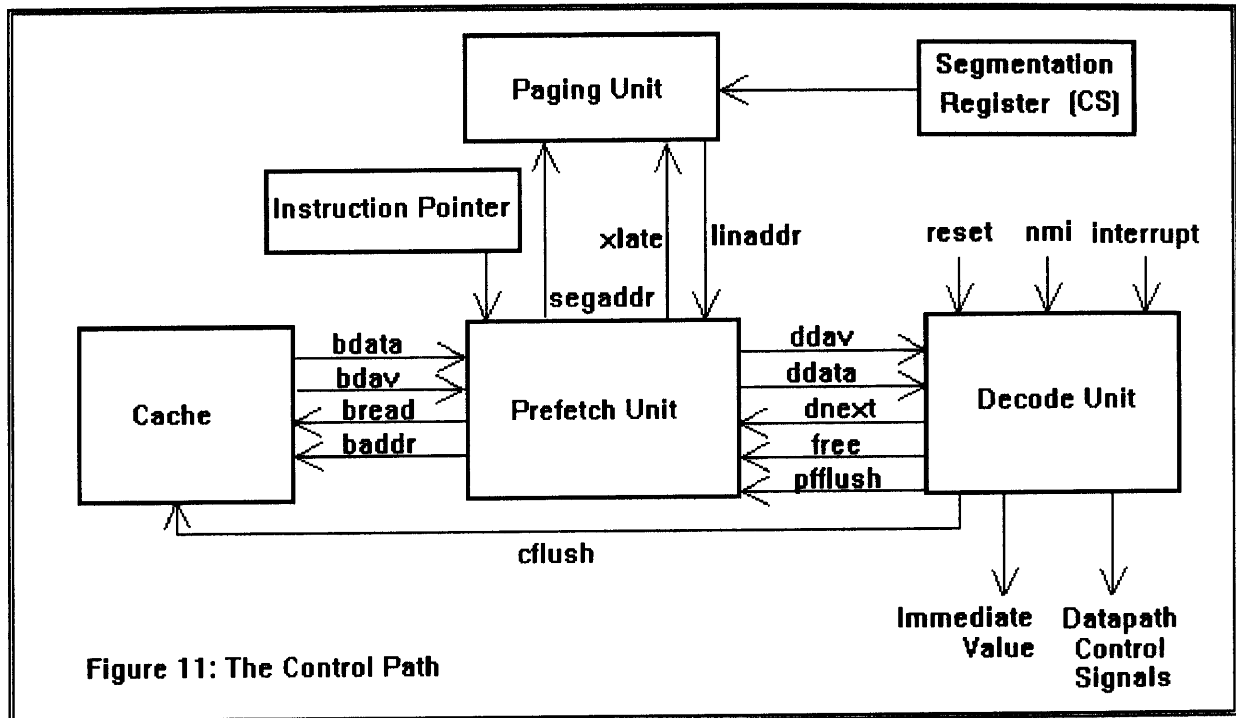


Figure 11: The Control Path

The prefetch unit gets instructions from the cache when it sees that the cache is free or as requested by the decode unit. The prefetch has a 32-byte internal buffer, with cache reads providing 16-bytes (one cache line) at a time. It knows where to get the next instructions from the value in the instruction pointer. With this simulation, the instruction pointer starts out at location $0000. This is a segmented offset, which together with the base in the code segmentation register (CS), must be translated by the paging unit into a physical address. The paging unit is a dedicated segmentation unit for sole use by the prefetch unit.

It should be noted that the prefetch's internal buffer is flushed upon a reset or a jump. In those circumstances, the instructions in the buffers will no longer be valid. While this will slow things down with a lot of jumps, the prefetch buffers are there so that the regular delays from memory accesses are diminished (see figure 12).

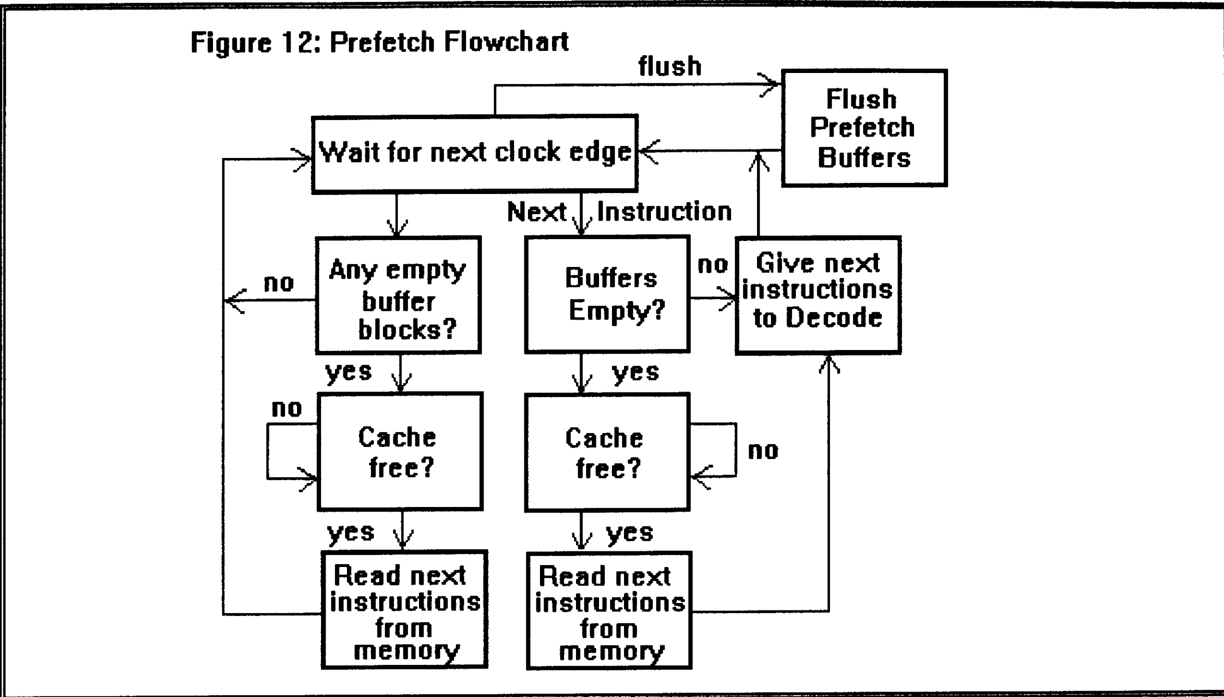


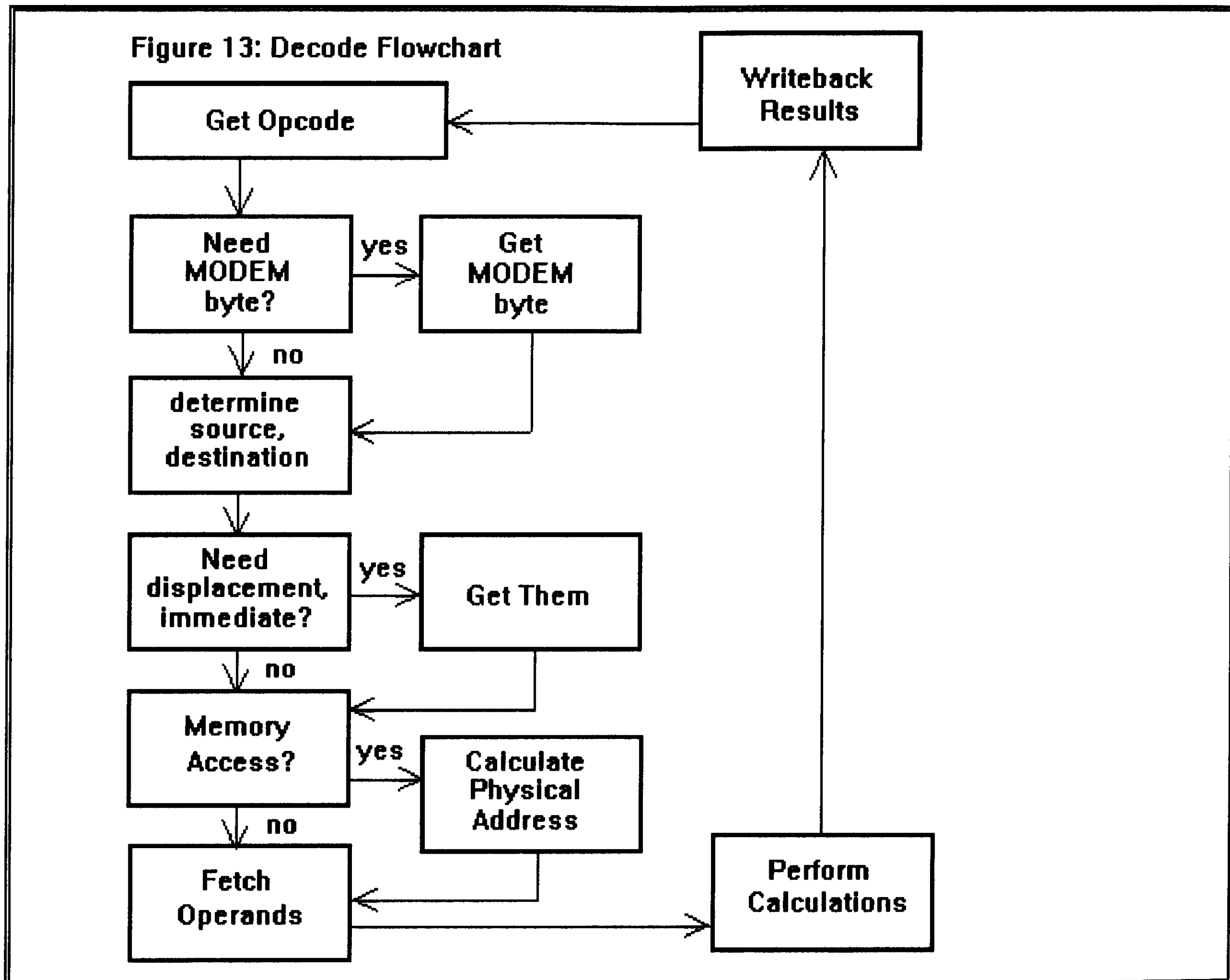
Figure 12: Prefetch Flowchart

The decode unit takes data from the prefetch unit four bytes at a time as needed. For this simulation, all of the instruction decoding and datapath control occurs in this unit, in addition to responding to the external reset pin. The decode unit first looks at the opcode and determines if a "modem" byte is required. The modem contains variable source and destination information. The decode unit will then check to see if memory displacements or immediate values are needed. Immediate values will be held in a buffer and deposited to the second operand bus (there are two). The possible need for memory access must next be determined. If so, the physical address is calculated and the operands are fetched. The ALU or shifter is then given the appropriate instructions and the result piped to the destination (see figure 13).

5. Instruction Execution

For a single instruction to execute, remember that the four phases of the microprocessor cycle, fetch, decode, execute, and writeback, must be completed. For an instruction to be fetched, it must first exist in external memory, at the location where the

instruction register is pointing. It obviously must be in a low level binary or hex format (more on this later). At power-up, the reset pin is asserted and the processor enters a reset state, flushing the prefetch buffers, the cache, and all other internal buffers. The instruction pointer resets to $0000, with $0000 held in all segmentation and general purpose registers.



Upon exiting the reset state, the decode unit will see that there are no instructions for it to work on. A request will thus be sent to the prefetch unit. The prefetch unit, seeing its buffers empty, requests the next instruction, pointed to by the instruction register, from the cache. The cache, having just been flushed, will been unable to find the instruction's address in its memory. So the request is passed on to the bus interface unit. The bus interface unit,

noting that its write buffers are empty (so a reordering is unnecessary), will execute a burst read from RAM.

The burst read fetches 16 bytes (four double words) from memory. These 16 bytes are then sent to the cache to fill a cache line. If the cache had not been empty, the new data would have overwritten the least recently used "way" in the set and replaced the tag address. Once in the cache, the new data would be sent to the prefetch unit to be place in its buffers. The first 4 bytes of the 16 will then finally reach the decode unit, sent by the prefetch unit. This ends the fetch phase.

The decode unit takes a look at the first byte of the four that it received. This should be the opcode. If it does not match any supported opcode, it will be treated as a NOP (basically ignoring it). Identifying the opcode, the decode unit determines the need for the modem, displacement, and immediate data. If needed, they are fetched from the 4-byte decode queue. The source and destinations are then decided and whether memory access is needed is resolved. At this point, the operands are fetched (with physical addresses calculated as needed) by the decode unit putting the appropriate commands to the datapath. This ends the decode phase.

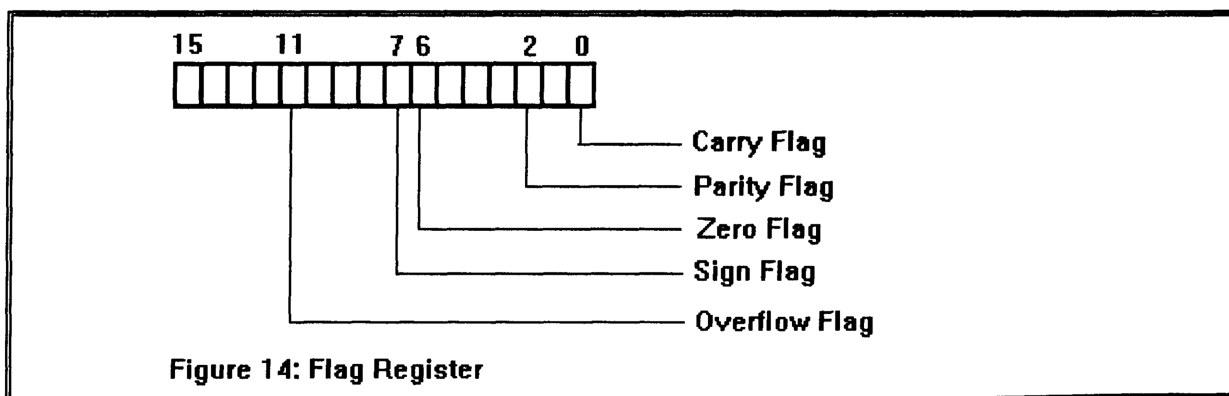


Figure 14: Flag Register

The execution phase occurs in the arithmetic logic unit or the shifter. They will set the flag register with information about the result (see figure 14). The result from the execution

phase can then be written to the general purpose registers, the segmentation registers, the cache, or the instruction register (in the case of a jump). The registers can be written to on the next immediate clock edge. The cache, on the other hand, will require at minimum the next cycle to complete, longer if a cache miss occurs. This concludes the writeback phase (see figure 15). The entire cycle then repeats itself.

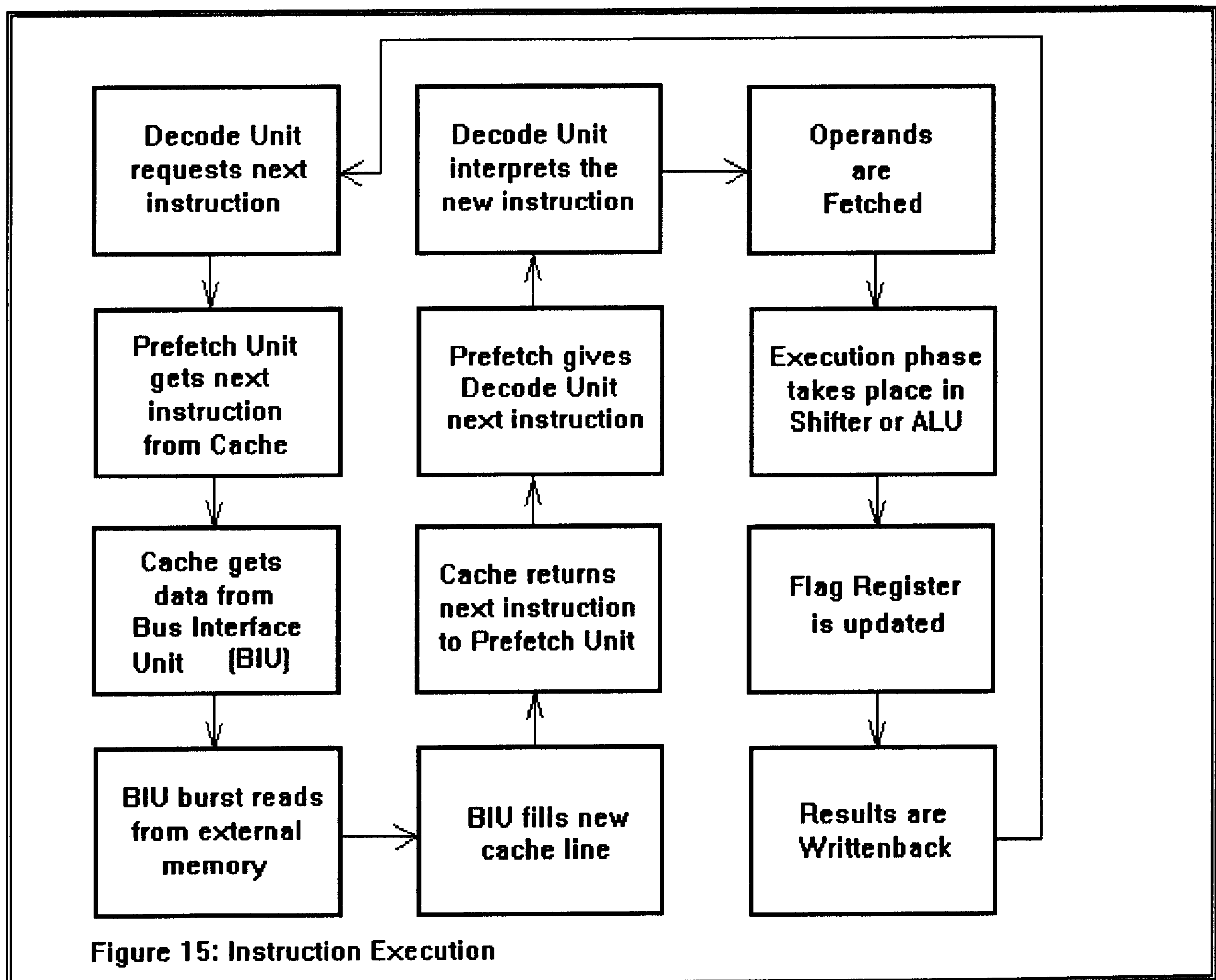


Figure 15: Instruction Execution

6. Differences with the Intel 486 SX

While care has been taken to adhere this model as closely as possible to the actual Intel 486 SX Microprocessor, certain details were simplified or altered to insure completion of this project. First and foremost, the model only supports microcode for the major assembly

instructions, though all memory access encodings supported in real mode are accepted. In keeping with real mode architecture, only 16 bit registers and buses are used. During a stack operation, while consistent with the 8086 but not with later processors,, the stack is updated first then the stack pointer, rather than the other way around. With 32-bit external memory, all data must be aligned correctly. The bus interface unit will not compensate for non-aligned data. And as mentioned earlier, initial power-up debugging features are not supported. Since the debugging feature is simply a way to insure the proper functionality of the physical chip, a software simulation of such is unnecessary.

Other noncompatibilities with the actual chip involve using only burst mode read and writes from the bus interface unit, no I/O other than external memory accesses, all numbers being sign extended, no page faults, no instruction prefixes, and no interrupts are supported. There is no processor control register, only certain flags are implemented, and the stack grows up in memory (towards larger address memory) rather than down. In addition, while many of the instructions exhibit proper timing, not all the instruction timings are exact. The complicated interaction between the different components and the lack of detailed schematics from Intel have prevented this. So while this microprocessor may be unable to execute some stock 486 code, the hand compiled code which does work, both on the model and on the real 486, will look remarkably similar to actual code.

III. The Verilog Model

1. 'x86 Assembly Code

There are over 120 assembly instructions that the 486 SX can run, however, this model only supports a small subset of the most commonly used commands. To run these instructions, they must first be compiled to a binary or hex format. The microprocessor obviously can only take in binary inputs and not full textual instructions. For each instruction, there may be several locations where the operands lie. They may be in registers, memory locations, the segmentation registers, immediate data, or in the instruction pointer in the case of jumps.

There are eight 16-bit general purpose registers available on the microprocessor, ax, bx, cx, dx, sp, bp, si, and di. While the sp register is used as the internal stack pointer and the ax is used as the accumulator for some instructions, all eight registers may be used for any purpose. The four segmentation registers, ds, cs, ss, and es are used to keep the base portion of a segmented address. The ss register is used for the stack, the cs for the instruction pointer, the ds for memory accesses, and the es for string operations not supported in this simulation.

To address memory, the base register in the segmented address can be either in the ds register or the ss register as desired. The offset value can be a simple 16-bit displacement value, a value in a register, a value in one register plus a value in another, or a combination of these. Most instructions require a "MODEM" byte which allows the processor to know exactly which addresses are needed. Table 2 shows the possible combinations of addresses and how they are encoded into binary.

The modem byte consists of three parts, mod, reg, and rm. As shown in table 2, the three bit reg (bits 5 to 3) specifies one of the eight general purpose registers. At times, when a segmentation register is required instead, the reg value will specify the proper segmentation register. The combination of the two bit mod (bits 7 and 6) and the three bit rm (bits 2 to 0) will specify a base-offset segmented address or another general purpose register as required.

In eight bits, the modem byte gives us information about both the source and destination for that instruction [7].

Table 2: The MODEM Byte

rm	mod = 00	mod = 01	mod = 10	mod = 11	reg	sreg
000	DS: [BX+SI]	DS: [BX+SI+disp8]	DS: [BX+SI+disp16]	ax	000	ds
001	DS: [BX+DI]	DS: [BX+DI+disp8]	DS: [BX+DI+disp16]	cx	001	cs
010	SS: [BP+SI]	SS: [BP+SI+disp8]	SS: [BP+SI+disp16]	dx	010	ss
011	SS: [BP+DI]	SS: [BP+DI+disp8]	SS: [BP+DI+disp16]	bx	011	es
100	DS: [SI]	DS: [SI+disp8]	DS: [SI+disp16]	sp	100	xxx
101	DS: [DI]	DS: [DI+disp8]	DS: [DI+disp16]	bp	101	xxx
110	DS: [disp16]	SS: [BP+disp8]	SS: [BP+disp16]	si	110	xxx
111	DS: [BX]	DS: [BX+disp8]	DS: [BX+disp16]	di	111	xxx

A look at table 3, the supported instruction set, will indicate that some instructions do not require a modem byte. This may be because the source or destinations are encoded in the opcode itself, or the operand may be immediate data or a direct memory displacement off the ds segmentation register. Of the instructions requiring a modem byte, a bit in the opcode may be necessary to determine if the reg field is the source or the destination. This direction bit (d) is high when the register specified by reg is the destination.

The accepted method of writing x86 assembly code is for the instruction to be followed by the destination, then the source. Any memory references are to be placed in square brackets. This go for memory displacements as well (see appendix 4 for examples).

The binary encoding of the assembly instruction begins with the opcode. The opcode will define the instruction, indicating if subsequent bytes, such as the modem or immediate bytes are needed. It may also include information about the direction of the modem byte and the width of the operands. The width bit will be high if 16-bit operands are used, otherwise 8-bit operands will be in effect. In this implementation, certain bits in the opcode may not be needed. They are indicated by an x in table 3, meaning that may take on any value.

So the opcode byte or bytes may be followed by a modem byte. The modem byte will specify if a displacement byte or bytes are needed. This may in turn be followed by immediate values, up to 16 bits. With the x86 binary encoding scheme, an instruction may vary from

only one byte up to seven bytes, depending on the instruction and how it is begin used. It is also important to remember that all values used in this simulation are in two's complement notation and will be sign extended to 16 bits as needed. Also, x86 assembly code uses the little endian method of storing data. This means that the least significant byte comes first. So *MOV ax, $ABCD* will be encoded as *$B8CDAB*, with *$ABCD* being a negative 2's complement number.

Table 3: Supported Instructions and Their Binary Encodings

MOV	reg/ mem, reg/ mem	100010dx	mod-reg-rm	disp	
MOV	reg, immed	1011w reg	immed		
MOV	reg/ mem, immed	1100011w	mod-000-rm	disp	immed
MOV	ax, mem	1010000x	disp		
MOV	sreg, reg/ mem	10001110	mod-sreg-rm	disp	
MOV	reg/ mem, sreg	10001100	mod-sreg-rm	disp	
CMP	reg/ mem, reg/ mem	001110dx	mod-reg-rm	disp	
CMP	reg/ mem, immed	100000xw	mod-111-rm	disp	immed
CMP	ax, immed	0011110w	immed		
POP	reg	01011 reg			
POP[1]	mem	10001111	mod-000-rm	disp	
PUSH	reg	01010 reg			
PUSH[2]	mem	11111111	mod-110-rm		
ADD	reg/ mem, reg/ mem	000000dx	mod-reg-rm	disp	
ADD	reg/ mem, immed	100000xw	mod-000-rm	disp	immed
ADD	ax, immed	0000010w	immed		
SUB	reg/ mem, reg/ mem	001010dx	mod-reg-rm	disp	
SUB	reg/ mem, immed	100000xw	mod-101-rm	disp	immed
SUB	ax, immed	0010110w	immed		
IMUL[3]	reg/ mem	1111011x	mod-101-rm	disp	
IMUL	reg, immed	011010ϖ1	mod-xxx-rm	disp	immed
IMUL	reg, reg/ mem	00001111	10101111	mod-reg-rm	disp
IDIV[4]	reg/ mem	1111011x	mod-111-rm	disp	
INC	reg/ mem	11111110	mod-000-rm	disp	
DEC	reg/ mem	11111110	mod-001-rm	disp	
AND	reg/ mem, reg/ mem	001000dx	mod-reg-rm	disp	
AND	reg/ mem, immed	100000xw	mod-100-rm	disp	immed
AND	ax, immed	0010010w	immed		
OR	reg/ mem, reg/ mem	000010dx	mod-reg-rm	disp	
OR	reg/ mem, immed	100000xw	mod-001-rm	disp	immed
OR	ax, immed	0000110w	immed		
XOR	reg/ mem, reg/ mem	001100dx	mod-reg-rm	disp	
XOR	reg/ mem, immed	100000xw	mod-110-rm	disp	immed

[1] This instruction will affect the value in the ax register.

[2] This instruction will affect the value in the ax register.

[3] location specified by MODEM = value in ax * value in location specified by MODEM

[4] ax = value at MODEM / value in ax, dx = value in modem mod value in ax

NOT	reg/ mem	1111011x	mod-010-rm	disp	
NEG	reg/ mem	1111011x	mod-011-rm	disp	
ROL	reg/ mem, 1	1101000x	mod-000-rm	disp	
ROL	reg/ mem, immed8	11000000	mod-000-rm	disp	immed8
ROR	reg/ mem, 1	1101000x	mod-001-rm	disp	
ROR	reg/ mem, immed8	11000000	mod-001-rm	disp	immed8
SHL	reg/ mem, 1	1101000x	mod-100-rm	disp	
SHL	reg/ mem, immed8	11000000	mod-100-rm	disp	immed8
SHR	reg/ mem, 1	1101000x	mod-101-rm	disp	
SHR	reg/ mem, immed8	11000000	mod-101-rm	disp	immed8
NOP[5]	10010000				
JMP[6]	disp8	11101011	disp8		
JMP	disp16	11101001	disp16		
JMP	reg/ mem	11111111	mod-100-rm	disp16	
Jcond	disp8	0111cccc	disp8		
Jcond	disp16	00001111	1000cccc	disp16	
JE	0100	equal			
JG	1111	greater than			
JGE	1101	greater than or equal			
JL	1100	less than			
JLE	1110	less than or equal			
JNC	0011	not carry			
JNE	0101	not equal			
JNL	1101	not less than			
JNLE	1111	not less than or equal			
JNO	0001	not overflow			
JNP	1011	not even parity			
JNS	1001	not negative			
JNZ	0101	not zero			
JO	0000	overflow			
JP	1010	even parity			
JPE	1010	parity even			
JPO	1011	parity odd			
JS	1000	negative			
JZ	0100	zero			

So to create a program for this processor, one must first write the machine instructions in assembly code. The assembly code must then be hand compiled into binary or hex. That must then be placed in the Verilog module ram, starting at location $0000. Location $0000 will be where the first instructions will taken from after the power-up reset is completed. Without I/O support, the best way to examine returned values is for the program to write results to external RAM and display the results found there upon completion. Inputs to the

[5] Any other opcode not previously specified can be used as a NOP.
[6] The jump displacement begins from the location of the first byte after the complete jump instruction, so a JMP -2 will put the processor in a continuous jump loop.

system will have to be predetermined and place in strategic locations in external RAM before the program starts (refer to appendix 4 for sample assembly code and their binary encodings).

2. Model Functionality

Verilog code is broken down into separate functional blocks called modules. The microprocessor itself is broken up into 13 independent modules operating simultaneously. In order for the simulation to do anything however, an external module must simulate the clock, RAM, and external control signals to the processor. The file *testing.v* (module *system*) does all this and provides a binary encoded program to exhaustively test out the simulation (more on this later). This "system" interfaces with the module *i486*, which is the highest level module for the microprocessor, connecting all the major components of the simulation together.

Verilog code is made of several basic building blocks. The different components of a system are placed in what are called modules. Modules have carefully defined inputs, outputs and functionalities. These modules are then connected together with "wires" in a higher level module. The highest level module has no inputs or outputs. It simply takes the other modules as submodules, defines the connections between them, and provides the external stimulus to start off the system. The module *system* is the highest level module in this simulation and provides the entire off-chip environment for the microprocessor.

Within the modules, the inputs, outputs, and functionality must be defined. The inputs are defined as wires and the outputs as registers. An I/O port is defined as an "inout". It has a register connected to it to provide for the output. This register is set to a high impedance state when the inout is to be used as an input. Internally, integers may be utilized as temporary storage variables in defining module functionality.

In defining the functionality of each module, the simulation applies three different building blocks, the initial block, the always block, and the task block. The initial block acts like a standard linear program. It starts at the beginning and ends at the end. Time passes (for

the simulation) while in the initial block only when a specific delay has been requested. For example, a command such as *#3000 finish;* could appear in an initial block to tell the Verilog simulator to end the simulation after 3000 simulation time units have passed.

The always block, as the name may imply, operates in a continuous loop. It will jump to the beginning of the block once the last line has been executed. Generally, one does not want this transition to occur instantaneously (in terms of simulation time). So a delay such as *always @(clk) begin* will prevent the loop from restarting until the clock signal (an input into this module) has toggled. Always blocks with absolutely no delays will obviously tie up the simulator as it continuously loops in circles. Tasks are used in this simulation as subsets in the always blocks. They act as subroutines or functions, with definite inputs and outputs, to perform repetitive tasks.

The power of Verilog comes in when there are 50 different always and initial blocks running in parallel at the same time, each dependent upon others. This may be spread out among many different modules. A short always block may have progressed through a hundred loops before a long (in simulation time) one has even gone through one. As mentioned previously, things in Verilog occur in parallel. There is no guarantee that one always block will finish with a particular function before another always block finishes with something else unless explicit delays or event triggers are applied. An expected signal that comes to early or too late may lock up an entire series of modules. Debugging for such a system can get quite complicated at times.

module i486 (aout, dout, din, bcdef, blast, brdy, ads, ain, rdy, a20m, hold, hlda,boff, breq, bsize, dpin, dpout, parity, ahold, eads, interrupt, nmi, reset, clk);[7]

As mentioned earlier, module i486 is the top level module for the microprocessor. There are only two levels of modules to this simulation. i486 takes all other modules (except

[7] found in the file i486.v

the external system module) as submodules and provides for the interconnections between them. It also contains the general purpose and segmentation registers, as well as the cache usage semaphore. The input and outputs to i486 are meant to simulate the input and output pins of the physical chip. Table 4 details the function of these pins.

Table 4: Module i486

aout	output [31:2]	address output connected to external 32-bit RAM
dout	output [31:0]	data output from the 486 to external RAM
din	input [31:0]	data input from external RAM to the 486
bcdef	output [4:0]	defines the bus cycle as reads (25, 26, or 27) or writes (30 or 31)
/blast	output	indicate the last byet being transfered during a burst cycle
/brdy	input	signal from memory indicating that data is ready
/ads	output	a request to external RAM to initiate a memory operation
ain	input [31:4]	address going into the 486 for cache line invalidation or in a multiprocessor system
/rdy	input	indicates data from RAM is ready in non-burst modes (not supported)
/a20m	input	masks address bit 20 to emulate 20-bit addressing (not supported)
hold	input	relinquishes control of the external bus
hlda	output	acknowledges receipt of the hold signal (above)
/boff	input	relinquishes control of the external bus (similiar to hold above)
breq	output	signals the start of a memory operation
bsize	input [1:0]	defines the size of the external bus (only 32-bit memory is supported, bsize=3)
dpin	input [3:0]	data parity inputs from the system
dpout	output [3:0]	data parity outputs from the 486
/parity	output	parity status pin
ahold	input	stops the 486 from driving the bus and waits for input onto the external bus
/eads	input	performs a cache invalidation on the address specified by the external bus
interrupt	input	external interrupt (not supported)
nmi	input	non-maskable interrupt (not supported)
reset	input	external soft reset signal
clk	input	the external clock

module biu (ads, rdy, a20m. bcdef, hold, hlda, boff. breq, brdy, blast, bsize, benable, aout, ain, dout, din, clk, cdout, cdin, a, cdav, cdread, cdwrite, miss);[8]

Table 5: Module biu

/ads	output	external signal connected to module i486
/rdy	input	external signal connected to module i486
/a20m	input	external signal connected to module i486
bcdef	output [4:0]	external signal connected to module i486
hold	input	external signal connected to module i486
hlda	output	external signal connected to module i486

[8] the bus interface unit is found in the file biu.v. The module emulates the functions described previously for the biu. The majority of the external control signals go directly into this module.

/boff	input	external signal connected to module i486
breq	output	external signal connected to module i486
/brdy	input	external signal connected to module i486
/blast	output	external signal connected to module i486
bsize	input [1:0]	external signal connected to module i486
benable	output [3:0]	external signal connected to module i486
aout	output [31:2]	external signal connected to module i486
ain	input [31:4]	external signal connected to module i486
dout	output [31:0]	external signal connected to module i486
din	input [31:0]	external signal connected to module i486
clk	input	external signal connected to module i486
cdout	output [31:0]	data output from biu to cache
cdin	input [31:0]	data input from cache to biu
a	input [31:2]	address of data sent from cache to biu or requested by cache
/cdav	output	acknowledge that data from cache has been received by the biu
/cdread	input	cache requests data from external RAM
/cdwrite	input	cache would like to write data to external RAM
miss	input	a cache miss has occured

module cache (cdout, cdin, a, cdav, cdread, cdwrite, miss, ahold, eads, ain, baddr, bdatain, bdataout, bread, bdav, bwrite, cflush, clk);[9]

Table 6: Module cache

cdout	input [31:0]	data line from biu to cache
cdin	output [31:0]	data line from cache to biu
a	output [31:2]	address line from cache to biu
/cdav	input	data ready signal from biu
/cdread	output	signal to biu to request a read from external RAM
/cdwrite	output	signal to biu to request a write to external RAM
miss	output	signal to biu that a cache miss has occured
ahold	input	external control signal that stops the external bus and waits for off-chip input
/eads	input	external signal to perform a cache line invalidation specified by the external bus
ain	input [31:4]	external address bus
baddr	input [31:0]	address line from the datapath to the cache
bdatain	input [31:4]	data line from the datapath to the cache
bdataout	output [31:0]	data line from the cache to the two datapath operand buses
/bread	input	a cache read request from the control path
/bdav	output	cache data is ready signal to the control path
/bwrite	input	a cache write request from the control path
/cflush	input	cache flush signal
clk	input	external clock signal

[9] the cache is found in the file cache.v

module pchk (dpin, dpout, parity, din, dout, benable, bsize, bcdef, ads, brdy, aout, clk);[10]

Table 7: Module pchk

dpin	input [3:0]	data parity pin inputs to insure data input bytes are even parity
dpout	output [3:0]	data parity pin outputs to insure output data bytes are even parity
/parity	output	asserts if parity check fails
din	input [31:0]	data into the microprocessor from external RAM
dout	input [31:0]	data leaving the microprocessor for external RAM
benable	input [3:0]	controls width of external memory (only 32-bit mode is supported)
bsize	input [1:0]	defines the size of the external bus (only 32-bit mode is supported)
bcdef	input [4:0]	defines a read or write to memory
/ads	input	request a for memory operation
/brdy	input	data ready signal from external RAM
aout	input [31:2]	output address line from the microprocessor to external RAM
clk	input	external clock signal

module tristate (in, out, state);[11]

Table 8: Module tristate

in	input	input to tristate gate
out	output	output of tristate gate, may be high, low, or a high impedence state
state	input	asserted, it passes the input to the output, otherwise it stays at high impedence

module shifter (bus1, sftout, dir, type, count, clk, setflags, fdone);[12]

Table 9: Module shifter

bus1	input [15:0]	datapath operand bus 1
sftout	output [15:0]	connected by a tristate to the databus
dir	input	shift or rotate direction, 0 = left, 1 = right
type	input	choose shift or rotate, 0 =shift, 1 = rotate
count	input [4:0]	number of locations to shift or rotate
clk	input	external clock signal
setflags	inout	output to flag module to set the flags (the alu does this also so a inout is needed)
fdone	input	signal from the flag module to indicate that it has finished setting the flag register

module segmentation (saddr, addrbus, xlate, segregs, clk, spage, sdisp);[13]

Table 10: Module segmentation

[10] the parity check unit is found in the file parity.v
[11] found in file sections.v
[12] found in file sections.v
[13] found in file sections.v

saddr	input [31:0]	segmentated offset of memory address
addrbus	output [31:0]	physical address output
xlate	input	control signal to start physical address translation
segregs	input [15:0]	segmented base of memory address
clk	input	external clock signal
spage	output	page fault indicater (not supported)
sdisp	input [15:0]	memory displacement

module paging (pfsaddr, linaddr, pxlate, cs, clk, ppage);[14]

Table 11: Module paging

pfsaddr	input [31:0]	segmentated offset of memory address
linaddr	output [31:0]	physical address output
/pxlate	input	control signal to start physical address translation
cs	input [15:0]	segmented base of memory address, CS register
clk	input	external clock signal
ppage	output	page fault indicater (not supported)

module address (load, addrcnt, pcin, eip, clk);[15]

Table 12: Module address

load	input	control signal to load new address into register
addrcnt	input	control signal to increment address by 4 bytes
pcin	input [31:0]	input address to be loaded into register
eip	output [31:0]	output of instruction pointer
clk	input	external clock signal

module flag (setflags, value, flagreg, set, fdone);[16]

Table 13: Module flag

setflags	input	flag register is updated if setflags and set are both asserted
value	input [15:0]	value of result from alu or shifter
flagreg	output [15:0]	output of flag register
set	input	flag register is updated if setflags and set are both asserted
fdone	output	signal indicating that the flags have completed being set

[14] found in file sections.v
[15] found in file sections.v
[16] found in file sections.v

module alu (op1, op2, aluout, instr, clk, setflags, alusupp, fdone);[17]

Table 14: Module alu

op1	input [15:0]	operand 1 from internal operand bus 1
op2	input [15:0]	operand 2 from internal operand bus 2
aluout	output [31:0]	result of computations
instr	input [7:0]	instructions to alu (see appendix 2 for details)
clk	input	external clock signal (not used)
setflags	inout	sets flag register (shared output with shifter)
alusupp	input[2:0]	supplementary input to alu (used in jump calculations)
fdone	input	signal indicating that the flags have completed being set

module prefetch (ddav, dnext, ddata, pfsaddr, linaddr, pxlate, ppage, get, pfflush, eip, free, bdav, bread, bdataout, baddr, clk);[18]

Table 15: Module prefetch

/ddav	output	data to decode unit is available signal
/dnext	input	signal from decode unit requesting next instruction
ddata	output [31:0]	data to decode unit (the next instruction)
pfsaddr	output [31:0]	segmented offset output to paging unit
linaddr	input [31:0]	physical address output from paging unit
/pxlate	output	command to paging unit to generate physical address
ppage	input	page fault indicator from paging unit (not used)
get	output	signal to take hold of the cache usage semaphore
pfflush	input	signal to flush the prefetch unit's internal buffers
eip	input [31:0]	output of instruction pointer
free	input	cache usage semaphore
/bdav	input	handshaking signal between cache and prefetch unit (see appendix 3 for details)
/bread	inout	signal to cache to request a value from memory
bdataout	input [31:0]	data from the cache to the prefetch unit
baddr	inout [31:0]	address to and from the cache from the prefetch unit
clk	input	external clock signal

module decode (ddata, ddav, dnext, pfflush, free, cflush, addrcnt, immed, interrupt, nmi, reset, instructions, get, bread, bwrite, bdav, disp, bus2, flag, clk);[19]

[17] found in file sections.v
[18] found in file prefetch.v
[19] found in file decode.v

The decode module contains all the microcode used in the simulation. The microcode gives the datapath instructions to execute the assembly code that the decoder identifies. The output "instruction" in the module carries the datapath controls. The control listing is detailed in appendix 2.

To create the microcode, the decode uses several task and always blocks as subroutines (see table 16). Fetch takes in information about the source and destination, and the instructions to the alu or shifter. The most significant bit in its input variable func chooses between the alu and the shifter (1 for shifter, 0 for alu). The following bits map to the instructions that control them (detailed in appendix 2). Fetch will output instructions to the datapath to obtain the requested operands and perform the requested function. In cases where the destination is a register, it will also load the register with the result. So a register to register operation requires only one clock cycle and not much more overhead than this task.

Table 16: Decode Subroutines

Task	Inputs	Outputs
Fetch	func[8:0], d, modem[7:0], dest, src	instructions[79:0], bread
Calc	modem[7:0]	instructions[79:0]
Next	incnt, t0, t1, t2, t3	var[7:0], outcnt, update
Signextend	original, result, in[31:0]	out[31:0]
Save	instructions[79:0]	instructions[79:0], bwrite
Jmp	immedin[15:0], count	resume, immed[15:0], rem

When an instruction is first decoded and a memory request has been identified, the internal variable "check" should be set high. Check, an always block, will then pull in any displacements to memory and any immediate values. It then proceeds to lock access to the cache and calls on the task "calc". Calc takes in the MODEM byte as input. It decodes the modem byte and goes about calculating the physical address from the segmented address. The main decode loop can resume operation at the negative edge of check. This whole process will take one clock cycle.

The task "next" is used to get the next byte from the decode's internal 4 byte queue. It takes as input the four bytes and a pointer indicating the next byte. It will output the

requested byte and update the pointer. If the next double word is needed from the prefetch unit, next will indicate such to another decode block that will update the internal decode queue. The process of taking the next byte requires no simulation time unless the queue is empty and a request to the prefetch unit is in progress.

As mentioned earlier, all data in this simulation is sign extended. The task "signextend" takes in a number, the number of bits that are significant in that number, and the number of significant bits desired in the output. It will then return the result sign extended. It is generally used to transform from 8 bits to 16 bits. This task takes zero simulation time.

While the subroutine fetch will writeback to registers automatically, if the destination is in memory, the procedure memsave is used. When the variable "memsave" is set to 1, an always block in the decode module will use the task "save" to deposit a value in the cache. This assumes that a valid address and valid data is already available and waiting on the internal data and address buses. This writeback procedure will take a minimum of one clock cycle (depending if this is a cache hit, etc.) and memsave will deassert when complete.

"Jmp" is used to calculate the new value for the instruction pointer during jumps. It must take as input the internal decode queue pointer and the value of the jump offset. It returns a value to increment to the instruction pointer, the new queue pointer, and a resume flag to indicate that a jump has occurred. This procedure does not flush the prefetch buffer, that must be done at the end of the microcode instruction separately. "Jmp" takes zero simulation time.

Table 17: Module decode

ddata	input [31:0]	incoming bytes from prefetch unit
/ddav	input	data is available signal from prefetch unit
/dnext	output	request for more data to prefetch unit
pfflush	output	flush the prefetch unit's internal buffers
free	input	cache usage semaphore
/cflush	output	flush the cache
addrcnt	output	increment the address register (instruction pointer)
immed	output [15:0]	immediate value to datapath
interrupt	input	external interrupt signal (not supported)
nmi	input	external non-maskable interrupt (not supported)
reset	input	external soft reset

instructions	output [79:0]	control signal to datapath
get	output	request for cache usage semaphore
/bread	inout	request for cache data (shared with prefetch)
/bwrite	inout	request to write to cache
/bdav	input	data is available signal from cache (see appendix 3 for details)
disp	output [15:0]	memory displacement to segmentation unit
bus2	input [15:0]	internal opernad bus 2
flagreg	input [15:0]	output of flag register
clk	input	external clock signal

3. Possible Changes and Modifications

Since this simulation is not quite the 486, several modifications can be made to make it conform more closely to the actual chip. Additions to the microcode can be made, such as adding subroutine functions. Some changes to the bus interface unit and its bcdef signal could provide for I/O. The addition of an always block in the decode unit could be used for interrupts.

Many parts of the simulation still work on a behavioral level. These sections may or may not be physically possible or practical. For example, the instruction decode unit could be made more explicitly. Currently, the pipeline there is behavioral. Registers could be placed so that it is properly pipelined and the opcode bits decode to a control address where the microcode for that instruction is placed. The ALU could also be more explicit. This is really not too difficult. It just requires making an adder, subtractor, a multiplier, and so forth with a multiplexer to select between them.

As the model stands now, it can be a bit tedious entering new code for the microprocessor. An assembler that only uses the supported instructions could be a big help. This assembler could generate the entirety of the system module. It should generate the clock, the external signals, and the external RAM all together. This way, the binary encodings for the instructions could be directly injected into RAM locations. This would not be difficult, the structural components are already fixed, so a few if statements and a MODEM look-up table would be all that is necessary.

Adding new microcode would probable be one of the first additions made. Here is an overview of how the subroutine function could be added to the decode module. First, a new else if statement would have to be added to the decode's main loop to search for the "call" opcode (11101000). We know that we will probably be returning to this location, so we will need to push the current address onto the stack. The instruction pointer will not be pointing at the current byte since it doesn't take into account the decode's internal 4-byte instruction queue. So the address of the next instruction is the value in the instruction pointer plus the value of the internal queue pointer. Before calculating the address of the following instruction, all the bytes which make up this current instruction (call) must be fetched from the queue first. In this case, the memory displacement is the only operand. Use the "next" subroutine to place the next byte in the queue into the variable disp.

Following the example for the push instruction, the physical stack address must first be calculated. Remember that the stack uses the ss segmentation register as the base and the sp register as the offset. So the value in the sp register is passed to the segmentation unit. No displacements are used here so we leave the disp tristate off. By the end of this clock cycle, a valid address (pointing to the top of the stack) should be sitting on the internal address bus leading to the cache.

On the next clock, the immediate byte should be set to the value of the queue pointer. The alu should be set to add, and the tristate on the instruction pointer be asserted. The ALU output tristate should also be asserted, with the tristate leading to the segmentation unit turned off. This will calculate the address of the next instruction. The subroutine memsave can then be asserted on the following clock cycle since both address and data will be valid waiting at the cache. Following that, the stack pointer should be incremented by 4 bytes (32-bits)

Now the address of the next instruction is on top of the stack. Our next goal is to jump to the new location. Following the example of the jmp instruction, the task jmp needs to be utilized to calculate the address we want to jump too. Its output should be stored as an

immediate value. Following that, set the alu to add, turn the instruction pointer tristate on, and execute the fetch subroutine. Fetch will place the value of the new address (sum of the immediate and instruction pointer values) into the instruction pointer. Don't forget to flush the prefetch buffers on the next clock cycle. And do not exit the microcode until the negative edge of ddav has been detected. This insures that the prefetch has finished flushing before the decode unit tries to retrieve data for its internal queue.

This completes the call command. To return from the subroutine, simply pop the old address off the stack onto internal operand bus 2. Bits 0 and 1 of the address should be placed in the decode's internal instruction queue pointer (set resume to 1 and count to bus2[1:0]). Then place the address into the instruction pointer, setting the last two bits to zero. This would operate similarly to a memory to register move. Now flush the prefetch buffers and wait for the negative edge of ddav. The program will now be at the point right after it encountered the call instruction. If the original call instruction was encoded for a direct value in memory rather than a displacement, the jmp subroutine would not have been necessary. A direct move into the instruction pointer as done here would be sufficient. Assuming no other use of the stack, the calls can be nested as far as memory will allow.

Adding additional hardware to the simulation is not difficult. Changing the size of registers or buses would only require the change of a single number. Changes such as allowing the biu to respond to external buses smaller than 32-bits would be confined to the biu module itself. The bsize external control signal determines the size of the external bus. A new if statement to catch the new bsize would be necessary. Then simply follow the handshaking signals for external memory operations shown in appendix 3, sending the 32-bit data one or two bytes at a time.

Here is how a non-maskable interrupt (nmi) function could be implemented. As mentioned earlier, an always loop would need to be added to the decode module that would trigger at a nmi (*always @(posedge nmi) begin*), with the nmi being a input to the module. Once triggered, it would have to signal the decode loop to stop at the end of the current

instruction. The main decode loop would need something like "*if (waitflag) @(negedge waitflag);*" where waitflag is the signal the new always block will set after detecting a nmi.

The nmi always block now has control of the datapath. It will need to push all the registers, including the instruction pointer, to the stack. Once done, the nmi simply has to look at a predefined location in external RAM where an interrupt vector is stored and jump there. This would be a lot like how the call command would be implemented. After the jump, the main decode loop can resume its operations, with the prefetch unit needing to fetch the new instructions into its buffer. It would service the interrupt as it would any other commands. At the end of the interrupt cycle, a return from interrupt, similar to the return from subroutine function would return the processor to its original state.

IV. Operation Verification

1. The System Tests

Since this simulation is being used as a benchmark to test out a massively parallel version of Verilog, it is important that there be extensive tests of the system. It would also be useful to have a tool to identify exactly where a problem exists if a bug should pop up. Because the simulation is currently identified by the tests as being fully functional, a error that shows up in the debugging would be an indication of a problem with the new Verilog implementation.

The debugging tools include a system test. This test is written in x86 assembly code and already inserted in RAM.[20] It will run a routine to test out its own component instructions as well as methodically test out each supported instruction, using each memory addressing mode. It then wraps up the test with a program which finds a few values of the fibinocci sequence.[21]

This system test is designed to root out small errors after the entire system is deemed functional. The test runs each instruction, compares the result to some expected value, then writes a error code to external memory. Since the execution of a single instruction involves a fairly complicated process and the system test requires that at least a few of the assembly instructions are functional, it will only be of help on more obscure problems. The component tests, covered in detail later, deals with testing on a much lower level.

At the conclusion of the system test, it will display a grid of errors on the screen to help in identifying problem areas. Table 17 details the errors that can be detected by the system test. The system test verifies its own code as well. The debugging printout must read "TEST: 000000ac", otherwise the testing procedure itself is not functioning properly. As stated earlier, a fibinocci sequence is generated at the end of the test. The results are printed out starting at external memory address $50. They continue until external address $68, with

[20] in the file testing.v, see appendix 1

[21] 1, 1, 2, 3, 5, 8, 13, 21, ... $n_x = n_{x-1} + n_{x-2}$

the value $ffffdead at location $70 to indicate the end of the sequence. On the grid itself, an "1" indicates a problem and a "0" indicates error-free operation (as far as the program can tell).

Table 18: System Test Errors

Instruction	bit 0	bit 1	bit 2	bit 3	bit 4	bit 5
MOV	ax, immed	reg, immed	mem, immed	reg, mem	mem, reg	reg, sreg
POP	reg	mem				
PUSH	reg	mem				
ADD	reg, reg	mem, immed	reg, immed			
DEC	mem					
IDIV	mem (result)	mem (mod)				
IMUL	reg	reg, immed	reg, mem			
INC	reg					
NEG	mem					
SUB	mem, reg	reg, immed	ax, immed			
AND	reg, reg	mem, immed	reg, immed			
NOT	mem					
OR	reg, reg	reg, immed				
XOR	mem, reg	mem, immed	reg, immed			
ROL	mem, 1	mem, #				
ROR	reg, 1	reg, #				
SHL	mem, 1	mem, #				
SHR	mem, 1	mem, #				
CMP	mem, reg	reg, immed				
NOP	no operands					
JMP	immed8	immed16	mem			
JCond	js immed8	jpo immed8	jpe immed8	jge immed16	jle immed16	jne immed16

2. Block-Level Testing

The block level tests are used when the system tests have failed for no apparent reasons or if the system testing routine itself is flawed. These tests verify the operation of individual blocks as well as the handshaking between them. They superimpose themselves as the top level module in the simulation, injecting inputs into the blocks and comparing the outputs to predefined values.

If the tests are positive, a display at the end will indicate such. If they find problems, an error code will be displayed indicating what went wrong. Table 18 shows what the different error codes mean. The different block tests cover the ram, pchk, paging,

segmentation, flag, address, shifter, alu, biu, cache, and prefetch modules. If these all return positive, any errors must lie in their interaction (examined by the system test) or in the decode block.

Table 19: Block-Level Tests

Module	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5
ram	burst write	burst read	end burst read	bad value		
pchk	parity good	parity bad	dp output	write to mem	dp output	
paging	cs = 0	cs != 0				
segmentation	segregs = 0	segregs != 0	disp != 0	disp = 16 bits		
flag	sign	parity	zero	different value		
address	load addr	count addr	not count addr			
shifter	left shift	right shift	right rotate	left rotate		
alu	add	return op1	return op2	compute jump	subtract	and
alu (bit+5)	or	not	xor	neg	multiply	div/mod
biu	write to mem	read from mem	different addr	different addr		
prefetch	flush	write	read	buffers	idle read	buffers
prefetch (bit+5)	buffers	buffers	forced read	forced read		
cache	cache write	cache hit	cache miss	cache hit	bad addr	flush
cache (bit+5)	cache line fill	different addr	cache hit	cache write	invalidation	

V. Conclusions

The 486 is a very complicated microprocessor. This project attempts to simulate a small portion of the actual chip, the 486 operating in real mode. To that end, the simulation understands the major assembly instructions used by the 486. It incorporates an architecture that is able to support that small subset. This does not imply that the architecture is anything like the actual chip. Only that it behaves like the 486 does in real mode, dependent upon similar building blocks.

Other than some specialized machines, microprocessors are a pretty homogeneous breed. There is a control structure that commands a data path. It has a centralized arithmetic logic unit which does all the processing. The fetch - decode - execute - writeback cycle is fairly universal. Speed improvements generally only come from better components, greater integration, and streamlining of logic. The cache on the 486, for example, speeds up operations dramatically. It uses faster memory than external ram, is integrated on-chip to reduce data transfer time, and its comparators are streamlined to enhance processing. Intel's new Pentium processor speeds up things even more by splitting up the cache into two parts so the control and data paths can access memory at the same time.

One of the most complicated parts of the 486 architecture is the decode unit. In all complex instruction set computers (CISC), assembly code must first be decoded and then translated into microcode that will assert its control over the actual processor. The new reduced instruction set computers (RISC) streamline this process considerably, eliminating microcode all together. Another architectural improvement is pipelining. The 486 is broken up into so many independent parts so that the time it takes for each part to do its task is reduced. This increases the throuput (number of instructions completed per time unit) at the expense of latency (number of time units required for a individual instruction). The ideal tradeoffs between the two are a science in themselves.

In supporting so many operations, it is difficult for a complex processor like the 486 to be completely designed on a transistor by transistor basis. So tools like Verilog are used in

non-time critical areas. This simulation is written in Verilog. It is very easy to use and syntactically similar to C. One of the most important uses of Verilog is the use of its simulation abilities. Many commercial programs actually generate Verilog code from schematics in order to simulate them effectively. More recently, VHDL, also a high level behavioral simulator, has begun edging into Verilog's traditional domain.

This simulation of the 486 attempts to recreate all the separate blocks of the processor. Only publicly available data was used. As detailed schematics were unavailable, educated guesses about the data and control paths had to be made on the basis of instruction timings alone. Despite this constraint, it is still useful to use the simulator as a base to gather information about possible improvements on the 486 design. A run of the program, for example, would clearly indicate the need for a separate instruction cache as indicated earlier. So this implementation has usage beyond its immediate application as a benchmark, especially if some minor changes are made to model it more closely to the actual chip.

Appendix 1

Simulation Code

testing.v - This the top level Verilog module which contains the ram, the clock, and the external control signals to the microprocessor. The compiled code for the system test is also included here

ram.v - This contains the external memory for the microprocessor. It is not used when testing.v is in effect.

i486.v - This file simulates the actual 486 chip. It holds together the separate modules of the microprocessor and handles input/ output lines to the chip.

biu.v - This file contains the bus interface unit of the microprocessor. It is responsible for data transfer between the internal cache and the external ram.

parity.v - This file checks the parity of incoming and outgoing data.

cache.v - This is the internal fast memory of the microprocessor. It is a 8K 4-way set associative unified instruction and data cache.

prefetch.v - This file contains the prefetch unit that periodically gets new instructions from the cache to store in its buffer. This is to speed up the decoding process.

decode.v - The decode module is responsible for interpreting incoming assembly instructions and running microcode to directly control the datapath.

sections.v - This file contains the modules tristate, shifter, segmentation, paging, address, flag, and alu. Most of the minor parts of the datapath are in this file.

totalmake - When used as a Makefile, this file will run the Verilog simulation, using testing.v. It will execute the system test followed by a short fibinocci sequence. Ignoring the 59 warnings at the start of execution, if the system test doesn't fail, the simulation should be in perfect working order.

Testing.v

```
module system;
reg [31:4] ain;
reg rdy, a20m, hold, boff, ahold, eads, interrupt, nmi, reset, brdy;
wire hlda, breq, parity, blast, ads;
reg [1:0] bsize;
reg [3:0] dpin;
reg [31:0] data [16384:0];
wire [3:0] dpout;
wire [31:2] aout;
wire [31:0] dout;
reg [31:0] din;
wire [4:0] bcdef;
integer rnw, temp, count;

clock CLK1 (clk);
i486 I1 (aout, dout, din, bcdef, blast, brdy, ads,
      ain, rdy, a20m, hold, hlda, boff, breq, bsize,
      dpin, dpout, parity, ahold, eads, interrupt, nmi, reset, clk);

initial begin                    // simulation off-chip parameters
 reset = 1;
 for (count=5120; count<5142; count=count+1)
  data[count] = 0;               // set test memory to zeros
 data[5142] = 'h2c;              // for testing the test code
 brdy = 1;
 rnw = 0;
 interrupt = 0;
 nmi = 0;
 rdy = 1;
 a20m = 1;
 hold = 0;
 boff = 1;
 ahold = 0;
 eads = 1;
 bsize = 3;                      // 32-bit external bus
 dpin = 0;                       // all data to system even parity
 #3 reset = 0;                   // finished reset cycle
end

always @(clk) begin                      // Simulation of 32-bit RAM ($0-$10000)
 if ((bcdef[0]==1||(bcdef[0]==0&&bcdef[1]==1))&&bcdef[2]==0)  rnw = 1;
      //read
```

```
 else rnw = 0;         //write
end

always @(negedge ads) begin
 #1;
 if (rnw)                //read cycle
  begin
  while (blast)
   begin
   temp=aout>>2;
   #2 din = data[temp];
   brdy=0;
   @(aout)  brdy=1;
   end
  temp=aout>>2;
  #1 din = data[temp];
  brdy=0;
  @(posedge ads)  #1 brdy = 1;
  end
 else                    //write cycle
  begin
  while (blast)
   begin
   temp=aout>>2;
   #1 data[temp] = dout;
   brdy=0;
   @(aout)  brdy=1;
   end
  temp=aout>>2;
  #2 data[temp] = dout;
  brdy=0;
  @(posedge ads)  #1 brdy = 1;
  end
 end

initial begin                  // debugging use and code entry
 data[0] =  'h90b06480;            // mov
 data[1] =  'hf8647405;
 data[2] =  'h814c0002;
 data[3] =  'h00bb0020;
 data[4] =  'h88470480;
 data[5] =  'h7f046474;
```

```
data[6] =  'h05814c00;
data[7] =  'h0100c647;
data[8] =  'h04a4807f;
data[9] =  'h04a47405;
data[10] = 'h814c0004;
data[11] = 'h008a4704;
data[12] = 'h3ca47405;
data[13] = 'h814c0008;
data[14] = 'h00884708;
data[15] = 'h807f08a4;
data[16] = 'h7405814c;
data[17] = 'h001000b1;
data[18] = 'h00b0088e;
data[19] = 'hc08cc38e;
data[20] = 'hc180fb08;
data[21] = 'h7405814c;
data[22] = 'h00200090;
data[23] = 'hb0a4bc00;            // push and pop
data[24] = 'h10bb0010;
data[25] = 'h5350807f;
data[26] = 'h04a47405;
data[27] = 'h814c0801;
data[28] = 'h005980f9;
data[29] = 'ha4740581;
data[30] = 'h4c040100;
data[31] = 'hbf0020b1;
data[32] = 'h64880dff;
data[33] = 'h35807f04;
data[34] = 'h64740581;
data[35] = 'h4c080200;
data[36] = 'h8f450880;
data[37] = 'h7d086474;
data[38] = 'h05814c04;
data[39] = 'h02005b81;
data[40] = 'hfb001074;
data[41] = 'h05814c04;
data[42] = 'h04009090;
data[43] = 'hb030b322;            // add
data[44] = 'h00d83c52;
data[45] = 'h7405814c;
data[46] = 'h0c0100bb;
data[47] = 'h00208807;
```

```
data[48] = 'h80070580;
data[49] = 'h3f577405;
data[50] = 'h814c0c02;
data[51] = 'h00050600;
data[52] = 'h3c587405;
data[53] = 'h814c0c04;
data[54] = 'h00909090;
data[55] = 'hc707b00a;          // dec
data[56] = 'hfe0f813f;
data[57] = 'haf0a7405;
data[58] = 'h814c1001;
data[59] = 'h00909090;
data[60] = 'hb004c707;          // idiv
data[61] = 'h0110f63f;
data[62] = 'h3d000474;
data[63] = 'h05814c14;
data[64] = 'h010080fa;
data[65] = 'h01740581;
data[66] = 'h4c140100;
data[67] = 'hb004b308;          // imul
data[68] = 'hf6eb80fb;
data[69] = 'h20740581;
data[70] = 'h4c180100;
data[71] = 'h6bc30280;
data[72] = 'hfb407405;
data[73] = 'h814c1802;
data[74] = 'h00bb0010;
data[75] = 'hb010c607;
data[76] = 'h040faf07;
data[77] = 'h3c407405;
data[78] = 'h814c1804;
data[79] = 'h00909090;
data[80] = 'hb064fec0;          // inc
data[81] = 'h3c657405;
data[82] = 'h814c1c01;
data[83] = 'h00909090;
data[84] = 'hf61f803f;          // neg
data[85] = 'hfc740581;
data[86] = 'h4c200100;
data[87] = 'hc60740b0;          // sub
data[88] = 'h12280780;
data[89] = 'h3f2e7405;
```

```
data[90] = 'h814c2401;
data[91] = 'h0080e803;
data[92] = 'h3c0f7405;
data[93] = 'h814c2402;
data[94] = 'h002c103c;
data[95] = 'hff740581;
data[96] = 'h4c240400;
data[97] = 'hb00fb30a;          // and
data[98] = 'h20d83c0a;
data[99] = 'h7405814c;
data[100] ='h280100bb;
data[101] ='h0010c607;
data[102] ='h0f80270a;
data[103] ='h803f0a74;
data[104] ='h05814c28;
data[105] ='h0200240e;
data[106] ='h80f80a74;
data[107] ='h05814c28;
data[108] ='h04009090;
data[109] ='hc60755f6;          // not
data[110] ='h17813faa;
data[111] ='hff740581;
data[112] ='h4c2c0100;
data[113] ='hb00abba0;          // or
data[114] ='h0008c381;
data[115] ='hfbaa0074;
data[116] ='h05814c30;
data[117] ='h010081c8;
data[118] ='ha00081f8;
data[119] ='haa007405;
data[120] ='h814c3004;
data[121] ='h00909090;
data[122] ='hbb0010c6;          // xor
data[123] ='h072ab07c;
data[124] ='h3007803f;
data[125] ='h56740581;
data[126] ='h4c340100;
data[127] ='h8137ff00;
data[128] ='h813fa900;
data[129] ='h7405814c;
data[130] ='h34020034;
data[131] ='h2a80f856;
```

```
data[132] ='h7405814c;
data[133] ='h34040090;
data[134] ='hc6070fd0;          // rol
data[135] ='h07803f1e;
data[136] ='h7405814c;
data[137] ='h380100c0;
data[138] ='h0705813f;
data[139] ='hc0037405;
data[140] ='h814c3802;
data[141] ='h00909090;
data[142] ='hb00fd0c8;          // ror
data[143] ='h3d078074;
data[144] ='h05814c3c;
data[145] ='h0100c0c8;
data[146] ='h053d003c;
data[147] ='h7405814c;
data[148] ='h3c020090;
data[149] ='hc6070fd0;          // shl
data[150] ='h27803f1e;
data[151] ='h7405814c;
data[152] ='h400100c0;
data[153] ='h2705813f;
data[154] ='hc0037405;
data[155] ='h814c4002;
data[156] ='h00909090;
data[157] ='hb00fd0e8;          // shr
data[158] ='h3c077405;
data[159] ='h814c4401;
data[160] ='h00c0e805;
data[161] ='h3c007405;
data[162] ='h814c4402;
data[163] ='h00909090;
data[164] ='hbb0010c6;          // cmp
data[165] ='h0764b064;
data[166] ='h38077405;
data[167] ='h814c4801;
data[168] ='h003c6474;
data[169] ='h05814c48;
data[170] ='h04009090;
data[171] ='heb05814c;          // jmp
data[172] ='h500100e9;
data[173] ='h0500814c;
```

```
data[174] ='h500200c6;
data[175] ='h0705ff27;
data[176] ='h814c5004;
data[177] ='h00909090;
data[178] ='hb0a77805;        // jcond
data[179] ='h814c5401;
data[180] ='h00b0077b;
data[181] ='h05814c54;
data[182] ='h0200b00f;
data[183] ='h7a05814c;
data[184] ='h5404003c;
data[185] ='h0f0f8d05;
data[186] ='h00814c54;
data[187] ='h08003c0e;
data[188] ='h0f8e0500;
data[189] ='h814c5410;
data[190] ='h003c120f;
data[191] ='h85050081;
data[192] ='h4c542000;
data[193] ='hbe005081;        // test procedure
data[194] ='hfe005074;
data[195] ='h05814c58;
data[196] ='hff0081fe;
data[197] ='h01507405;
data[198] ='h814c58a0;
data[199] ='h00909090;
data[200] ='hb650b001;        // fibinocci procedure
data[201] ='h89048944;
data[202] ='h048b048b;
data[203] ='h5c0401d8;
data[204] ='h89440882;
data[205] ='hc60483fe;
data[206] ='h64007ced;
data[207] ='hb8adde89;
data[208] ='h440c9090;
#3650;
$display("MOV:  %b",data[5120]);
$display("POP:  %b",data[5121]);
$display("PUSH: %b",data[5122]);
$display("ADD:  %b",data[5123]);
$display("DEC:  %b",data[5124]);
$display("IDIV: %b",data[5125]);
```

```
 $display("IMUL:  %b",data[5126]);
 $display("INC:   %b",data[5127]);
 $display("NEG:   %b",data[5128]);
 $display("SUB:   %b",data[5129]);
 $display("AND:   %b",data[5130]);
 $display("NOT:   %b",data[5131]);
 $display("OR:    %b",data[5132]);
 $display("XOR:   %b",data[5133]);
 $display("ROL:   %b",data[5134]);
 $display("ROR:   %b",data[5135]);
 $display("SHL:   %b",data[5136]);
 $display("SHR:   %b",data[5137]);
 $display("CMP:   %b",data[5138]);
 $display("NOP:   %b",data[5139]);
 $display("JMP:   %b",data[5140]);
 $display("JCond: %b",data[5141]);
 $display("TEST:  %h",data[5142]);
 data[27] = 0;
 data[29] = 0;
 for (count=20;count<30;count=count+1)
  $display("$%h : %h",count*4, data[count]);
 $finish;                        // end simulation
end

endmodule
```

Ram.v

```
module ram (addr, in, out, bcdef, blast, brdy, ads);
// 32-bit addressing only

input [31:2] addr;
output [31:0] out;
input [31:0] in;
input [4:0] bcdef;
input blast, ads;
output brdy;

reg brdy;
reg [31:0] data [255:0];
reg [31:0] out;
integer rnw, temp, count;

initial begin
 brdy = 1;
 rnw = 0;  //default write cycle
 end

always @(negedge ads) begin
 if ((bcdef[0]==1||(bcdef[0]==0&&bcdef[1]==1))&&bcdef[2]==0)  rnw = 1;
       //read
 else rnw = 0;         //write
 if (rnw)              //read cycle
  begin
  while (blast)
   begin
   temp=addr>>2;
   #2 out = data[temp];
   brdy=0;
   @(addr)  brdy=1;
   end
  temp=addr>>2;
  #1 out = data[temp];
  brdy=0;
  @(posedge ads)  #1 brdy = 1;
  end
 else                  //write cycle
  begin
  while (blast)
   begin
```

```
    temp=addr>>2;
    #1 data[temp] = in;
    brdy=0;
    @(addr) brdy=1;
    end
   temp=addr>>2;
   #2 data[temp] = in;
   brdy=0;
   @(posedge ads) #1 brdy = 1;
   end
  end

initial begin                    // debugging use and code entry
  data[0] = 'hb650b001;          // Fibinocci Series
  data[1] = 'h89048944;
  data[2] = 'h048b048b;
  data[3] = 'h5c0401d8;
  data[4] = 'h89440882;
  data[5] = 'hc60483fe;
  data[6] = 'h64007ced;
  data[7] = 'hb8adde89;
  data[8] = 'h440c9090;
  #550;
//  for (count=20;count<30;count=count+1)
//    $display("$%h : %h",count*4,data[count]);
end

endmodule
```

```
module i486 (aout, dout, din, bcdef, blast, brdy, ads,
        ain, rdy, a20m, hold, hlda, boff, breq, bsize,
        dpin, dpout, parity, ahold, eads, interrupt, nmi, reset, clk);
output [31:2] aout;
output[31:0] dout;
input [31:0] din;
output [4:0] bcdef;
output blast, ads, hlda, breq, parity;
input brdy, rdy, a20m, hold, boff, ahold, eads, interrupt, nmi, reset, clk;
input [31:4] ain;
input [1:0] bsize;
input [3:0] dpin;
output [3:0] dpout;
integer cycles;

reg [15:0] ax, cx, dx, bx, sp, bp, si, di;     // General Purpose Regs
reg [15:0] ds, cs, ss, es;                     // Segmentation registers
reg free;
wire [79:0] instructions;
wire [15:0] bus1, bus2, disp, sdisp;
wire [31:0] addrbus, baddr, databus, saddr, pfsaddr, linaddr;
wire [15:0] segregs, immed, flagreg;
wire [31:0] eip,ddata, segaddr, bdataout, cdout, cdin, bdatain, sftout, aluout;
wire [31:2] a;
wire [3:0] benable;
wire blast, brdy, ads, get;

paging PG1 (pfsaddr, linaddr, pxlate, cs, clk, ppage);
address A1 (instructions[2], addrcnt, databus, eip, clk);
prefetch PF1 (ddav, dnext, ddata, pfsaddr, linaddr, pxlate, ppage, get,
        pfflush, eip, free, bdav, bread, bdataout, baddr, clk);
pchk P1 (dpin, dpout, parity, din, dout, benable, bsize, bcdef, ads, brdy,
      aout, clk);
biu B1 (ads, rdy, a20m, bcdef, hold, hlda, boff, breq, brdy, blast,
     bsize, benable, aout, ain, dout, din, clk,
     cdout, cdin, a, cdav, cdread, cdwrite, miss);
cache C1 (cdout, cdin, a, cdav, cdread, cdwrite, miss, ahold, eads, ain,
      baddr, bdatain, bdataout, bread, bdav, bwrite, cflush, clk);
segmentation SEG1 (saddr, addrbus, instructions[31],segregs, clk, page, sdisp);
decode D1 (ddata, ddav, dnext, pfflush, free, cflush, addrcnt,
       immed, interrupt, nmi, reset, instructions, get,
       bread, bwrite, bdav, disp, bus2, flagreg, clk);
```

```
flag F1 (setflags, databus, flagreg, instructions[4], fdone);
shifter S1 (bus1, sftout, instructions[14], instructions[15],
        instructions[20:16], clk, setflags, fdone);
alu ALU1 (bus1, bus2, aluout, instructions[29:22], clk, setflags,
        instructions[11:9], fdone);
tristate T3 (flagreg, bus2, instructions[0]);
tristate T4 (addrbus, baddr, instructions[5]);
tristate T5 (databus, bdatain, instructions[6]);
tristate T6 (bdataout, bus1, instructions[7]);
tristate T7 (bdataout, bus2, instructions[8]);
tristate T8 (eip, bus1, instructions[3]);
tristate T9 (sftout, databus, instructions[13]);
tristate T10 (aluout, databus, instructions[21]);
tristate T11 (databus, saddr, instructions[30]);
tristate T12 (ds, segregs, instructions[32]);
tristate T13 (cs, segregs, instructions[33]);
tristate T14 (ss, segregs, instructions[34]);
tristate T15 (es, segregs, instructions[35]);
tristate T16 (ax, bus1, instructions[48]);
tristate T17 (cx, bus1, instructions[49]);
tristate T18 (dx, bus1, instructions[50]);
tristate T19 (bx, bus1, instructions[51]);
tristate T20 (sp, bus1, instructions[52]);
tristate T21 (bp, bus1, instructions[53]);
tristate T22 (si, bus1, instructions[54]);
tristate T23 (di, bus1, instructions[55]);
tristate T24 (ax, bus2, instructions[56]);
tristate T25 (cx, bus2, instructions[57]);
tristate T26 (dx, bus2, instructions[58]);
tristate T27 (bx, bus2, instructions[59]);
tristate T28 (sp, bus2, instructions[60]);
tristate T29 (bp, bus2, instructions[61]);
tristate T30 (si, bus2, instructions[62]);
tristate T31 (di, bus2, instructions[63]);
tristate T32 (disp, sdisp, instructions[1]);
tristate T33 (immed, bus2, instructions[12]);
tristate T34 (ds, bus2, instructions[64]);
tristate T35 (cs, bus2, instructions[65]);
tristate T36 (ss, bus2, instructions[66]);
tristate T37 (es, bus2, instructions[67]);

always @(posedge clk) begin            // register loads
```

```
 if (instructions[40]) ax= databus;
 if (instructions[41]) cx= databus;
 if (instructions[42]) dx= databus;
 if (instructions[43]) bx= databus;
 if (instructions[44]) sp= databus;
 if (instructions[45]) bp= databus;
 if (instructions[46]) si= databus;
 if (instructions[47]) di= databus;
 if (instructions[36]) ds= databus;
 if (instructions[37]) cs= databus;
 if (instructions[38]) ss= databus;
 if (instructions[39]) es= databus;
end

initial begin
 free = 1;                          // memory is initially free
 cycles=0;
 ax=0;
 cx=0;
 dx=0;
 bx=0;
 sp=0;
 bp=0;
 si=0;
 di=0;
 ds=0;
 cs=0;
 ss=0;
 es=0;
end

always @(get) begin                 // cache usage semaphore
 if (get) free = 0;
 else free = 1;
end

always @(posedge clk) begin                  // debugging displays
 cycles = cycles + 1;
// $display ("ax: %h, bx: %h, cx: %h, dx: %h, si: %h, sp: %h, ds: %h, di:
%h",ax,bx,cx,dx,si,sp,ds,di);
// $display ("cycles: %h", cycles);
end
```

endmodule

Biu.v

```
module biu(ads ,rdy ,a20m ,bcdef,hold,hlda,boff ,breq,brdy ,blast ,
        bsize ,benable ,aout,ain,dout,din,clk,
        cdout,cdin,a,cdav,cdread,cdwrite,miss);
input [31:0] cdin, din;
output [31:0] cdout, dout;
input rdy , a20m , hold, boff , brdy , clk, cdread, cdwrite, miss;
output cdav, ads , hlda, breq, blast ;
output [4:0] bcdef ;
input [1:0] bsize ;
output [3:0] benable ;
output [31:2] aout;
input [31:2] a;
input [31:4] ain;
reg [31:0] cdout, dout;
reg cdav, ads , hlda, breq, blast;
reg [4:0] bcdef ;
reg [3:0] benable ;
reg [31:2] aout;

reg [31:0] addr [3:0];          //write buffers
reg [31:0] data [3:0];          //write buffers
integer empty, status;          //pointer to buffers
integer count;

initial begin
  empty = 'b1111;     //all buffers empty
  status = 0;   //no cache misses in buffers
  benable  = 0;         //defaults to all bytes on 32 bit bus active
  bcdef  = 31; //defaults to data write state
  cdav = 1;
  ads = 1;
  blast = 1;
  breq = 0;
end

always @(clk) begin
  if (hold || !boff )
   begin @(posedge clk)
   $display("bus is floated");
   aout = 30'bz;        //floats addresses
   dout = 32'bz;        //floats datalines
   if (hold) hlda = 1;
```

```
  end
 else begin
 hlda = 0;
 if (!cdread)
  begin
  if (status)  //cache misses exist in buffers
   begin
   bcdef = 31;       //memory write
   for (count=0; count<4; count=count+1)
    begin
    if (!empty[count])       //specified buffer not empty
     begin
     if (bsize  == 3)        //32-bit bus
      begin
      benable  = 0;          //all 4-bytes enabled
      @(posedge clk);
      aout = addr[count];
      dout = data[count];
      ads =0;
      breq = !ads ;
      bcdef[4] = 0;          //plock
      if (count == 3 || (count<3 && empty[count+1]==1))  blast =0;
      @(negedge brdy );
      end
     else if (bsize [0] == 0)        //8-bit data bus
      begin
      $display("8-bit bus burst writes not supported");
      end
     else     //16-bit data bus
      begin
      $display("16-bit bus burst writes not supported");
      end
     end
    end
   @(posedge clk) ads =1;
   breq = !ads ;
   bcdef[4] = 1;     //plock
   @(posedge brdy ) blast =1;
   empty=15;         //all buffers empty now
   status = 0;       //no cache misses in buffers
   bcdef = 27;       //memory read
   for (count=0; count<4; count=count+1)
```

```
       begin
       if (bsize  == 3) // 32-bit bus
        begin
        benable  = 0;  //all 4 bytes enabled
        @(posedge clk);
        aout = a;
        ads  = 0;
        breq = !ads ;
        bcdef[4] = 0;  //plock
        if (count == 3)  blast =0;
        @(negedge brdy );
        cdout=din;
        cdav=0;
        @(posedge cdread);
        cdav=1;
        if (count < 3) @(negedge cdread);
        end
       else if (bsize  == 0)     //8-bit bus
        begin
        $display("8-bit bus not supported at this time");
        end
       else       //16-bit bus
        begin
        $display("16-bit bus not supported at this time");
        end
       end
     @(posedge clk);
     ads =1;
     breq = !ads ;
     bcdef[4] = 1;     //plock
     @(posedge brdy )  blast  = 1;
     end
    else //no cache misses in buffer
     begin
     bcdef = 27;        //memory read
     for (count=0; count<4; count=count+1)
      begin
      if (bsize  == 3) // 32-bit bus
       begin
       benable  = 0;  //all 4 bytes enabled
         @(posedge clk);
       aout = a;
```

```
        ads  = 0;
        breq = !ads ;
        bcdef[4] = 0;  //plock
          if (count == 3)  blast =0;
        @(negedge brdy);
        cdout=din;
        cdav=0;
        @(posedge cdread);
        cdav=1;
        if (count < 3) @(negedge cdread);
        end
      else if (bsize  == 0)     //8-bit bus
        begin
        $display("8-bit bus not supported at this time");
        end
      else        //16-bit bus
        begin
        $display("16-bit bus not supported at this time");
        end
      end
    @(posedge clk);
    ads =1;
    breq = !ads ;
    bcdef[4] = 1;      //plock
    @(posedge brdy )  blast  = 1;
    for (count=0;count<4;count=count+1)
      if (empty[count] != 1) //makes buffers cache misses
        status[count] = 1;
    end
  end
else if (!cdwrite)
  begin
  if (!empty) //buffers full
    begin
    bcdef = 31;        //memory write
    for (count=0; count<4; count=count+1)
      begin
      if (!empty[count])      //specified buffer not empty
        begin
        if (bsize  == 3)        //32-bit bus
          begin
          benable  = 0;         //all 4-bytes enabled
```

```
        @(posedge clk);
        aout = addr[count];
        dout = data[count];
        ads =0;
        breq = !ads ;
        bcdef[4] = 0;        //plock
        if (count == 3 || (count<3 && empty[count+1]==1))  blast =0;
        @(negedge brdy );
        end
      else if (bsize [0] == 0)        //8-bit data bus
        begin
        $display("8-bit bus burst writes not supported");
        end
      else     //16-bit data bus
        begin
        $display("16-bit bus burst writes not supported");
        end
      end
    end
   @(posedge clk) ads =1;
   breq = !ads ;
   bcdef[4] = 1;     //plock
   @(posedge brdy ) blast =1;
   empty=15;        //all buffers empty now
   status = 0;
   end
  for (count=0; count<4; count=count+1)        //write to buffers
   begin
   if (empty[count] == 1)
     begin
     if (miss)  status[count] = 1;   //cache miss
     data[count] = cdin;
     addr[count] = 0;  //set all bits to zero
     addr[count] = a;
     cdav=0;
     @(posedge cdwrite) cdav=1;
       empty[count] = 0;
     count = 4;        //skips out of loop
     end
   end
  end
 else if (empty<15)  //buffers not all empty
```

```
    begin
    $display("idle write from buffers to memory");
    bcdef = 31;          //memory write
    for (count=0; count<4; count=count+1)
     begin
     if (!empty[count])        //specified buffer not empty
       begin
       if (bsize  == 3) //32-bit bus
         begin
         benable  = 0;  //all 4-bytes enabled
         @(posedge clk);
         aout = addr[count];
         dout = data[count];
         ads =0;
         breq = !ads ;
         bcdef[4] = 0;  //plock
         if (count == 3 || (count<3 && empty[count+1]==1))  blast =0;
         @(negedge brdy );
         end
       else if (bsize [0] == 0)//8-bit data bus
         begin
         $display("8-bit bus burst writes not supported");
         end
       else       //16-bit data bus
         begin
         $display("16-bit bus burst writes not supported");
         end
       end
     end
    @(posedge clk) ads =1;
    breq = !ads ;
    bcdef[4] = 1;       //plock
    @(posedge brdy ) blast =1;
    empty=15; //all buffers empty now
    status = 0;   //so obviously no cache misses
    end
  end
  end

task bread;
input cdread;
output cdav;
```

```
output [31:0] cdout;
input [31:2] a;
output ads;
input brdy;
output blast;
output [4:0] bcdef;
output [31:2] aout;
input [31:0] din;
output breq;
input [1:0] bsize;
input clk;
integer count, temp;
 begin
 bcdef = 27;  //memory read
 for (count=0; count<4; count=count+1)
  begin
  if (bsize  == 3)     // 32-bit bus
   begin
   benable  = 0;      //all 4 bytes enabled
   @(posedge clk);
   aout = a;
   ads  = 0;
   breq = !ads ;
   bcdef[4] = 0;      //plock
   if (count == 3)  blast =0;
   @(negedge brdy );
   cdout=din;
   cdav=0;
   @(posedge cdread);
   cdav=1;
   if (count < 3) @(negedge cdread);
   end
  else if (bsize  == 0)        //8-bit bus
   begin
   $display("8-bit bus not supported at this time");
   end
  else //16-bit bus
   begin
   $display("16-bit bus not supported at this time");
   end
  end
 @(posedge clk);
```

```
  ads =1;
  breq = !ads ;
  bcdef[4] = 1;          //plock
  @(posedge brdy )  blast  = 1;
  end
endtask

endmodule
```

Parity.v

```
module pchk (dpin,dpout,parity,datain,dataout,benable,bsize,bcdef,ads,brdy,
        aout,clk);
input [31:0] datain, dataout;
input [31:2] aout;
input [3:0] dpin, benable;
input [4:0] bcdef;
input [1:0] bsize;
input clk, ads, brdy;
output [3:0] dpout;
output parity;
reg parity;
reg [3:0] dpout;
reg [3:0] temp;
reg [31:0] data;
integer read, write, bus;

always @(negedge brdy or negedge ads or aout) begin
parity = 1;
temp = 0;
read = 0;
write = 0;
if ((bcdef[0] == 1 || (bcdef[0] == 0 && bcdef[1] == 1))  && bcdef[2] == 0)
 begin
 read = 1;
 data=datain;
 end
if (bcdef[2] == 1 && bcdef[1] == 1)
 begin
 write = 1;
 data = dataout;
 end
parity = 1;
if (write || read)
 begin
 if (bsize == 3)       //32-bit bus
  begin
  if (!benable[3])     //4th byte active
   begin
   if (check(data[31:24], dpin[3], write))          //odd
    temp[3] = 1;
   end
  if (!benable[2])     //3rd byte active
```

```
    begin
    if (check(data[23:16], dpin[2], write))         //odd
      temp[2] = 1;
    end
  end
 if (bsize[0] && !benable[1])       //at least 16 bits wide, 2nd byte enabled
  begin
  if (check(data[15:8], dpin[1], write))   //odd
   temp[1] = 1;
  end
 if (!benable[0])       //1st byte active
  begin
  if (check(data[7:0], dpin[0], write))      //odd
   temp[0] = 1;
  end
 if (read)       //outputs parity on next cycle for reads
  begin
  @(posedge clk) parity = !(temp[0] || temp[1] || temp[2] || temp[3]);
  @(posedge clk);
  end
 else
  begin         //output data parity immediately on writes
  dpout[0] = temp[0];
  dpout[1] = temp[1];
  dpout[2] = temp[2];
  dpout[3] = temp[3];
  @(posedge ads or aout);
  end
 end
end

function check;        //checks for odd parity
input [7:0] data;
input dpin, write;
integer count, result;
begin
 result = data[0];
 if (!write && dpin) result = result + 1;  //check parity bit for reads
 check = result[0];
end
endfunction
```

```
endmodule
```

Cache.v

```
module cache (cdout,cdin,a,cdav,cdread,cdwrite,miss,ahold,eads,aextern,
          baddr, bdatain, bdataout, bread, bdav, bwrite, flush, clk);
input [31:0] cdout, baddr, bdatain;
output [31:0] cdin, bdataout;
output [31:2] a;
input [31:4] aextern;
output cdread, cdwrite, miss, bdav;
input clk, cdav, bread, bwrite, flush, ahold, eads;
reg [31:0] cdin, bdataout;
reg [31:2] a;
reg cdread, cdwrite, miss, bdav;
reg [20:0] tag0 [127:0];
reg [20:0] tag1 [127:0];
reg [20:0] tag2 [127:0];
reg [20:0] tag3 [127:0];
reg [31:0] data0 [2047:0];
reg [31:0] data1 [2047:0];
reg [31:0] data2 [2047:0];
reg [31:0] data3 [2047:0];
reg [3:0] valid [127:0];
reg [2:0] lru [127:0];
integer count, way, tempaddr, temp;

initial
  begin
  cdread = 1;
  cdwrite = 1;
  bdav = 1;
  miss = 0;     //default to not cache miss
  for (count=0; count<128; count=count+1)
    valid[count] = 0;  //all cache line start off invalid
  end

always @(clk)
  begin
  if (!flush)
    begin
    $display("cache flushed");
    for (count=0; count<128; count=count+1)
      valid[count] = 0;    //flush all
    end
  else if (ahold && !eads)   //external cache invalidation
```

```
  begin
  $display("external cache invalidation cycle");
  tempaddr = aextern;
  temp = 4;
if (tag0[tempaddr[10:4]]==tempaddr[31:11] && (valid[tempaddr[10:4]] & 'b0001))
   temp = 0;
if (tag1[tempaddr[10:4]]==tempaddr[31:11] && (valid[tempaddr[10:4]] & 'b0010))
   temp = 1;
if (tag2[tempaddr[10:4]]==tempaddr[31:11] && (valid[tempaddr[10:4]] & 'b0100))
   temp = 2;
if (tag3[tempaddr[10:4]]==tempaddr[31:11] && (valid[tempaddr[10:4]] & 'b1000))
   temp = 3;
  way = temp;
  if (way == 0)
   valid[tempaddr[10:4]]=valid[tempaddr[10:4]]&'b1110;//make way zero invalid
  if (way == 1)
   valid[tempaddr[10:4]]=valid[tempaddr[10:4]]&'b1101;//make way one invalid
  if (way == 2)
   valid[tempaddr[10:4]]=valid[tempaddr[10:4]]&'b1011;//make way two invalid
  if (way == 3)
  valid[tempaddr[10:4]]=valid[tempaddr[10:4]]&'b0111;//make way three invalid
  end
 else if (!bread)      //bus read from cache
  begin
  temp = 4;
  if (tag0[baddr[10:4]] == baddr[31:11] && (valid[baddr[10:4]] & 'b0001))
   temp = 0;
  if (tag1[baddr[10:4]] == baddr[31:11] && (valid[baddr[10:4]] & 'b0010))
   temp = 1;
  if (tag2[baddr[10:4]] == baddr[31:11] && (valid[baddr[10:4]] & 'b0100))
   temp = 2;
  if (tag3[baddr[10:4]] == baddr[31:11] && (valid[baddr[10:4]] & 'b1000))
   temp = 3;
  way = temp;
  if (way > 3)        //cache miss
   begin
   miss = 1; //read from memory to cache
   temp = 4;
   if (valid[baddr[10:4]] == 15);
    begin    //all ways valid
    if (!(lru[baddr[10:4]] & 'b001))       //bit 0 zero
     begin
```

```
      if (!(lru[baddr[10:4]] & 'b010))      //bit 1 zero
       begin
       lru[baddr[10:4]] = lru[baddr[10:4]] | 'b001;      //bit 0 one
       lru[baddr[10:4]] = lru[baddr[10:4]] | 'b010;      //bit 1 one
       temp = 0;
       end
      else
       begin
       lru[baddr[10:4]] = lru[baddr[10:4]] | 'b001;      //bit 0 one
       lru[baddr[10:4]] = lru[baddr[10:4]] & 'b101;      //bit 1 zero
       temp = 1;
       end
      end
    else      //bit 0 one
      begin
      if (!(lru[baddr[10:4]] & 'b100))      //bit 2 zero
       begin
       lru[baddr[10:4]] = lru[baddr[10:4]] & 'b110;
       lru[baddr[10:4]] = lru[baddr[10:4]] | 'b100;
       temp = 2;
       end
      else
       begin
       lru[baddr[10:4]] = lru[baddr[10:4]] & 'b110;
       lru[baddr[10:4]] = lru[baddr[10:4]] & 'b011;
       temp = 3;
       end
      end      //else lru[set][1]  == 1
    end      //valid analysis
  way = temp;
  if (way > 3)      //invalid line exists
    begin
    if (!(valid[baddr[10:4]] & 'b0001))
      temp = 0;      //way zero invalid
    else if (!(valid[baddr[10:4]] & 'b0010))
      temp = 1;
    else if (!(valid[baddr[10:4]] & 'b0100))
      temp = 2;
    else if (!(valid[baddr[10:4]] & 'b1000))
      temp = 3;
    way = temp;      //find out which way
     end
```

```
  tempaddr = baddr & 'b11111111111111111111111111110000;
  for (count=0; count<4; count=count+1)
   begin    //request new data from memory
   a = tempaddr;
   @(clk) cdread = 0;
   if (way == 0)
    @(negedge cdav) data0[tempaddr[10:2]] = cdout;
   if (way == 1)
    @(negedge cdav) data1[tempaddr[10:2]] = cdout;
   if (way == 2)
    @(negedge cdav) data2[tempaddr[10:2]] = cdout;
   if (way == 3)
    @(negedge cdav) data3[tempaddr[10:2]] = cdout;
   @(clk) cdread = 1;
   tempaddr=tempaddr+4;
   end      //burst read from memory
  if (way==0) //make the new way valid and update tag
   begin
   valid[baddr[10:4]] = valid[baddr[10:4]] | 'b0001;
     tag0[baddr[10:4]] = baddr[31:11];
     end
  else if (way ==1)
   begin
   valid[baddr[10:4]] = valid[baddr[10:4]] | 'b0010;
     tag1[baddr[10:4]] = baddr[31:11];
     end
  else if (way ==2)
   begin
   valid[baddr[10:4]] = valid[baddr[10:4]] | 'b0100;
     tag2[baddr[10:4]] = baddr[31:11];
     end
  else if (way ==3)
   begin
   valid[baddr[10:4]] = valid[baddr[10:4]] | 'b1000;
     tag3[baddr[10:4]] = baddr[31:11];
     end
  end       //cache miss read from memory
 if (way == 0)      // return cache data to bus
  bdataout = data0[baddr[10:2]];
 if (way == 1)
  bdataout = data1[baddr[10:2]];
 if (way == 2)
```

```
    bdataout = data2[baddr[10:2]];
   if (way == 3)
    bdataout = data3[baddr[10:2]];
   bdav = 0;
$display("bread: baddr:%h, bdataout:%h",baddr, bdataout);
   @(posedge bread) bdav = 1;
   miss = 0;
   end //bread function
  else if (!bwrite)      //bus write to cache
   begin
$display("bwrite: baddr:%h, bdatain:%h",baddr, bdatain);
   temp = 4;
   if (tag0[baddr[10:4]] == baddr[31:11] && (valid[baddr[10:4]] & 'b0001))
    temp = 0;
   if (tag1[baddr[10:4]] == baddr[31:11] && (valid[baddr[10:4]] & 'b0010))
    temp = 1;
   if (tag2[baddr[10:4]] == baddr[31:11] && (valid[baddr[10:4]] & 'b0100))
    temp = 2;
   if (tag3[baddr[10:4]] == baddr[31:11] && (valid[baddr[10:4]] & 'b1000))
    temp = 3;
   way = temp;
   if (way == 0)      //update cache
    data0[baddr[10:2]] = bdatain;
   if (way == 1)
    data1[baddr[10:2]] = bdatain;
   if (way == 2)
    data2[baddr[10:2]] = bdatain;
   if (way == 3)
    data3[baddr[10:2]] = bdatain;
   if (way > 3)
    miss = 1;
   a = baddr;
   cdin = bdatain;     //write through directly to memory
   cdwrite = 0;
   @(negedge cdav) cdwrite  = 1;
   bdav = 0;
   @(posedge bwrite) bdav = 1;
   miss = 0;
   end //write to biu
  end   //always

endmodule
```

Cache.v

Prefetch.v

```
module prefetch (ddav, dnext, ddata, segaddr, linaddr, xlate, page, get,
            flush, pc, free, bdav, bread, bdataout, baddr, clk);
output ddav, xlate, get;
input dnext, bdav, free, flush, page, clk;
output [31:0] ddata, segaddr;
input [31:0] linaddr, bdataout, pc;
inout [31:0] baddr;
inout bread;
reg bread_reg, get;
wire bread=bread_reg;
reg [31:0] baddr_reg;
wire [31:0] baddr = baddr_reg;
reg [31:0] ddata, segaddr;
reg ddav, xlate;
reg [31:0] data [7:0];
integer count, empty, current, pcreg, base, fault, fetch, idle, iflush;

initial begin
 idle = 0;                    // don't force prefetch
 baddr_reg = 32'bz;           //high impedence states on bus
 bread_reg = 'bz;
 iflush = 0;                  // internal flush flag cleared
 pcreg = pc;
 fault = 0;                   //page fault indicator
 ddav = 1;
 get = 'bz;
 xlate = 1;
 current = 0;                 //currently send buffer set to #0
 empty = 255;                         //all buffers empty
 end

always @(posedge page) begin
 empty[count] = 1;
 fault = 1;
 end

always @(clk) begin
 fetch = 0;                           // assumes no idle prefetch
 if (!dnext)
  begin
  if (get && empty[current]) @(posedge fetch); // wait for prefetch to finish
//$display("empty:%b, current:%h",empty,current);
```

```
   if (empty[current])                         // current buffer empty
    begin
    current = 0;
    if (empty[current])                   //all buffers empty
     begin
     idle = 100;                          // force idle prefetch
     @(posedge fetch);                    //wait for idle prefetch
     end
    end
   ddata = data[current];                 //send current byte over
   ddav = 0;                              //data available
   @(posedge dnext) ddav = 1;
   empty[current] = 1;                         //make sent buffer empty
   current = current + 1;
   if (current == 8) current = 0;
   end
end

always @(posedge clk) begin
  if (flush || iflush)
   begin
   idle = 0;
   empty = 255;                           //all buffers flushed
   iflush = 0;                            // internal flush flag cleared
   current = 0;
   pcreg = pc;
   fault = 0;
   ddav=0;
   get = 'bz;                             // release cache lock
   @(clk) ddav = 1;
$display("prefetch buffers flushed, pcreg:%h",pcreg);
   end
  else if ((idle>3)&&free&&(empty[3:0]=='b1111 || empty[7:4]=='b1111)&&!fault)
   begin
$display("idle prefetch; idle = %d",idle);
   get = 1;
   if (empty[3:0] == 'b1111) base = 0;
   else if (empty[7:4] == 'b1111) base = 4;
   if (!flush)                            // exit if flush
    begin
    for (count=base; count<base+4; count=count+1)
     begin
```

```
        segaddr = pcreg+((count-base)*4);
        xlate = 0;
        @(clk) baddr_reg = linaddr;
        xlate = 1;
        baddr_reg = linaddr;
        if (flush)
          begin
          iflush = 1;
          count = base+3;
          end
        bread_reg = 0;
        @(negedge bdav) data[count] = bdataout;
        bread_reg = 1;
        end
      end
    if (base==0)
      empty[3:0] = 'b0000;          //lower 4 dwords no longer empty
    else if (base==4)
      empty[7:4] = 'b0000;          //last 4 dword buffers filled
    pcreg = 4+segaddr;
    baddr_reg = 32'bz;                    //reset bus to high impedence
    bread_reg = 'bz;
    get = 'bz;
    fetch = 1;
    idle = -1;
//$display("exit idle prefetch");
    end                             //end automatic prefetch
  if (free &&(empty[3:0]=='b1111 || empty[7:4]=='b1111)&& !fault)
    idle = idle + 1;                // allows idle prefetch next clk
  else idle = 0;
//$display("idle:%h",idle);
  end                               //end always

endmodule
```

Decode.v

```
module decode (ddata, ddav, dnext, pfflush, free, cflush, addrcnt,
        immed, interrupt, nmi, reset, instructions, get,
        bread, bwrite, bdav, disp, bus2, flag, clk);
`define REG 1
`define MEM 2
`define EIP 3
`define IMM 4
`define SREG 5
`define NADA 0

output [79:0] instructions;
output dnext, pfflush, cflush, addrcnt, get;
output [15:0] immed, disp;
input [15:0] bus2, flag;
input ddav, clk, interrupt, nmi, reset, bdav, free;
input [31:0] ddata;          // opcodes from prefetch
inout bread, bwrite;
reg dnext, pfflush, cflush, d, w, s, update, addrcnt;
reg bread_reg, bwrite_reg, check, memsave, get;
wire bread = bread_reg;
wire bwrite = bwrite_reg;
reg [15:0] immed, disp;
reg [79:0] instructions;
reg [7:0] temp [3:0];
reg [7:0] opcode, modem;
reg [8:0] func;
reg [1:0] num;
integer count, dest, src, resume, rem, getimm, cmp, cc, jump, imul, idiv;
integer autodisp;

initial begin
 check = 0;
 imul = 0;                  // not multiplying now
 idiv = 0;
 autodisp = 0;                      // will not allow disp through to seg
 cmp = 0;                   // not compare command
 getimm = 0;
 resume = 0;                // no jumps in progress
 get = 'bz;                 // not using memory
 memsave = 0;
 disp = 0;                  // no displacement
 pfflush = 0;               // positive assertion
```

```
  cflush = 1;                    // negative assertion
  update = 0;                    // initial reset will provide prefetch update
  bread_reg = 'bz;                     // negatiove assertion
  bwrite_reg = 'bz;                    // cache requests
  instructions=0;
end

always @(posedge update or posedge pfflush)   // request stuff from prefetch
  begin
$display("entered update, pfflush = %b",pfflush);
  update = 1;                    // just to make sure of it
  if (pfflush) @(negedge ddav); // wait for pfflush to finish
  pfflush = 0;
  @(clk) dnext = 0;
  @(negedge ddav);
  temp[0] = ddata[31:24];   // temp = top 8 bits in register
  temp[1] = ddata[23:16];
  temp[2] = ddata[15:8];
  temp[3] = ddata[7:0];
  dnext = 1;                     // got new 32 bit instr from prefetch
  if (resume) count = rem;
  else count = 0;
  if (!resume)
   begin
   addrcnt = 1;
   @(posedge clk) addrcnt=0;       // increments EIP
   end
  resume = 0;
  update = 0;
$display("exit update");
end

always @(posedge clk) begin
  func=1;                        // ALU return op1 command
  imul = 0;
  idiv = 0;
  opcode = 0;
  cmp = 0;
  d=0;
  s=0;
  w=0;
  modem=0;
```

```
  immed = 0;
  bread_reg='bz;
  bwrite_reg='bz;
  getimm = 0;
  disp = 0;                    // zero memory displacement
  get = 'bz;                   // memory bus is free
  instructions = 0;
  autodisp = 0;
  dest=`NADA;
  src=`NADA;
  if (update) @(negedge update);
  next(opcode, count, count, temp[0], temp[1], temp[2], temp[3], update);
$display("opcode:%h",opcode);
  if (reset)                   // nmi and interrupt not supported
   begin
   cflush = 0;
   @(posedge clk);
   pfflush = 1;
   cflush = 1;
   @(negedge update);
   @(posedge clk);
   immed=0;
   instructions[2]=1;          // EIP load
   instructions[12]=1;               // immed
   instructions[21]=1;               // ALUout
   instructions[30]=1;               // segin
   instructions[31]=1;               // seg xlate
   instructions[33]=1;               // CS out
   instructions[29:22]=2;      // ALU out = op2
$display ("reset done");
   end
  else if (opcode[7:2] == 0)         // ADD reg/mem, reg/mem
   begin
   d=opcode[1];                // direction of data flow
   w=opcode[0];                // w parameter indicates width of operands
   if (update) @(negedge update);
   next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
   if (d==0 && modem[7:6] != 'b11)
    begin
    dest=`MEM;
    src=`REG;
    end
```

```
  else if (modem[7:6] != 'b11)
   begin
   dest=`REG;
   src=`MEM;
   end
  else
   begin
   dest=`REG;
   src=`REG;
   end
  if (dest==`MEM || src==`MEM)
   begin
   check = 1;                    // check for memory access
   @(negedge check);                    // calculate addr if so
   end
  func = 0;                 // ALU ADD
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;  //wait for cache to get operand
   end
  if (dest==`MEM)
   begin
   memsave=1;                         // memory save if needed
   @(negedge memsave);
   end
  end
 else if (opcode[7:2]=='b100000)          // add reg/mem, immed
  begin
  src=`IMM;                             // immed on src
  w=opcode[0];
  if (update) @(negedge update);
  next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
  if (modem[7:6]!='b11) dest=`MEM;
  else dest=`REG;
  if (dest==`MEM || src==`MEM)
   begin
   getimm = 1;
   check =1;                            // check for memory access
   @(negedge check);
   end
```

```
  if (!getimm)
   begin
   immed=0;                          // get immediate values
   if (update) @(negedge update);
   next(immed[7:0],count,count,temp[0],temp[1],temp[2],temp[3],update);
   if(w==1)                    // 16 bit operands
    begin
    if (update) @(negedge update);
    next(immed[15:8],count,count,temp[0],temp[1],temp[2],temp[3],update);
    end
   else signextend(8,16,immed,immed); // sign extend 8 to 16 bits
   end
  func=0;                              // alu add function
  if (modem[5:3] == 'b101)                    // this is actually sub
   func = 4;
  if (modem[5:3] == 'b100)                    // this is actually and
   func = 5;
  if (modem[5:3] == 'b001)                    // this is actually or
   func = 6;
  if (modem[5:3] == 'b110)                    // this is actually xor
   func = 8;
  if (modem[5:3] == 'b111)                    // this is actually cmp
   begin
   cmp = 1;
   func = 4;                           // alu subtract
   end
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;   //wait for cache to get operand
   end
  if (dest==`MEM  && !cmp)
   begin
   memsave=1;                         // memory save if needed
   @(negedge memsave);
   end
  end
 else if (opcode[7:1]=='b0000010)             // add accum, immed
  begin
  dest=`REG;
  src=`IMM;
```

```
 w=opcode[0];
 modem=0;                        // reg=AX
 immed=0;
 if (update) @(negedge update);
 next(immed[7:0],count,count,temp[0],temp[1],temp[2],temp[3],update);
 if(w==1)                        // 16 bit operands
  begin
  if (update) @(negedge update);
  next(immed[15:8],count,count,temp[0],temp[1],temp[2],temp[3],update);
  end
 else signextend(8,16,immed,immed);   // sign extend 8 to 16 bits
 func=0;
 fetch(instructions, func, d, modem, bread_reg, dest, src);
 if (get)
  begin
  @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
  bread_reg='b1;   //wait for cache to get operand
  end
 end
else if (opcode[7:2]=='b100010) // mov reg/mem, reg/mem
 begin
 d=opcode[1];
 w=opcode[0];
 if (update) @(negedge update);
 next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
 if (d==0 && modem[7:6] != 'b11)
  begin
  src=`REG;
  dest=`MEM;
  end
 else if (modem[7:6] != 'b11)
  begin
  src=`MEM;
  dest=`REG;
  end
 else
  begin
  dest=`REG;
  src=`REG;
  end
 if (dest==`MEM || src==`MEM)
  begin
```

```
    check=1;
    @(negedge check);
    end
   func=2;                          // alu returns op2
   fetch(instructions, func, d, modem, bread_reg, dest, src);
   if (get)
    begin
    if (src == `MEM) instructions[40+modem[5:3]] = 1;
    @(negedge bdav)
    bread_reg='b1;  //wait for cache to get operand
    end
   if (dest==`MEM)
    begin
    memsave=1;                          // memory save if needed
    @(negedge memsave);
    end
   end
  else if (opcode[7:4]=='b1011)          // mov reg,immed
   begin
   d=1;                             // dest is "reg" in modem
   w=opcode[3];
   dest=`REG;
   src=`IMM;
   modem[5:3]=opcode[2:0];
   immed = 0;
   if (update) @(negedge update);
   next(immed[7:0],count,count,temp[0],temp[1],temp[2],temp[3],update);
   if (w==1)
    begin
    if (update) @(negedge update);
    next(immed[15:8],count,count,temp[0],temp[1],temp[2],temp[3],update);
    end
   else signextend(8,16,immed,immed);  // sign extend 8 to 16 bits
   func = 2;                        // alu puts op2 on bus
   fetch(instructions, func, d, modem, bread_reg, dest, src);
   if (get)
    begin
    @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
    bread_reg='b1;  //wait for cache to get operand
    end
   end
  else if (opcode[7:1] == 'b1100011)      //mov reg/mem,immed
```

```
  begin
  w=opcode[0];
  src=`IMM;
  if (update) @(negedge update);
  next(modem,count,count,temp[0],temp[1],temp[2],temp[3],update);
  if (modem[7:6] != 'b11)  dest=`MEM;
  else dest=`REG;
  if (dest==`MEM || src==`MEM)
   begin
   getimm = 1;
   check=1;
   @(negedge check);
   end
  if (!getimm)
   begin
   immed = 0;
   if (update) @(negedge update);
   next(immed[7:0],count,count,temp[0],temp[1],temp[2],temp[3],update);
   if (w==1)
    begin
    if (update) @(negedge update);
    next(immed[15:8],count,count,temp[0],temp[1],temp[2],temp[3],update);
    end
   else signextend(8,16,immed,immed); // sign extend 8 to 16 bits
   end
  func = 2;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;   //wait for cache to get operand
   end
  if (dest==`MEM)
   begin
   memsave=1;                          // memory save if needed
   @(negedge memsave);
   end
  end
else if (opcode[7:1] == 'b1010000)                // mov accum,mem
 begin
 w=opcode[0];
 d=1;                                          // mem -> reg
```

```
  dest = `REG;
  src = `MEM;
  modem = 0;
  if (dest==`MEM || src==`MEM)
   begin
   check=1;
   @(negedge check);
   end
  func = 2;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;  //wait for cache to get operand
   end
  end
 else if (opcode[7:1] == 'b1010001)              // mov mem,accum
  begin
  w=opcode[0];
  d=0;
  dest = `MEM;
  src = `REG;
  modem = 0;
  if (dest==`MEM || src==`MEM)
   begin
   check=1;
   @(negedge check);
   end
  func = 2;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;  //wait for cache to get operand
   end
  if (dest==`MEM)
   begin
   memsave=1;                        // memory save if needed
   @(negedge memsave);
   end
  end
 else if (opcode[7:0] == 'b10001110)             // mov sreg,reg/mem
```

```
  begin
  d = 1;
  dest = `SREG;
  if (update) @(negedge update);
  next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
  if (modem[7:6] != 'b11) src = `MEM;
  else src = `REG;
  if (dest==`MEM || src==`MEM)
   begin
   check=1;
   @(negedge check);
   end
  func = 2;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;  //wait for cache to get operand
   end
  end
else if (opcode[7:0] == 'b10001100)            // mov reg/mem, sreg
  begin
  src = `SREG;
  d = 0;
  if (update) @(negedge update);
  next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
  if (modem[7:6] != 'b11) dest = `MEM;
  else dest = `REG;
  if (dest==`MEM || src==`MEM)                 // calculates effective addr
   begin                                       // (includes getting disp)
   check=1;
   @(negedge check);
   end
  func = 2;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;                  //wait for cache to get operand
   end
  if (dest==`MEM)
   begin
```

```
    memsave=1;                          // memory save if needed
    @(negedge memsave);
    end
   end
  else if (opcode[7:0] == 'b11101011)             // jmp
   begin
   immed = 0;
   if (update) @(negedge update);
   next(immed[7:0], count, count, temp[0], temp[1], temp[2], temp[3], update);
   src = `IMM;
   dest = `EIP;
   modem = 0;
   d = 0;                         // modem and d are irrevelent here
   func = 0;                      // alu add
   signextend(8,16,immed,immed);
   if (update) @(negedge update);
   jmp(resume, immed, immed, count, rem);     // keeps addresses even
   fetch(instructions, func, d, modem, bread_reg, dest, src);
   @(posedge clk) instructions = 0;
   pfflush = 1;                   // flush the prefetch
   @(negedge ddav);
   end
  else if (opcode[7:0] == 'b11101001)             // jmp with 16 bit disp
   begin
   immed = 0;
   if (update) @(negedge update);
   next(immed[7:0], count, count, temp[0], temp[1], temp[2], temp[3], update);
   if (update) @(negedge update);
   next(immed[15:8], count,count, temp[0], temp[1], temp[2], temp[3], update);
   src = `IMM;
   dest = `EIP;
   modem = 0;
   func = 0;
   if (update) @(negedge update);
   jmp(resume, immed, immed, count, rem);     // keeps addresses even
   fetch(instructions, func, d, modem, bread_reg, dest, src);
   @(posedge clk) instructions = 0;
   pfflush = 1;                   // flush the prefetch
   @(negedge ddav);
   end
  else if (opcode[7:0] == 'b11111111)             // jmp reg/mem
   begin
```

```
 d = 0;
 dest = `EIP;
 if (update) @(negedge update);
 next(modem,count,count,temp[0],temp[1],temp[2],temp[3],update);
 if (modem[5:3] != 'b110) begin
 if (modem[7:6] == 'b11) src = `REG;
 else src = `MEM;
 if (src == `MEM || dest == `MEM)
  begin
  check = 1;
  @(negedge check);
  end
 func = 3;                                  // aluout = even addrs op1;
 if (update) @(negedge update);
 fetch(instructions, func, d, modem, bread_reg, dest, src);
 instructions[11:9] = count;
 if (get) @(negedge bdav) bread_reg = 'b1;
 @(clk) jmp (resume, immed, bus2, count, rem);       // even addrs
 get = 'bz;
 @(clk) instructions = 0;
 pfflush = 1;
 @(negedge ddav);
 end
 end
else if (opcode[7:2] == 'b001110)                   // CMP reg/mem, reg/mem
 begin
 cmp = 1;
 d = opcode[1];
 w = opcode[0];
 if (update) @(negedge update);
 next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
 if (d==0 && modem[7:6] != 'b11)
  begin
  dest = `MEM;
  src = `REG;
  end
 else
  begin
  dest=`REG;
  src=`REG;
  end
 if (dest==`MEM || src==`MEM)
```

```
    begin
    check = 1;                    // check for memory access
    @(negedge check);                   // calculate addr if so
    end
   func = 4;                // ALU SUB
   fetch(instructions, func, d, modem, bread_reg, dest, src);
   if (get)
    begin
    @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
    bread_reg='b1;  //wait for cache to get operand
    end
   end
  else if (opcode[7:1]=='b0011110)          // cmp accum, immed
   begin
   cmp = 1;
   dest=`REG;
   src=`IMM;
   w=opcode[0];
   modem=0;                   // reg=AX
   immed=0;
   if (update) @(negedge update);
   next(immed[7:0],count,count,temp[0],temp[1],temp[2],temp[3],update);
   if(w==1)                   // 16 bit operands
    begin
    if (update) @(negedge update);
    next(immed[15:8],count,count,temp[0],temp[1],temp[2],temp[3],update);
    end
   else signextend(8,16,immed,immed);  // sign extend 8 to 16 bits
   func=4;
   fetch(instructions, func, d, modem, bread_reg, dest, src);
   if (get)
    begin
    @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
    bread_reg='b1;  //wait for cache to get operand
    end
   end
  else if (opcode[7:4] == 'b0111 || opcode == 'b00001111)// conditional jumps
   begin
   if (opcode[7:0] == 'b00001111)        // near jump
    begin
    if (update) @(negedge update);
    next(opcode[7:0],count,count,temp[0],temp[1],temp[2],temp[3],update);
```

```
  if (opcode[7:0] == 'b10101111) imul = 1;
  end
 if (!imul) begin
 jump = 0;                          // default don't jump
 immed = 0;
 if (update) @(negedge update);
 next(immed[7:0], count, count, temp[0], temp[1], temp[2], temp[3], update);
 if (opcode[7:4] == 'b1000)              // near jump
  begin
  if (update) @(negedge update);
  next(immed[15:8],count,count,temp[0],temp[1],temp[2],temp[3],update);
  end
 else signextend(8,16,immed,immed);  // short jump
 src = `IMM;
 dest = `EIP;
 modem = 0;
 d = 0;
 func = 0;
 cc = opcode[3:0];
 if (cc == 'b0100) begin if (flag[6]) jump = 1; end
 if (cc == 'b1111) begin if (!flag[6] && !flag[7]) jump = 1; end
 if (cc == 'b1101) begin if (!flag[7]) jump = 1; end
 if (cc == 'b1100) begin if (flag[7]) jump = 1; end
 if (cc == 'b1110) begin if (!flag[7] || flag[6]) jump = 1; end
 if (cc == 'b0011) begin if (!flag[0]) jump = 1; end
 if (cc == 'b0101) begin if (!flag[6]) jump = 1; end
 if (cc == 'b1101) begin if (!flag[7]) jump = 1; end
 if (cc == 'b1111) begin if (!flag[6] && !flag[7]) jump = 1; end
 if (cc == 'b0001) begin if (!flag[11]) jump = 1; end
 if (cc == 'b1011) begin if (!flag[2]) jump = 1; end
 if (cc == 'b1001) begin if (!flag[7]) jump = 1; end
 if (cc == 'b0101) begin if (!flag[6]) jump = 1; end
 if (cc == 'b0000) begin if (flag[11]) jump = 1; end
 if (cc == 'b1010) begin if (flag[2]) jump = 1; end
 if (cc == 'b1011) begin if (!flag[2]) jump = 1; end
 if (cc == 'b1000) begin if (flag[7]) jump = 1; end
 if (cc == 'b0100) begin if (flag[6]) jump = 1; end
 if (jump)
  begin
  if (update) @(negedge update);
  jmp(resume, immed, immed, count, rem);
  fetch(instructions, func, d, modem, bread_reg, dest, src);
```

```
  @(posedge clk); instructions = 0;
  pfflush = 1;
  @(negedge ddav);
  end
 end                              //end imul test
 end
else if (opcode[7:1] == 'b1101000)              // shl reg/mem, 1
 begin
 d = 0;
 if (update) @(negedge update);
 next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
 if (modem[7:6] == 'b11) dest = `REG;
 else dest = `MEM;
 if (src == `MEM || dest == `MEM)
  begin
  check = 1;
  @(negedge check);
  end
 func = 0;
 func[8:7] = 'b10;                    // selects shifter
 func[6:2] = 'b1;                     // count
 func[1:0] = 'b00;                    // selects shift left
 if (modem[5:3]=='b101 || modem[5:3]=='b001)       // actually right request
  func[0] = 'b1;
 if (modem[5:4] == 'b00)              // actually rotate requested
  func[1] = 'b1;
 fetch(instructions, func, d, modem, bread_reg, dest, src);
 if (get)
  begin
  @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
  bread_reg='b1;  //wait for cache to get operand
  end
 if (dest==`MEM)
  begin
  memsave=1;                       // memory save if needed
  @(negedge memsave);
  end
 end
else if (opcode[7:1] == 'b1100000)              // shl reg/mem, immed8
 begin
 d=0;
 w=0;                                 // 8-bit
```

```
  immed = 0;
  if (update) @(negedge update);
  next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
  if (modem[7:6] == 'b11) dest = `REG;
  else dest = `MEM;
  if (src == `MEM || dest == `MEM)
   begin
   getimm = 1;
   check = 1;
   @(negedge check);
   end
  if (!getimm) begin
   if (update) @(negedge update);
   next(immed[7:0],count,count, temp[0], temp[1], temp[2], temp[3],update);
   end
  func = 0;
  func[8:7] = 'b10;                          // selects shifter
  func[6:2] = immed[4:0];                    // count
  func[1:0] = 'b00;                          // selects shift left
  if (modem[5:3]=='b101 || modem[5:3]=='b001)       // actually shift right request
   func[0] = 'b1;
  if (modem[5:4] == 'b00)                    // actually rot<ate requested
   func[1] = 'b1;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;  //wait for cache to get operand
   end
  if (dest==`MEM)
   begin
   memsave=1;                               // memory save if needed
   @(negedge memsave);
   end
  end
else if (opcode[7:2] == 'b001000)                  // AND reg/mem, reg/mem
 begin
 d=opcode[1];           // direction of data flow
 w=opcode[0];           // w parameter indicates width of operands
 if (update) @(negedge update);
 next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
 if (d==0 && modem[7:6] != 'b11)
```

```
   begin
   dest=`MEM;
   src=`REG;
   end
  else if (modem[7:6] != 'b11)
   begin
   dest=`REG;
   src=`MEM;
   end
  else
   begin
   dest=`REG;
   src=`REG;
   end
  if (dest==`MEM || src==`MEM)
   begin
   check = 1;                    // check for memory access
   @(negedge check);                  // calculate addr if so
   end
  func = 5;                     // ALU AND
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;                     //wait for cache to get operand
   end
  if (dest==`MEM)
   begin
   memsave=1;                    // memory save if needed
   @(negedge memsave);
   end
  end
 else if (opcode[7:1]=='b0010010)          // and accum, immed
  begin
  dest=`REG;
  src=`IMM;
  w=opcode[0];
  modem=0;                     // reg=AX
  immed=0;
  if (update) @(negedge update);
  next(immed[7:0],count,count,temp[0],temp[1],temp[2],temp[3],update);
  if(w==1)                    // 16 bit operands
```

```
   begin
   if (update) @(negedge update);
   next(immed[15:8],count,count,temp[0],temp[1],temp[2],temp[3],update);
   end
  else signextend(8,16,immed,immed);  // sign extend 8 to 16 bits
  func=5;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;  //wait for cache to get operand
   end
  end
 else if (opcode[7:2] == 'b000010)                  // OR reg/mem, reg/mem
  begin
  d=opcode[1];                 // direction of data flow
  w=opcode[0];                 // w parameter indicates width of operands
  if (update) @(negedge update);
  next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
  if (d==0 && modem[7:6] != 'b11)
   begin
   dest=`MEM;
   src=`REG;
   end
  else if (modem[7:6] != 'b11)
   begin
   dest=`REG;
   src=`MEM;
   end
  else
   begin
   dest=`REG;
   src=`REG;
   end
  if (dest==`MEM || src==`MEM)
   begin
   check = 1;                    // check for memory access
   @(negedge check);                    // calculate addr if so
   end
  func = 6;                      // ALU OR
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
```

```
    begin
    @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
    bread_reg='b1;                          //wait for cache to get operand
    end
   if (dest==`MEM)
    begin
    memsave=1;                          // memory save if needed
    @(negedge memsave);
    end
   end
  else if (opcode[7:1]=='b0000110)              // or accum, immed
   begin
   dest=`REG;
   src=`IMM;
   w=opcode[0];
   modem=0;                        // reg=AX
   immed=0;
   if (update) @(negedge update);
   next(immed[7:0],count,count,temp[0],temp[1],temp[2],temp[3],update);
   if(w==1)                       // 16 bit operands
    begin
    if (update) @(negedge update);
    next(immed[15:8],count,count,temp[0],temp[1],temp[2],temp[3],update);
    end
   else signextend(8,16,immed,immed);   // sign extend 8 to 16 bits
   func=6;
   fetch(instructions, func, d, modem, bread_reg, dest, src);
   if (get)
    begin
    @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
    bread_reg='b1;   //wait for cache to get operand
    end
   end
  else if (opcode[7:2] == 'b001100)              // XOR reg/mem, reg/mem
   begin
   d=opcode[1];               // direction of data flow
   w=opcode[0];               // w parameter indicates width of operands
   if (update) @(negedge update);
   next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
   if (d==0 && modem[7:6] != 'b11)
    begin
    dest=`MEM;
```

```
   src=`REG;
   end
  else if (modem[7:6] != 'b11)
   begin
   dest=`REG;
   src=`MEM;
   end
  else
   begin
   dest=`REG;
   src=`REG;
   end
  if (dest==`MEM || src==`MEM)
   begin
   check = 1;                    // check for memory access
   @(negedge check);                  // calculate addr if so
   end
  func = 8;                      // ALU XOR
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;                       //wait for cache to get operand
   end
  if (dest==`MEM)
   begin
   memsave=1;                      // memory save if needed
   @(negedge memsave);
   end
  end
 else if (opcode[7:1]=='b0011010)              // xor accum, immed
  begin
  dest=`REG;
  src=`IMM;
  w=opcode[0];
  modem=0;                       // reg=AX
  immed=0;
  if (update) @(negedge update);
  next(immed[7:0],count,count,temp[0],temp[1],temp[2],temp[3],update);
  if(w==1)                     // 16 bit operands
   begin
   if (update) @(negedge update);
```

```
    next(immed[15:8],count,count,temp[0],temp[1],temp[2],temp[3],update);
    end
  else signextend(8,16,immed,immed);  // sign extend 8 to 16 bits
  func=8;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
    begin
    @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
    bread_reg='b1;  //wait for cache to get operand
    end
  end
else if (opcode[7:1] == 'b1111011)             // bitwise not
  begin
  d = 0;
  src = `NADA;
  if (update) @(negedge update);
  next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
  if (modem[5:3] == 'b111) idiv = 1;
  if (modem[5:3] == 'b101) imul = 2;
  if (!idiv && !imul) begin
  if (modem[7:6] == 'b11)  dest = `REG;
  else dest = `MEM;
  if (dest==`MEM || src==`MEM)
    begin
    check = 1;                   // check for memory access
    @(negedge check);                 // calculate addr if so
    end
  func = 7;                    // ALU not
  if (modem[5:3] == 'b011)            // this is actually neg
    func = 9;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
    begin
    @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
    bread_reg='b1;                    //wait for cache to get operand
    end
  if (dest==`MEM)
    begin
    memsave=1;                    // memory save if needed
    @(negedge memsave);
    end
  end
```

```
  end
 else if (opcode[7:2] == 'b001010)              // sub reg/mem, reg/mem
  begin
  d=opcode[1];           // direction of data flow
  w=opcode[0];           // w parameter indicates width of operands
  if (update) @(negedge update);
  next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
  if (d==0 && modem[7:6] != 'b11)
   begin
   dest=`MEM;
   src=`REG;
   end
  else if (modem[7:6] != 'b11)
   begin
   dest=`REG;
   src=`MEM;
   end
  else
   begin
   dest=`REG;
   src=`REG;
   end
  if (dest==`MEM || src==`MEM)
   begin
   check = 1;              // check for memory access
   @(negedge check);             // calculate addr if so
   end
  func = 4;               // ALU sub
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;               //wait for cache to get operand
   end
  if (dest==`MEM)
   begin
   memsave=1;              // memory save if needed
   @(negedge memsave);
   end
  end
 else if (opcode[7:1]=='b0010110)           // sub accum, immed
  begin
```

```
  dest=`REG;
  src=`IMM;
  w=opcode[0];
  modem=0;                        // reg=AX
  immed=0;
  if (update) @(negedge update);
  next(immed[7:0],count,count,temp[0],temp[1],temp[2],temp[3],update);
  if(w==1)                      // 16 bit operands
   begin
   if (update) @(negedge update);
   next(immed[15:8],count,count,temp[0],temp[1],temp[2],temp[3],update);
   end
  else signextend(8,16,immed,immed);  // sign extend 8 to 16 bits
  func=4;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;  //wait for cache to get operand
   end
  end
 else if (opcode[7:3] == 'b01000)          // inc reg
  begin
  src = `IMM;
  dest = `REG;
  modem[5:3] = opcode[2:1];
  d = 1;                                   // dest uses modem[5:3]
  immed = 1;
  func = 0;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  end
 else if (opcode[7:3] == 'b01001)          // dec reg
  begin
  src = `IMM;
  dest = `REG;
  modem[5:3] = opcode[2:1];
  d = 1;                                   // dest uses modem[5:3]
  immed = -1;
  func = 0;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  end
 else if (opcode[7:2] == 'b011010 && opcode[0] == 'b1)              // imul
```

```
  begin
  w = !opcode[1];
  dest = `REG;
  src = `IMM;
  if (update) @(negedge update);
  next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
  immed = 0;
  if (update) @(negedge update);
  next(immed[7:0],count,count,temp[0],temp[1],temp[2],temp[3],update);
  if (w==1)
   begin
   if (update) @(negedge update);
   next(immed[15:8],count,count,temp[0],temp[1],temp[2],temp[3],update);
   end
  else signextend(8,16,immed,immed);   // sign extend 8 to 16 bits
  func = 10;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;                           //wait for cache to get operand
   end
  end
 else if (opcode[7:3] == 'b01010) // push reg
  begin
  if (free) get = 1;
  else @(posedge free) get = 1;
  instructions[52] = 1;             // release tristate1 on sp
  instructions[29:22] = 1;          // aluout = op1
  instructions[21] = 1;             // aluout tristate released
  instructions[30] = 1;             // segin
  instructions[34] = 1;             // ssout to seg
  instructions[31] = 1;             // xlate
  @(posedge clk) instructions=0; // wait for addr calc
  instructions[48+opcode[2:0]]=1;         // release tristate1 on reg
  instructions[29:22] = 1;          // aluout = op1
  instructions[21] = 1;             // aluout tristate released
  memsave = 1;
  @(negedge memsave);
  @(posedge clk) instructions=0;
  immed = 4;
  instructions[12] = 1;             // immed out
```

```
  instructions[52] = 1;           // release tristate1 on sp
  instructions[29:22] = 0;        // alu = add
  instructions[21] = 1;           // aluout
  instructions[44] = 1;           // sp load
  @(posedge clk) instructions=0;
  end
 else if (opcode[7:2] == 'b011010 && opcode[0] == 0)        // push immed
  begin
  immed = 0;
  w = opcode[1];                  // size of immed
  if (update) @(negedge update);
  next(immed[7:0],count,count,temp[0],temp[1],temp[2],temp[3],update);
  if (w==1)
   begin
   if (update) @(negedge update);
   next(immed[15:8],count,count,temp[0],temp[1],temp[2],temp[3],update);
   end
  else signextend(8,16,immed,immed);  // sign extend 8 to 16 bits
  if (free) get = 1;
  else @(posedge free) get = 1;
  instructions[52] = 1;           // release tristate1 on sp
  instructions[29:22] = 1;        // aluout = op1
  instructions[21] = 1;           // aluout tristate released
  instructions[30] = 1;           // segin
  instructions[34] = 1;           // ssout to seg
  instructions[31] = 1;           // xlate
  @(posedge clk) instructions=0; // wait for addr calc
  instructions[12]=1;                     // release tristate immed
  instructions[29:22] = 2;        // aluout = op2
  instructions[21] = 1;           // aluout tristate released
  memsave = 1;
  @(negedge memsave);
  @(posedge clk) instructions=0;
  immed = 4;
  instructions[12] = 1;           // immed out
  instructions[52] = 1;           // release tristate1 on sp
  instructions[29:22] = 0;        // alu = add
  instructions[21] = 1;           // aluout
  instructions[44] = 1;           // sp load
  get = 'bz;
  @(posedge clk) instructions=0;
  end
```

```
else if (opcode[7:3] == 'b01011) // pop reg
  begin
  disp = 'hfffc;
  if (free) get = 1;
  else @(posedge free) get = 1;
  instructions[52] = 1;          // release tristate1 on sp
  instructions[29:22] = 1;       // aluout = op1
  instructions[21] = 1;          // aluout tristate released
  instructions[30] = 1;          // segin
  instructions[34] = 1;          // ssout to seg
  instructions[31] = 1;          // xlate
  instructions[1] = 1;           // allow disp through
  @(posedge clk) instructions=0; // wait for stack addr calc
  instructions[5] = 1;           // cacheaddrin
  bread_reg = 0;
  @(negedge bdav) get= 'bz;
  instructions[7] = 1;           // cacheout1
  instructions[21] = 1;          // aluout
  instructions[29:22] = 1;       // aluout = op1
  instructions[40+opcode[2:0]]=1;        // load reg
  bread_reg = 1;
  @(posedge clk) instructions=0;
  immed = 4;
  instructions[12] = 1;          // immed out
  instructions[52] = 1;          // release tristate1 on sp
  instructions[29:22] = 4;       // alu = sub
  instructions[21] = 1;          // aluout
  instructions[44] = 1;          // sp load
  @(posedge clk) instructions=0;
  end
else if (opcode[7:0] == 'b10001111)      // pop mem
  begin
  if (update) @(negedge update);
  next(modem,count,count,temp[0],temp[1],temp[2],temp[3],update);
  disp = 'hfffc;
  @(clk) if (free) get = 1;
  else @(posedge free) get =1;
  instructions[52] = 1;          // sp register on bus1
  instructions[29:22] = 1;       // aluout = bus1
  instructions[21] = 1;          // aluout
  instructions[30] = 1;          // segin
  instructions[31] = 1;          // xlate
```

```
    instructions[34] = 1;              // ss out to seg unit
    instructions[1] = 1;               // allow disp through
    instructions[5] = 1;               // cache addr in
    @(posedge clk);                    // wait for stack addr calc
    d =1;
    dest = `REG;
    src = `MEM;
    func = 2;                          // put stack value in ax
    fetch(instructions, func, d, modem, bread_reg, dest, src);
    @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
    bread_reg='b1;                     // wait for cache to get operand
    @(posedge clk) instructions=0; // decrement stack pointer
    immed = 4;
    instructions[12] = 1;              // immed out
    instructions[52] = 1;              // release tristate1 on sp
    instructions[29:22] = 4;           // alu = sub
    instructions[21] = 1;              // aluout
    instructions[44] = 1;              // sp load
    @(posedge clk) instructions=0; // move ax to modem addr
    d = 0;
    src = `REG;
    dest = `MEM;
    get = 'bz;
    check = 1;                          // check for memory access
    @(negedge check);                            // determine memory address
    func = 2;
    fetch(instructions, func, d, modem, bread_reg, dest, src);
    @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
    bread_reg='b1;                     // wait for cache to get operand
    memsave = 1;
    @(negedge memsave);
    @(posedge clk) instructions=0;
    end
  else if (opcode[7:0] == 'b11111110)              // inc reg/mem
    begin
    src=`IMM;                                 // immed on src
    w=opcode[0];
    if (update) @(negedge update);
    next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
    if (modem[7:6]!='b11) dest=`MEM;
    else dest=`REG;
    if (dest==`MEM || src==`MEM)
```

```
   begin
   check =1;                          // check for memory access
   @(negedge check);
   end
  immed = 1;                              // add 1
  if (modem[5:3] == 'b001)                // this is actually dec
   func = 4;
  else func=0;                            // alu add function
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)                      // read from memory
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;                     //wait for cache to get operand
   end
  if (dest==`MEM)
   begin
   memsave=1;                       // memory save if needed
   @(negedge memsave);
   end
  opcode = 0;
  end
 if (opcode[7:0] == 'b11111111)  // push mem
  begin
  d=1;
  dest = `REG;
  src = `MEM;
  modem[5:3] = 000;                    // ax is our temporary register
  check = 1;
  @(negedge check);                    // calc addr from modem
  func = 2;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get) @(negedge bdav) bread_reg = 1;
  instructions[40] = 1;           // load ax
  @(posedge clk) instructions=0; // [modem] -> ax
  instructions[52] = 1;           // sp register on bus1
  instructions[29:22] = 1;        // aluout = bus1
  instructions[21] = 1;           // aluout
  instructions[30] = 1;           // segin
  instructions[31] = 1;           // xlate
  instructions[34] = 1;           // ss out to seg unit
  instructions[1] = 1;            // allow disp through
  instructions[5] = 1;            // cache addr in
```

```
  @(posedge clk);              // wait for stack addr calc
  d =0;
  dest = `MEM;
  src = `REG;
  func = 2;                    // put stack value in ax
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;                    // wait for cache to get operand
   end                         // now stack -> ax
  memsave = 1;
  @(negedge memsave);
  @(posedge clk) instructions=0;
  immed = 4;
  instructions[12] = 1;        // immed out
  instructions[52] = 1;        // release tristate1 on sp
  instructions[29:22] = 0;     // alu = add
  instructions[21] = 1;        // aluout
  instructions[44] = 1;        // sp load
  end
 if (idiv)                           // idiv
  begin
  dest = `REG;                       // first mov ax -> dx
  src = `REG;
  d=0;
  opcode = modem;
  modem = 'b11000010;                      // ax -> dx
  func = 2;                          // aluout = bus2
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  @(posedge clk) instructions = 0;
  d = 1;                             // then ax = ax / modem
  modem = opcode;
  modem[5:3] = 'b000;                      // force AX reg
  if (modem[7:6]!='b11) src=`MEM;
  else src=`REG;
  if (dest==`MEM || src==`MEM)
   begin
   check =1;                         // calculate memory addr
   @(negedge check);
   end
  func = 11;                         // div
```

```
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get) @(negedge bdav) bread_reg=1;
  instructions[40] = 1;                     // load ax
  @(posedge clk) modem[5:3] = 'b010;         // force DX reg
  func = 12;                                // finally dx = dx mod modem
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get) @(negedge bdav) bread_reg=1;
  instructions[42] = 1;                     // load dx
  end
 if (imul == 1)                             // imul reg,reg/mem
  begin
  d = 1;                                    // dest = reg
  dest = `REG;
  if (update) @(negedge update);
  next(modem, count, count, temp[0], temp[1], temp[2], temp[3], update);
  if (modem[7:6]!='b11) src=`MEM;
  else src=`REG;
  if (dest==`MEM || src==`MEM)
   begin
   check =1;                                // check for memory access
   @(negedge check);
   end
  func = 10;
  fetch(instructions, func, d, modem, bread_reg, dest, src);
  if (get)
   begin
   @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
   bread_reg='b1;                           //wait for cache to get operand
   end
  end
 if (imul == 2)              // imul reg/mem
  begin
  if (modem[7:6]!='b11) dest=`MEM;
  else dest=`REG;
  if (dest==`MEM || src==`MEM)
   begin
   check =1;                                // check for memory access
   @(negedge check);
   end
  src = `REG;                                     // multiply with AX
  modem[5:3] = 0;                           // force AX register
  d = 0;                                    // dest determined by mod,r/m
```

```
    func = 10;                                   // multiply
    fetch(instructions, func, d, modem, bread_reg, dest, src);
    if (get)
     begin
     @(negedge bdav) if (src == `MEM) instructions[40+modem[5:3]] = 1;
     bread_reg='b1;                              //wait for cache to get operand
     end
    if (dest==`MEM)
     begin
     memsave=1;                                 // memory save if needed
     @(negedge memsave);
     end
    end
end

task fetch;
output [79:0] instructions;
input [8:0] func;
input d;
input [7:0] modem;
output bread_reg;
input dest;
integer dest;
input src;
integer src;
reg [79:0] instr;
 begin
 instr = 0;
 bread_reg = 'bz;
 if (dest==`REG)              // dest = register
  begin
  if (!cmp) instr[40+modem[2:0]] = 1;             // load register
  instr[48+modem[2:0]] = 1;                 // register tristate 1
  end
 if (dest==`MEM)              // dest = cache
  begin
  instr[7] = 1;        // cache 1
  instr[5] = 1;        // let cache addr through
  bread_reg = 0;              // negative assertion;
  end
 if (dest==`EIP)              // EIP
  begin
```

```
  instr[2] = 1;                  // eip in
  instr[21] = 1;                 // alu out
  instr[3] = 1;                  // eip out
  end
 if (dest==`SREG)
  instr[36+modem[5:3]] = 1;                // allows loading of sreg
 if (src == `SREG)
  instr[64+modem[5:3]] = 1;                // outputs sreg on bus2
 if (src==`REG)
  begin
  if (dest == `SREG) instr[56+modem[2:0]] = 1;
  else instr[56+modem[5:3]] = 1;           // register tristate 2
  end
 if (dest==`REG && d==1)                   // reverse direction
  begin
  instr[40+modem[2:0]] = 0;                // reverse previous load
  if (src != `MEM && !cmp) instr[40+modem[5:3]] = 1;      // load register
  if (src!=`REG)
   begin
   instr[48+modem[2:0]] = 0;
   instr[48+modem[5:3]] = 1;               // tristate reg
   end
  end
 if (src==`MEM)                            // Cache read
  begin
  instr[8]=1;
  instr[5]=1;
  bread_reg=0;
  end
 if (src==`IMM) instr[12] = 1;             // Immediate Value
 if (func[8] == 1)                  // pick Shift(1) or ALU(0)
  begin
  instr[13] = 1;                    // shift out
  instr[20:14] = func [6:0];
  end
 else
  begin
  instr[21] = 1;                    // alu out
  instr[29:22] = func[7:0];
  end
 instr[4] = 1;                      // set flags enabled
 instructions = instr;
```

```
  end
endtask

task calc;
output [79:0] instructions;
input [7:0] modem;
reg [2:0] rm;
reg [79:0] instr;
 begin
 instr=0;
 rm=modem[2:0];
 if (rm == 0)
   begin
   instr[51] = 1;        // BX1
   instr[62] = 1;        // SI2
   instr[32] = 1;        // DSOUT
   end
 if (rm == 1)
   begin
   instr[51] = 1;
   instr[63] = 1;
   instr[32] = 1;
   end
 if (rm == 2)
   begin
   instr[53] = 1;
   instr[62] = 1;
   instr[34] = 1;
   end
 if (rm == 3)
  begin
  instr[53] = 1;
  instr[63] = 1;
  instr[34] = 1;
  end
 if (rm == 4)
  begin
  instr[29:22] = 1;          // aluout = op1
  instr[54] = 1;
  instr[32] = 1;
  end
 if (rm == 5)
```

```
    begin
    instr[29:22] = 1;
    instr[55] = 1;
    instr[32] = 1;
    end
   if (rm == 6)
    begin
    instr[29:22] = 2;           // aluout = op2
    instr[12] = 1;
    instr[32] = 1;
    end
   if (rm == 7)
    begin
    instr[29:22] = 1;
    instr[51] = 1;
    instr[32] = 1;
    end
   instr[30] = 1;        // SEGIN
   instr[31] = 1;       // xlate
   instr[5] = 1;         // CADDR
   instr[21] = 1;        // ALUOUT
   instructions = instr;                // assumes [29:22] == 0 is simple add
   end
 endtask

 task next;
 output [7:0] var;
 output outcnt;
 integer outcnt;
 input incnt;
 integer incnt;
 input [7:0] t0;
 input [7:0] t1;
 input [7:0] t2;
 input [7:0] t3;
 output update;
 integer num;
 reg [7:0] temp [3:0];

  begin
  temp[0] = t0;
  temp[1] = t1;
```

```
  temp[2] = t2;
  temp[3] = t3;
  var = temp[incnt];
  num = incnt + 1;
  if (num > 3)
   begin
   outcnt = 0;
   update = 1;
   end
  else
   begin
   outcnt = num;
   update = 0;
   end
  end
endtask

task signextend;
input original;
integer original;
input result;
integer result;
input [31:0] in;
output [31:0] out;
reg [15:0] temp16;
reg [31:0] temp32;
reg nochange;
 begin
 nochange = 1;
 if (original == 8)                // it has to be!
  begin
  if (in[7] == 'b1)                // 2's comp
   begin
   if (result == 16) temp16 = in[6:0] - 128;
   if (result == 32) temp32 = in[6:0] - 128;
   nochange = 0;
   end
  end
 if (original == 16)
  begin
  if (in[15] == 'b1)               // 2's comp
   begin
```

```
    if (result == 32) temp32 = in[14:0] - 32768;
    if (result == 16) temp16 = in;
    nochange = 0;
    end
   end
  if (result == 16 && !nochange) out = temp16;
  if (result == 32 && !nochange) out = temp32;
  if (nochange) out = in;
  end
endtask

task save;
output [79:0] instructions;
output bwrite_reg;
input [79:0] instrs;
reg [79:0] instr;
  begin
  instr=instrs;
  instr[5]=1;
  instr[6]=1;                 // cache addr and data
  bwrite_reg = 0;             // negative assertion;
  instructions = instr;
  end
endtask

task jmp;
output resume;
output [15:0] immed;
input [15:0] immedin;
input count;
integer count;
output rem;
integer rem;
integer blocks, remainder;
  begin
  resume = 1;
  blocks = immedin / 4;
  remainder = immedin % 4;
  blocks = blocks + ((count+remainder) / 4);
  rem = ((count+remainder) % 4);
  immed = blocks *4;                    // gives number of address units
//$display("rem:%h, blocks:%h, immedin:%h, count:%h",rem,blocks,immedin,count);
```

```
  end
 endtask

 always @(posedge check) begin
  num = modem[7:6];
  if (num==0 && modem[2:0]==6) num=2;
  if (num==1||num==2)
   begin
   disp = 0;
   if (update) @(negedge update);
   next(disp[7:0],count,count,temp[0],temp[1],temp[2],temp[3],update);
   end
  if (num==2)
   begin
   if (update) @(negedge update);
   next(disp[15:8],count,count,temp[0],temp[1],temp[2],temp[3],update);
   end
  if (num==1) signextend(8,16,disp,disp); // sign extend 8 to 16 bits
  if (getimm)
   begin
   immed = 0;
   if (update) @(negedge update);
   next(immed[7:0],count,count,temp[0],temp[1],temp[2],temp[3],update);
   if (w==1)
    begin
    if (update) @(negedge update);
    next(immed[15:8],count,count,temp[0],temp[1],temp[2],temp[3],update);
    end
   else signextend(8,16,immed,immed);   // sign extend 8 to 16 bits
   end
  @(clk) if (free) get = 1;
  else @(posedge free) get = 1;
  calc(instructions, modem);
  if (num==1||num==2||autodisp==1) instructions[1] = 1;       // allow disp through
  @(posedge clk) instructions[31]=0;                // return xlate to zero
  instructions[1] = 0;
  check = 0;
 end

 always @(posedge memsave) begin
  @(posedge clk) save(instructions, bwrite_reg, instructions);
  @(negedge bdav) bwrite_reg='b1;
```

```
 memsave=0;
end

endmodule
```

Sections.v

```
module tristate (in, out, state);
input [31:0] in;
output [31:0] out;
input state;            // 1=pass, 0=high Z
reg [31:0] out;
always @(state or in) begin
 if (!state) out=32'bz;
 if (state) out=in;
end
endmodule

module shifter (in1, out, dir, type, count, clk, setflags, fdone);
//16 bit shifter
input [15:0] in1;
output [15:0] out;
input dir, type, clk, fdone;  // dir= 0:left 1:right, type= 0:shift 1:rotate
input [4:0] count;
inout setflags;
reg [15:0] in, temp;
reg [15:0] out;
reg set_reg, t;
wire setflags=set_reg;
integer i;

always @(fdone) begin
 set_reg='bz;
end

always @(in1 or type or count or dir) begin
 in = in1;
 if (!type)
  begin
  out = (dir)?
   (in >> count) : (in << count);
  end
 else
  begin
  temp = in;
  if (dir)
   begin
   for (i=0; i<count; i=i+1)
    begin
```

```
      t = temp[0];
      temp[14:0] = temp[15:1];
      temp[15] = t;
      end
    end
   else
    begin
    for (i=0; i<count; i=i+1)
      begin
      t = temp[15];
      temp[15:1] = temp[14:0];
      temp[0] = t;
      end
    end
   out = temp;
   end
  set_reg = 1;                  // sets the flags
//$display("in:%h, type:%h, dir:%h, count:%h, out:%h",in,type,dir,count,out);
end
endmodule

module segmentation (in, out, xlate, segregs, clk, page, disp);
input [31:0] in;
input [15:0] segregs, disp;
output [31:0] out;
input xlate, clk;
output page;                        // returns 1 if page fault
reg [31:0] out;
integer temp, tempdisp;
reg page;
// segaddr ,-=> linaddr, does not give page faults
initial begin
  page = 0;
end
always @(posedge xlate) begin             // negative assertion
  temp=0;
  tempdisp = 0;
  @(clk) if (segregs) temp=segregs<<2;
  if (disp[15] == 'b1)                // 2's complement
    tempdisp = disp[14:0] - 32768;
  else if (disp[15] == 'b0) tempdisp = disp;
  if (tempdisp) temp = temp + tempdisp;
```

```
 out = in+temp;
//$display("SEG: in=%h, out=%h, disp=%h, segregs=%h",in,out,disp,segregs);
end
endmodule

module paging (segaddr, linaddr, xlate, csreg, clk, page);
input [31:0] segaddr;
input [15:0] csreg;
output [31:0] linaddr;
input xlate, clk;
output page;                         // returns 1 if page fault
reg [31:0] linaddr, temp;
reg page;
// exclusive use by prefetch
initial begin
 page = 0;
end
always @(negedge xlate) begin              // negative assertion
 temp=csreg<<2;
 linaddr = segaddr+temp;
end
endmodule

module address (load, count, pcin, eip, clk);      // controls EIP
input load, count, clk;
input [31:0] pcin;
output [31:0] eip;
reg [31:0] eip;
initial begin
 eip=0;
end
always @(posedge clk) begin
 if (load) eip = pcin;
 else if (count) eip = eip + 4;       // assumes increments of 4 bytes
end
endmodule

module flag (setflags, value, flagreg, set, fdone);
input setflags, set;
input [15:0] value;
output [15:0] flagreg;
output fdone;
```

```
reg fdone;
reg [15:0] flagreg;
integer count, number;

initial begin
  fdone = 0;                              // signal to alu and shifter
end

always @(posedge setflags) begin
//$display("in flag routine, fdone=%b",fdone);
  if (set) begin
  #1 number = 0;
  flagreg = 0;
  if (value[15:8]) flagreg[0]=1;          // carry
  for (count=0; count<16; count=count+1)
    if (value[count] == 1) number = number + 1;
  if (!number[0]) flagreg[2]=1;                 // even parity
  if (value[15:4]) flagreg[4]=1;          // aux carry
  if (value==0) flagreg[6]=1;                   // zero
  flagreg[7]=value[15];                         // sign
  if (value[15]) flagreg[11]=1;                 // overflow
//$display("value:%h carry:%b, zero:%b, sign:%b, over:%b, even:%b", value,
flagreg[0], flagreg[6], flagreg[7], flagreg[11], flagreg[2]);
  end
  fdone = !fdone;
end
endmodule

module alu (op1, op2, out, instr, clk, setflags, alusupp, fdone);
input [15:0] op1, op2;
input [2:0] alusupp;
output [31:0] out;
input [7:0] instr;              // 255 different operations
reg [31:0] out;
integer top1, top2;
input clk, fdone;
inout setflags;
reg set_reg;
wire setflags=set_reg;
integer temp1, temp2, temp3, temp4;

always @(fdone) begin
```

```
 set_reg='bz;
end

always @(op1 or op2 or instr) begin
 top1= 0;
 top2=0;
 if (op1[15] == 'b1)                // 2's complement
  top1 = op1[14:0] - 32768;
 else if (op1[15] == 'b0) top1 = op1;
 if (op2[15] == 'b1)                // 2's complement
  top2 = op2[14:0] - 32768;
 else if (op2[15] == 'b0) top2 = op2;
 if (instr==0)             // add
  out = top1 + top2;
 else if (instr==1)        // return op1
  out = top1;
 else if (instr==2)        // return op2
  out = top2;
 else if (instr==3)        // even up addr with offset on op2
  begin
  temp1 = top2 / 4;
  temp2 = top2 % 4;
  temp1 = temp1 + ((alusupp+temp2) / 4);
  temp1 = temp1 * 4;
  out = temp1 + top1;
  end
 else if (instr==4)        // subtract
  out = top1 - top2;
 else if (instr==5)        // bitwise and
  out = op1 & op2;         // not sign extended
 else if (instr==6)        // bitwise or
  out = op1 | op2;         // not sign extended
 else if (instr==7)        // bitwise not
  out = ~op1;
 else if (instr==8)        // bitwise xor
  begin
  temp1 = op1 | op2;
  temp2 = op1 & op2;
  temp2 = ~temp2;
  out = temp1 & temp2;
  end
 else if (instr==9)        // not
```

```
    out = -1 * top1;            // it is sign extended
  else if (instr=='ha)          // imul
    out = top1 * top2;
  else if (instr=='hb)          // idiv
    out = top2 / top1;
  else if (instr=='hc)          // mod
    out = top2 % top1;
//$display("op1:%h, op2:%h, top1:%h, top2:%h, func:%h, out:%h, set:%b", op1, op2,
top1, top2, instr, out,setflags);
  set_reg = 1;                  // set flags
end
endmodule
```

Clock.v

```
module clock(clk);
output clk;
reg clk;

initial begin
  clk = 0;
end

always begin
  #1 clk = !clk;
end

endmodule
```

Totalmake.v

```
# Follow the directions in the comment header for assigning file names to the
# make variables.
#
# To compile from the Unix prompt: type "make".
#
# To compile within emacs: type "meta-x compile" then hit return to select
# the default make parameters.
#
# Makefile for Verilog programs.
#
# TERMINOLOGY:
#
# extended Verilog:  Verilog programs that have cpp directives in them.
#                    This Makefile assumes that the filename suffix for
#                    these programs is ".v".
#
# raw Verilog:               Verilog programs that are suitable for processing by
#               the Verilog compiler/simulator.
#                    This Makefile assumes that the filename suffix for
#                    these programs is ".vraw".
#
#
# IMPORTANT PARAMETERS:
#
# Add the names of the .vraw files that you want to create to the
# variable RAW_VERILOG_FILES. In order for this Makefile to be successful,
# there should be a file <xxxx.v> in the current directory for every file
# <xxxx.vraw> named in the RAW_VERILOG_FILES variable.
#
# HOW TO USE:
#
# The following targets have been defined in this Makefile. The
# "simulate" target is the default.
#
# update - creates new versions of the raw Verilog files from the
#        corresponding extended Verilog files.
#
# simulate - updates the files (as explained above) and starts
#          up the simulator with all the files defined in the
#          RAW_VERILOG_FILES variable.
#
```

Totalmake.v

```
#
***************************************************************************
*

# The names of all the files in your Verilog program.

RAW_VERILOG_FILES =  i486.v sections.v decode.v prefetch.v cache.v parity.v
clock.v biu.v testing.v



# Pathname of the Verilog program.
VERILOG =  /home/srco/src/verilog/verilog/exe/verilog  -f
/home/srco/src/verilog/verilog/passwd
XVERILOG =  /home/srco/src/verilog/verilog/exe/verilog-grx  -f
/home/srco/src/verilog/verilog/passwd

# Pathname of cpp.
CPP = /usr/lib/cpp

# Flags to pass to cpp.
CPP_FLAGS = -B -C -P


#
***************************************************************************
*
#
***************************************************************************
*
#
***************************************************************************
*
# YOU DO NOT NEED TO CHANGE ANYTHING BELOW THIS LINE.
#
***************************************************************************
*

# Default rule making raw Verilog files from extended Verilog files.
%.vraw : %.v
	$(CPP) $(CPP_FLAGS) $< $@
```

Totalmake.v

```
# ************************************************
# Dependencies.
# ************************************************

simulate : update
	$(VERILOG) $(RAW_VERILOG_FILES)

xsimulate : update
	$(XVERILOG) $(RAW_VERILOG_FILES)

update : $(RAW_VERILOG_FILES)
```

Appendix 2

Datapath and ALU Controls

ALU Functions

Input	Function
0	Add
1	Pipe operand bus 1 to the output without alteration
2	Pipe operand bus 2 to the output without alteration
3	Compute jump address (op1 = Instr ptr, op2 = offset, alusupp = decode queue ptr)
4	Subtract
5	Bitwise and
6	Bitwise or
7	Bitwise not
8	Bitwise xor
9	Neg (make 2's complement input negative)
10	Multiply
11	Divide (op2 / op1)
12	Mod (op2 / op1)

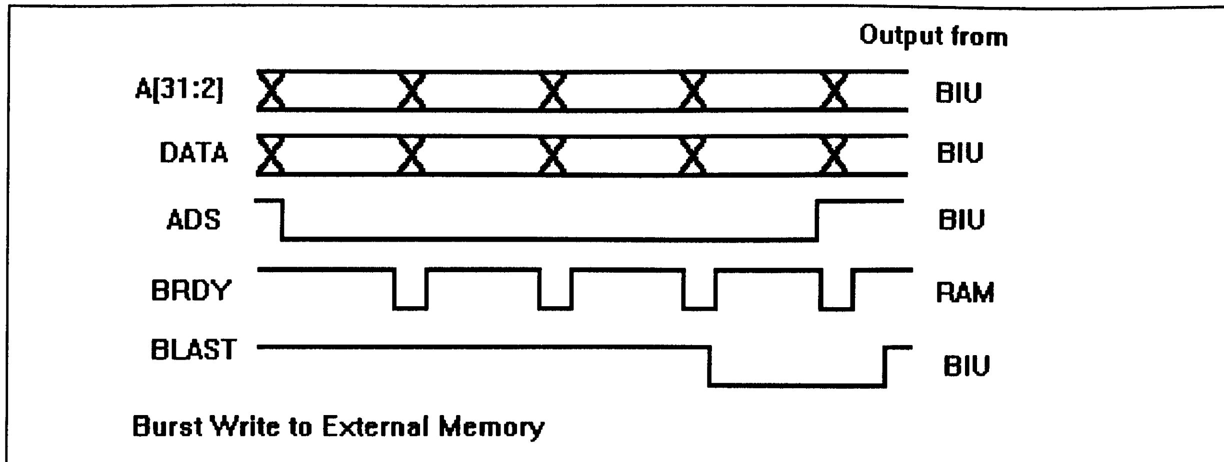
Datapath Control Instructions

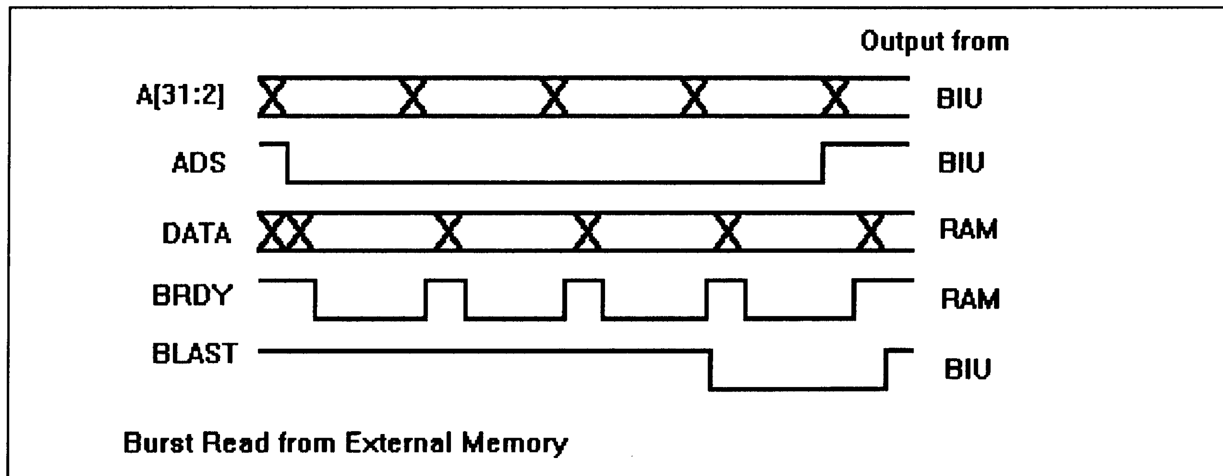
Instruction Bit	Controls
0	Releases flag value to operand bus 2
1	Releases memory displacement value to segmentation unit
2	Releases input from databus to instruction pointer
3	Releases instruction pointer to operand bus 1
4	Sets flag register with databus output
5	Releases input from address bus to cache address pins
6	Releases input from databus to cache data pins
7	Releases cache output to operand bus 1
8	Releases cache output to operand bus 2
9 - 11	ALU supplemental value [0:2]
12	Releases Immediate value from decode unit to operand bus 2
13	Releases output of barrel shifter onto databus
14	Direction bit to the barrel shifter (1=right, 0=left)
15	Type bit to the barrel shifter (1=rotate, 0=shift)
16 - 20	Amount to shift or rotate [0:4]
21	Releases output of ALU onto databus
22 - 29	ALU function select [0:7]
30	Releases value on databus to segmentation unit

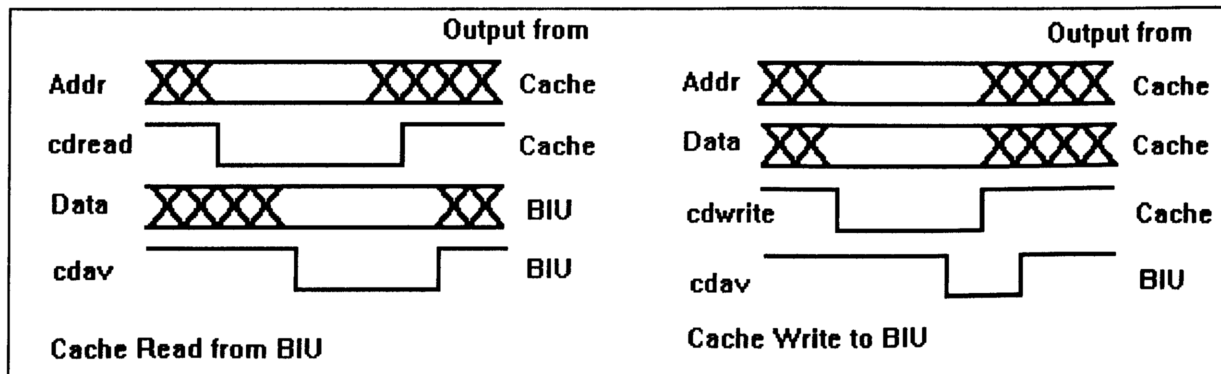
31	Translate signal to segmentation unit
32 - 35	Releases [DS, CS, SS, ES] segmentation register value to segmentation unit
36 - 39	Load segmentation registers [DS, CS, SS, ES]
40 - 47	Load general purpose register [AX, CX, DX, BX, SP, BP, SI, DI]
48 - 55	Releases register [AX, CX, DX, BX, SP, BP, SI, DI] value to operand bus 1
56 - 63	Releases register [AX, CX, DX, BX, SP, BP, SI, DI] value to operand bus 2
64 - 67	Releases segmentation register [DS, CS, SS, ES] to operand bus 2

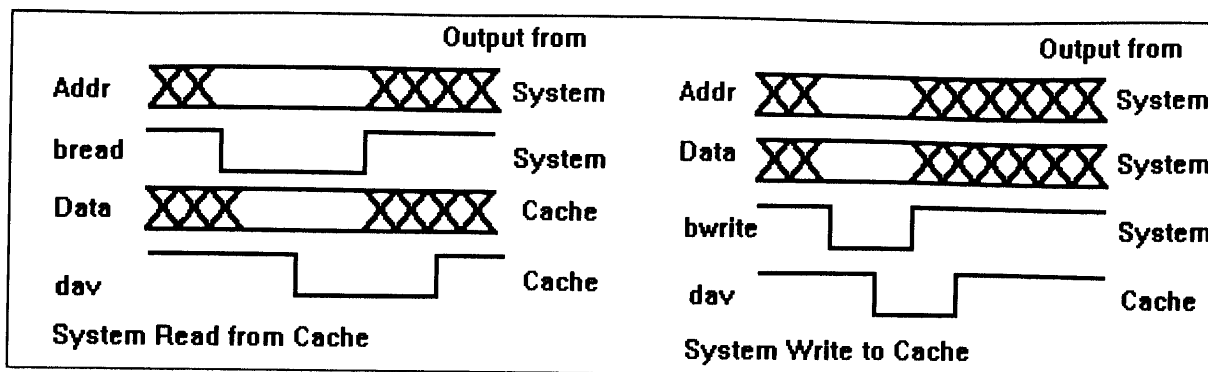
Appendix 3

Handshaking Signals








Output from
Addr
System
bread
System
Data
Cache
dav
Cache
System Read from Cache
Output from
Addr
System
Data
System
bwrite
System
dav
Cache
System Write to Cache

Appendix 4

Verification

testcode - This file contains the assembly instructions for the system test and the fibinocci sequence. It also shows the binary encodings of these instructions (in hex).

rtest.v - This file tests out the external ram by writing values to it and reading them back.

ptest.v - This module tests out the parity check unit. Simulated inputs are given to the unit with its outputs compared to expected values.

pagetest.v - This module tests the paging unit by translating a few segmented addresses.

segtest.v - This tests the physical address translation ability of the segmentation unit, including the use of memory displacements.

flagtest.v - This tests the different values of the flag register when given predefined inputs.

addrtest.v - This file tests the address register, making sure it counts properly when requested.

shftest.v - This is a test of the shift register, testing both shifts and rotates, both left and right.

alutest.v - This tests out all the arithmetic and logic functions of the alu.

btest.v - This module extensively tests the bus interface unit and the handshaking between it and the external ram. The ram's operation should be verified before using this test.

ctest.v - This module tests the cache's operation and the interaction between it, the biu and the ram. The biu and the ram's operation should be verified before this test.

pftest.v - This test will verify the prefetch unit and the handshaking between it, the cache, the biu, and the ram. The other units should be verified prior to this test.

analysis - This file should be "sourced". It uses the various makefiles to run all the module tests in proper sequence followed by the complete system test. Results will scroll off screen so pauses will be necessary.

Testing Procedure

```
NOP                     $90             // Tests for MOV
MOV AX, $64             $B064
CMP AX, $64             $80F864
JE $05                  $7405
OR [SI+$0], $0002       $814C000200
MOV BX, $2000           $BB0020
MOV [BX+$4], AX         $884704
CMP [BX+$4], $64        $807F0464
JE $05                  $7405
OR [SI+$0], $0001       $814C000100
MOV [BX+$4], $A4        $C64704A4
CMP [BX+$4], $A4        $807F04A4
JE $05                  $7405
OR [SI+$0], $0004       $814C000400
MOV AX, [BX+$4]         $8A4704
CMP AX, $A4             $3CA4
JE $05                  $7405
OR [SI+$0], $0008       $814C000800
MOV [BX+$8], AX         $884708
CMP [BX+$8], $A4        $807F08A4
JE $05                  $7405
OR [SI+$0], $0010       $814C001000
MOV CX, 0               $B100
MOV AX, $08             $B008
MOV DS, AX              $8EC0
MOV BX, DS              $8CC3
MOV DS, CX              $8EC1           // Restore DS reg to original level
CMP BX, $08             $80FB08
JE $05                  $7405
OR [SI+$0], $0020       $814C002000
                                // Tests for POP and PUSH
MOV AX, $A4             $B0A4
MOV SP, $1000           $BC0010         // Stack starts at location $1000
MOV BX, $1000           $BB0010
PUSH BX                 $53
PUSH AX                 $50
CMP [BX+4], $A4         $807F04A4
JE $05                  $7405
OR [SI+$8], $0001       $814C080100
POP CX                  $59
```

Testcode

```
CMP CX, $A4          $80F9A4
JE $05               $7405
OR [SI+$4], $0001    $814C040100
MOV DI, $2000        $BF0020         // Test memory locations
MOV CX, $64          $B164
MOV [DI], CX         $880D
PUSH [DI]            $FF35
CMP [BX+4], $64      $807F0464
JE $05               $7405
OR [SI+$8], $0002    $814C080200
POP [DI+$8]          $8F4508
CMP [DI+$8], $64     $807D0864
JE $05               $7405
OR [SI+$4], $0002    $814C040200
POP BX               $5B       // Verifies old values in stack
CMP BX, $1000        $81FB0010
JE $05               $7405
OR [SI+$4], $0004    $814C040400
                           // Test ADD
MOV AX, $30          $B030
MOV BX, $22          $B322
ADD AX, BX           $00D8
CMP AX, $52          $3C52
JE $05               $7405
OR [SI+$OC], $0001   $814C0C0100
MOV BX, $2000        $BB0020
MOV [BX], AX         $8807
ADD [BX], $05        $800705
CMP [BX], $57        $803F57
JE $05               $7405
OR [SI+$0C], $0002   $814C0C0200
ADD AX, $0006        $050600
CMP AX, $58          $3C58
JE $05               $7405
OR [SI+$0C], $0004   $814C0C0400
                           // Test DEC
MOV [BX], $0AB0      $C707B00A
DEC [BX]             $FE0F
CMP [BX], $0AAF      $813FAF0A
JE $05               $7405
OR [SI+$10], $0001   $814C100100
                           // Test IDIV
```

Testcode

```
MOV AX, $04             $B004
MOV [BX], $1001         $C7070110
IDIV [BX]               $F63F
CMP AX, $0400           $3D0004
JE $05                  $7405
OR [SI+$14], $0001      $814C140100
CMP DX, $01             $80FA01
JE $05                  $7405
OR [SI+$14], $0001      $814C140100
                                // Tests IMUL
MOV AX, $04             $B004
MOV BX, $08             $B308
IMUL BX                 $F6EB
CMP BX, $20             $80FB20
JE $05                  $7405
OR [SI+$18], $0001      $814C180100
IMUL BX, $02            $6BC302
CMP BX, $40             $80FB40
JE $05                  $7405
OR [SI+$18], $0002      $814C180200
MOV BX, $1000           $BB0010
MOV AX, $10             $B010
MOV [BX], $04           $C60704
IMUL AX, [BX]           $0FAF07
CMP AX, $40             $3C40
JE $05                  $7405
OR [SI+$18], $0004      $814C180400
                                // Tests INC
MOV AX, $64             $B064
INC AX                  $FEC0
CMP AX, $65             $3C65
JE $05                  $7405
OR [SI+$1C], $0001      $814C1C0100
                                // Tests NEG
NEG [BX]                $F61F           // BX:$1000:04 from above
CMP [BX], $FC           $803FFC         // 2's Comp
JE $05                  $7405
OR [SI+$20], $0001      $814C200100
                                // Tests SUB
MOV [BX], $40           $C60740
MOV AX, $12             $B012
SUB [BX], AX            $2807
```

Testcode

```
CMP [BX], $2E           $803F2E
JE $05                  $7405
OR [SI+$24], $0001      $814C240100
SUB AX, $03             $80E803        // using SUB reg/mem, immed
CMP AX, $0F             $3C0F
JE $05                  $7405
OR [SI+$24], $0002      $814C240200    // using SUB ax, immed
SUB AX, $10             $2C10          // Assumes ax:$0f from previous
CMP AX, $FF             $3CFF          // -1 in 2's comp
JE $05                  $7405
OR [SI+$24], $0004      $814C240400
                               // Tests AND
MOV AX, $0F             $B00F
MOV BX, $0A             $B30A
AND AX, BX              $20D8
CMP AX, $0A             $3C0A
JE $05                  $7405
OR [SI+$28], $0001      $814C280100
MOV BX, $1000           $BB0010
MOV [BX], $0F           $C6070F
AND [BX], $0A           $80270A
CMP [BX], $0A           $803F0A
JE $05                  $7405
OR [SI+$28], $0002      $814C280200
AND AX, $0E             $240E     // Assumes ax:0a from above
CMP AX, $0A             $80F80A
JE $05                  $7405
OR [SI+$28], $0004      $814C280400
                               // Tests NOT
MOV [BX], $55           $C60755
NOT [BX]                $F617
CMP [BX], $FFAA         $813FAAFF
JE $05                  $7405
OR [SI+$2C], $0001      $814C2C0100
                               // Tests OR
MOV AX, $0A             $B00A
MOV BX, $00A0           $BBA000
OR BX, AX               $08C3
CMP BX, $00AA           $81FBAA00
JE $05                  $7405
OR [SI+$30], $0001      $814C300100
OR AX, $00A0            $81C8A000
```

Testcode

```
CMP AX, $00AA          $81F8AA00
JE $05                 $7405
OR [SI+$30], $0004     $814C300400
                           // Tests XOR
MOV BX, $1000          $BB0010
MOV [BX], $2A          $C6072A
MOV AX, $7C            $B07C
XOR [BX], AX           $3007
CMP [BX], $56          $803F56
JE $05                 $7405
OR [SI+$34], $0001     $814C340100
XOR [BX], $00FF        $8137FF00          // Assumes bx:$56 from previous
CMP [BX], $00A9        $813FA900
JE $05                 $7405
OR [SI+$34], $0002     $814C340200
XOR AX, $2A            $342A
CMP AX, $56            $80F856
JE $05                 $7405
OR [SI+$34], $0004     $814C340400
                           // Tests ROL
MOV [BX], $0F          $C6070F
ROL [BX], 1            $D007
CMP [BX], $1E          $803F1E
JE $05                 $7405
OR [SI+$38], $0001     $814C380100
ROL [BX], 5            $C00705            // Assumes bx:1e from previous
CMP [BX], $03c0        $813FC003
JE $05                 $7405
OR [SI+$38], $0002     $814C380200
                           // Tests ROR
MOV AX, $0F            $B00F
ROR AX, 1;             $D0C8
CMP AX, $8007          $3D0780
JE $05                 $7405
OR [SI+$3C], $0001     $814C3C0100
ROR AX, 5              $C0C805            // Assumes ax:$8007 from previous
CMP AX, $3C00          $3D003C
JE $05                 $7405
OR [SI+$3C], $0002     $814C3C0200
                           // Tests SHL
MOV [BX], $0F          $C6070F
SHL [BX], 1            $D027
```

Testcode

```
CMP [BX], $1E           $803F1E
JE $05                  $7405
OR [SI+$40], $0001      $814C400100
SHL [BX], 5             $C02705          // Assumes bx:1e from previous
CMP [BX], $03c0         $813FC003
JE $05                  $7405
OR [SI+$40], $0002      $814C400200
                                // Tests SHR
MOV AX, $0F             $B00F
SHR AX, 1;              $D0E8
CMP AX, $07             $3C07
JE $05                  $7405
OR [SI+$44], $0001      $814C440100
SHR AX, 5               $C0E805          // Assumes ax:$07 from previous
CMP AX, $00             $3C00
JE $05                  $7405
OR [SI+$44], $0002      $814C440200
                                // Tests CMP
MOV BX, $1000           $BB0010
MOV [BX], $64           $C60764
MOV AX, $64             $B064
CMP [BX], AX            $3807
JE $05                  $7405
OR [SI+$48], $0001      $814C480100
CMP AX, $64             $3C64
JE $05                  $7405
OR [SI+$48], $0004      $814C480400
                                // Tests JMP
JMP $05                 $EB05
OR [SI+$50], $0001      $814C500100
JMP $0005               $E90500
OR [SI+$50], $0002      $814C500200
MOV [BX], $05           $C60705
JMP [BX]                $FF27
OR [SI+$50], $0004      $814C500400
                                // Tests JCond
MOV AX, $A7             $B0A7            // 2's Comp
JS $05                  $7805
OR [SI+$54], $0001      $814C540100
MOV AX, $07             $B007
JPO $05                 $7B05
OR [SI+$54], $0002      $814C540200
```

Testcode

```
MOV AX, $0F              $B00F
JPE $05                  $7A05
OR [SI+$54], $0004       $814C540400
CMP AX, $0F              $3C0F
JGE $0005                $0F8D0500
OR [SI+$54], $0008       $814C540800
CMP AX, $0E              $3C0E
JLE $0005                $0F8E0500
OR [SI+$54], $0010       $814C541000
CMP AX, $12              $3C12
JNE $0005                $0F850500
OR [SI+$54], $0020       $814C542000
                                 // Tests the test procedure $00ac = OK
MOV SI, $5000            $BE0050             // Test flag storage location
CMP SI, $5000            $81FE0050           // Verify CMP when equal
JE $05                   $7405               // should jump
OR [SI+$58], $00FF       $814C58FF00         // Changes flag test value
CMP SI, $5001            $81FE0150           // Verify CMP when not equal
JE $05                   $7405               // should not jump
OR [SI+$58], $00A0       $814C58A000         // puts $ac in $5058 (test flag)
                                 // Fibinocci Sequence
MOV SI, $50              $B650
MOV AX, $01              $B001
MOV [SI], AX             $8904
MOV [SI+04], AX          $894404             // enter sed values into memory
MOV AX, [SI]             $8B04
MOV BX, [SI+04]          $8B5C04
ADD AX, BX               $01D8
MOV [SI+08], AX          $894408
ADD SI, $04              $82C604             // increment memory location
CMP SI, $0064            $83FE6400           // check for end of loop
JL -19                   $7CED               // return to fibinocci loop
MOV AX, $DEAD            $B8ADDE             // writes $DEAD to memory after
MOV [SI+$OC], AX         $89440C
NOP                      $90
NOP                      $90
                                 // End of Tests
```

Rtest.v

```
module rtest;
reg [31:2] addr;
reg [31:0] in;
reg [4:0] bcdef;
reg blast, ads;
wire clk, brdy;
wire [31:0] out;
reg [3:0] test;

clock C1 (clk);
ram R1 (addr, in, out, bcdef, blast, brdy, ads);

initial begin
 ads=1;
 test = 0;
 blast=1;
 #30 $finish;
 end

initial begin
 #3;
 bcdef = 31;  //write section
 @(posedge clk);
 addr = 'b1000;
 in = 1;
 ads=0;
 @(negedge brdy);
 @(posedge clk);
 addr = 'b1100;
 in = 2;
 @(negedge brdy);
 @(posedge clk);
 blast=0;
 addr = 'b10000;
 in = 3;
 @(negedge brdy);
 @(posedge clk);
 ads=1;
 @(posedge brdy);
 blast = 1;
 bcdef = 27;  //read from memory
 @(posedge clk);
```

```
  addr = 'b1000;
  ads=0;
  @(negedge brdy) if (out != 1) test[0] = 1;
  @(posedge clk);
  addr = 'b1100;
  @(negedge brdy) if (out != 2) test[1] = 1;
  @(posedge clk);
  blast=0;
  addr = 'b10000;
  @(negedge brdy) if (out != 3) test[2] = 1;
  @(posedge clk);
  ads = 1;
  @(posedge brdy) if (out != 32'bx) test[3] = 1;
  blast=1;
  if (!test) $display ("RAM tests OK");
  else $display ("RAM tests FAILED : %b", test);
  end
endmodule
```

Ptest.v

```
module ptest;
reg [31:0] datain, dataout;
reg [31:2] aout;
reg [3:0] dpin, benable;
reg [4:0] bcdef;
reg [1:0] bsize;
reg ads, rdy;
wire [3:0] dpout;
wire parity;
wire clk;
reg [4:0] test;

clock C1 (clk);
pchk P1 (dpin,dpout,parity,datain,dataout,benable,bsize,bcdef,ads,rdy,
        aout,clk);

initial begin
  test = 0;
  #100 $finish;
  end

initial begin
  #3;
  aout = 'b1010;
  datain = 987654321;
  dpin = 2;
  benable = 0;
  bcdef = 'b1;
  bsize = 3;
  rdy=0;
  #3 if (parity != 0) test[0] = 1;
  rdy=1;
  #6;
  aout = 'b1011;
  dpin = 1;
  rdy=0;
  #3 if (parity != 1) test[1] = 1;
  rdy=1;
  #6;
  aout = 'b1010;
  dataout = 1234567890;
  bcdef = 'b111;
```

```
 ads=0;
 #1 if (dpout != 'b1000 || parity != 1) test[2] = 1;
 #1 ads=1;
 #6;
 aout = 'b1011;
 dataout = 987654321;
 bcdef = 'b110;
 ads=0;
 #1 if (dpout != 'b0001 || parity != 1) test[3] = 1;
 #1 ads=1;
 #6;
 bsize = 1;
 aout = 'b1100;
 dataout = 12592;
 ads=0;
 #1 if (dpout != 'b0010 || parity != 1) test[4] = 1;
 #1 ads=1;
 if (!test) $display ("Parity tests OK");
 else $display ("Parity tests FAILED: %b",test);
end

endmodule
```

Pagetest.v

```
module pagetest;
reg [31:0] segaddr;
reg [15:0] csreg;
reg xlate;
wire [31:0] linaddr;
wire page;
reg [1:0] test;

clock C1 (clk);
paging P1 (segaddr, linaddr, xlate, csreg, clk, page);

initial begin
  test = 0;
  xlate = 1;
  #50 $finish;
end

initial begin
  #3 csreg = 0;
  segaddr = 'h1000;
  xlate = 0;
  @(posedge clk) xlate = 1;
  if (linaddr != 'h1000) test[0] = 1;
  @(posedge clk) csreg = 'h0010;
  segaddr = 'h1000;
  xlate = 0;
  @(posedge clk) xlate = 1;
  if (linaddr != 'h1040) test[1] = 1;
  if (!test) $display ("Paging tests OK");
  else $display ("Paging tests FAILED: %b", test);
end

endmodule
```

Segtest.v

```
module segtest;
reg [31:0] in;
reg [15:0] segregs, disp;
reg xlate;
wire [31:0] out;
wire page;
reg [3:0] test;

clock C1 (clk);
segmentation S1 (in, out, xlate, segregs, clk, page, disp);

initial begin
  test = 0;
  disp = 0;
  xlate = 0;
  #100 $finish;
end

initial begin
  #3 segregs = 0;
  in = 'h1000;
  xlate = 1;
  @(posedge clk) xlate = 0;
  if (out != 'h1000) test[0] = 1;
  @(posedge clk) segregs = 'h0010;
  in = 'h1000;
  xlate = 1;
  @(posedge clk) xlate = 0;
  if (out != 'h1040) test[1] = 1;
  @(posedge clk) segregs = 'h0010;
  in = 'h1000;
  disp = 'h32;
  xlate = 1;
  @(posedge clk) xlate = 0;
  if (out != 'h1072) test[2] = 1;
  @(posedge clk) disp = 'hffce;
  xlate = 1;
  @(posedge clk) xlate = 0;
  if (out != 'h100e) test[3] = 1;
  if (!test) $display ("Segmentation tests OK");
  else $display ("Segmentation tests FAILED: %b", test);
end
```

endmodule

Flagtest.v

```
module flagtest;
reg setflags, set;
reg [15:0] value;
wire [15:0] flagreg;
wire fdone;
reg [3:0] test;

clock C1 (clk);
flag F1 (setflags, value, flagreg, set, fdone);

initial begin
  setflags = 0;
  set = 0;
  test = 0;
  #100 $finish;
end

initial begin
  #3;
  value = 0;
  setflags = 1;
  set = 1;
  @(fdone) setflags = 0;
  set = 0;
  if (flagreg[2] !=1 || flagreg[6] !=1 || flagreg[7] !=0) test[0] = 1;
  @(posedge clk);
  value = 'h1000;
  setflags = 1;
  set = 0;
  @(fdone) setflags = 0;
  set = 0;
  if (flagreg[2] !=1 || flagreg[6] !=1 || flagreg[7] !=0) test[1] = 1;
  @(posedge clk);
  value = 'h1000;
  setflags = 1;
  set = 1;
  @(fdone) setflags = 0;
  set = 0;
  if (flagreg[2] !=0 || flagreg[6] !=0 || flagreg[7] !=0) test[2] = 1;
  @(posedge clk);
  value = 'ha000;
  setflags = 1;
```

```
  set = 1;
  @(fdone) setflags = 0;
  set = 0;
  if (flagreg[2] !=1 || flagreg[6] !=0 || flagreg[7] !=1) test[3] = 1;
  @(posedge clk);
  if (!test) $display ("Flag tests OK");
  else $display ("Flag tests FAILED: %b",test);
end
endmodule
```

Addrtest.v

```
module addrtest;
reg load, count;
reg [31:0] pcin;
wire [31:0] eip;
reg [2:0] test;

clock C1 (clk);
address A1 (load, count, pcin, eip, clk);

initial begin
  load = 0;
  count = 0;
  test = 0;
  #50 $finish;
end

initial begin
  #3;
  pcin = 'h1000;
  load = 1;
  @(negedge clk) load = 0;
  if (eip != 'h1000) test[0] = 1;
  @(negedge clk) count = 1;
  @(negedge clk) count = 0;
  if (eip != 'h1004) test[1] = 1;
  @(negedge clk);
  if (eip != 'h1004) test[2] = 1;
  if (!test) $display ("Address tests OK");
  else $display ("Address tests FAILED: %b",test);
end
endmodule
```

Shftest.v

```
module shftest;
reg [15:0] in1;
wire [15:0] out;
reg dir, type, fdone;
reg [4:0] count;
wire setflags;
reg [3:0] test;

clock C1 (clk);
shifter S1 (in1, out, dir, type, count, clk, setflags, fdone);

initial begin
  in1 = 0;
  type = 0;
  count = 0;
  dir = 0;
  fdone = 0;
  test = 0;
  #100 $finish;
end

initial begin
  #3;
  in1 = 'b1001;
  count = 2;
  @(posedge setflags) fdone = !fdone;
  if (out != 'b100100) test[0] = 1;
  @(posedge clk);
  in1 = 'b100100;
  dir = 1;
  @(posedge setflags) fdone = !fdone;
  if (out != 'b1001) test[1] = 1;
  @(posedge clk);
  in1 = 'b1001;
  type = 1;
  dir = 1;
  count = 4;
  @(posedge setflags) fdone = !fdone;
  if (out != 'h9000) test[2] = 1;
  @(posedge clk);
  in1 = 'h9000;
  dir = 0;
```

```
  count = 4;
  @(posedge setflags) fdone = !fdone;
  if (out != 'b1001) test[3] = 1;
  @(posedge clk);
  if (!test) $display ("Shifter tests OK");
  else $display ("Shifter tests FAILED: %b",test);
end

endmodule
```

Alutest.v

```
module alutest;
reg [15:0] op1, op2;
reg [2:0] alusupp;
wire [31:0] out;
reg [7:0] instr;
wire setflags;
reg fdone;
reg [12:0] test;

clock C1 (clk);
alu A1 (op1, op2, out, instr, clk, setflags, alusupp, fdone);

initial begin
  instr = 0;
  op1 = 0;
  op2 = 0;
  alusupp = 0;
  fdone = 0;
  test = 0;
  #100 $finish;
end

initial begin
  @(posedge clk);
  op1 = 'h0010;
  @(posedge setflags) fdone = !fdone;
  if (out != 'h0010) test[0] = 1;
  @(posedge clk);
  op1 = 'ha000;
  instr = 1;
  @(posedge setflags) fdone = !fdone;
  if (out != 'hffffa000) test[1] = 1;
  @(posedge clk);
  instr = 2;
  @(posedge setflags) fdone = !fdone;
  if (out != 0) test[2] = 1;
  @(posedge clk);
  op1 = 'h0010;
  op2 = 'h0010;
  alusupp = 'h01;
  instr = 3;
  @(posedge setflags) fdone = !fdone;
```

```
if (out != 'h0020) test[3] = 1;
@(posedge clk);
op1 = 'h0010;
op2 = 'h0008;
instr = 4;
@(posedge setflags) fdone = !fdone;
if (out != 'h0008) test[4] = 1;
@(posedge clk);
op1 = 'b1111;
op2 = 'b1010;
instr = 5;
@(posedge setflags) fdone = !fdone;
if (out != 'b1010) test[5] = 1;
@(posedge clk);
op1 = 'b1010;
op2 = 'b1000;
instr = 6;
@(posedge setflags) fdone = !fdone;
if (out != 'b1010) test[6] = 1;
@(posedge clk);
op1 = 'b1010;
instr = 7;
@(posedge setflags) fdone = !fdone;
if (out != 'hfffffff5) test[7] = 1;
@(posedge clk);
op1 = 'b1111;
op2 = 'b1010;
instr = 8;
@(posedge setflags) fdone = !fdone;
if (out != 'b0101) test[8] = 1;
@(posedge clk);
op1 = 'h0010;
instr = 9;
@(posedge setflags) fdone = !fdone;
if (out != 'hfffffff0) test[9] = 1;
@(posedge clk);
op1 = 'h0010;
op2 = 'h0002;
instr = 10;
@(posedge setflags) fdone = !fdone;
if (out != 'h0020) test[10] = 1;
@(posedge clk);
```

```
  op2 = 'h0010;
  op1 = 'h0003;
  instr = 11;
  @(posedge setflags) fdone = !fdone;
  if (out != 'h0005) test[11] = 1;
  @(posedge clk);
  instr = 12;
  @(posedge setflags) fdone = !fdone;
  if (out != 'h0001) test[12] = 1;
  @(posedge clk);
  if (!test) $display ("ALU test OK");
  else $display ("ALU test FAILED: %b",test);
end

endmodule
```

Btest.v

```
module btest;
reg [31:0] cdin;
reg rdy, a20m, hold, boff, cdread, cdwrite, miss;
reg [1:0] bsize;
reg [31:4] ain;
reg [3:0] dpin;
reg [31:2] a;
wire [31:0] cdout, dout, din;
wire cdav, ads, hlda, breq, blast, brdy, parity, clk;
wire [4:0] bcdef;
wire [3:0] benable, dpout;
wire [31:2] aout;
reg [3:0] test;

clock C1 (clk);
ram R1 (aout, dout, din, bcdef, blast, brdy, ads);
biu B1 (ads, rdy, a20m, bcdef, hold, hlda, boff, breq, brdy, blast,
      bsize, benable, aout, ain, dout, din, clk,
      cdout, cdin, a, cdav, cdread, cdwrite, miss);

initial begin
 #100 $finish;
 end

initial begin   //default values
 test = 0;
 hold = 0;
 boff  = 1;
 bsize  = 3;   //32-bit bus
 cdwrite = 1; //neg assert
 cdread = 1;  //neg assert
 dpin = 0;     //assumes all even parity already
 miss = 0;    // all cache nonmisses
 rdy  = 1;
 end

initial begin
 #3;   //wait for clock to get up to speed (1 1/2 cycles)
 a='b1000; //1000:5
 cdin = 5;
 cdwrite=0;
 @(negedge cdav)  #1 cdwrite=1;
```

```
@(posedge cdav);
a='b1100; //1100:3
cdin = 3;
cdwrite=0;
@(negedge cdav) #1 cdwrite=1;
@(posedge cdav);
a='b10000; //10000:2
cdin = 2;
cdwrite=0;
@(negedge cdav) #1 cdwrite=1;
@(posedge cdav);
a='b10100; //10100:4
cdin = 4;
cdwrite=0;
@(negedge cdav) #1 cdwrite=1;
@(posedge cdav);
a='b11000; //11000:1
cdin  = 1;
cdwrite=0;
@(negedge cdav) #1 cdwrite=1;
@(posedge cdav);
#20;
a='b10000;
cdread=0;
@(negedge cdav);
if (cdout != 2) test[0] = 1;
@(clk) cdread=1;
@(clk);
a='b1100;
cdread=0;
@(negedge cdav);
if (cdout != 3) test[1] = 1;
@(clk) cdread=1;
@(clk);
a='b10100;
cdread=0;
@(negedge cdav);
if (cdout != 4) test[2] = 1;
@(clk) cdread=1;
@(clk);
a='b1000;
cdread=0;
```

```
  @(negedge cdav);
  if (cdout != 5) test[3] = 1;
  @(clk) cdread=1;
  if (!test) $display ("BIU tests OK");
  else $display ("BIU tests FAILED: %b",test);
  end
endmodule
```

Ctest.v

```
module ctest;
reg [31:0] baddr, bdatain;
reg write, read, ahold, eads, rdy, a20m, hold, boff, flush;
reg [31:4] ain;
reg [1:0] bsize;
reg [3:0] dpin;
wire [31:0] cdout, cdin, dout, din, bdataout;
wire cdav, cdread, cdwrite, ads, hlda, breq, blast, brdy, parity,
   bdav, bread, bwrite, clk, iclk, miss, rdone, wdone;
wire [4:0] bcdef;
wire [3:0] benable, dpout;
wire [31:2] aout, a;
reg [10:0] test;

clock T1 (clk);
ram R1 (aout, dout, din, bcdef, blast, brdy, ads);
biu B1 (ads, rdy, a20m, bcdef, hold, hlda, boff, breq, brdy, blast,
      bsize, benable, aout, ain, dout, din, clk,
      cdout, cdin, a, cdav, cdread, cdwrite, miss);
cache C1 (cdout, cdin, a, cdav, cdread, cdwrite, miss, ahold, eads, ain,
       baddr, bdatain, bdataout, bread, bdav, bwrite, flush, clk);
read RD1 (bread, bdataout, bdav, clk, read, rdone);
write WR1 (bwrite, bdav, clk, write, wdone);

initial begin
 #300 $finish;
 end

initial begin  //default values
 test = 0;
 read = 0;
 write = 0;
 ahold = 0;
 eads = 1;
 rdy = 1;
 a20m = 1;
 hold = 0;
 boff = 1;
 flush = 1;
 bsize = 3;    //32-bit bus
 dpin = 0;     //assumes all even parity
 end
```

```
always @(posedge miss) begin
  $display("cache miss");
  end

initial begin   //main test loop
  #3;   //wait for clock to get up to speed
  baddr = 'ha0;
  bdatain = 20;
  @(clk);
  write = 1;
  @(posedge wdone) write = 0;
  baddr = 'hb0;
  bdatain = 30;
  @(clk);
  write = 1;
  @(posedge wdone) write = 0;
  baddr = 'hc0;
  bdatain = 40;
  @(clk);
  write = 1;
  @(posedge wdone) write = 0;
  baddr = 'hd0;
  bdatain = 50;
  @(clk);
  write = 1;
  @(posedge wdone) write = 0;
  baddr = 'ha0;
  @(clk) read = 1;
  @(posedge rdone) if (bdataout != 20) test[0] = 1;
  read=0;
  baddr = 'ha0;
  @(clk) read = 1;
  @(posedge rdone) if (bdataout != 20) test[1] = 1;
  read=0;
  baddr = 'hb0;
  @(clk) read = 1;
  @(posedge rdone) if (bdataout != 30) test[2] = 1;
  read=0;
  baddr = 'hb0;
  @(clk) read = 1;
  @(posedge rdone) if (bdataout != 30) test[3] = 1;
```

```
read=0;
baddr = 'hcc;
@(clk) read = 1;
@(posedge rdone) if (bdataout != 32'bx) test[4] = 1;
read=0;
flush = 0;
#3;   //wait for cache to flush
flush = 1;
baddr = 'hcc;
@(clk) read = 1;
@(posedge rdone) if (bdataout != 32'bx) test[5] = 1;
read=0;
baddr = 'hc0;
@(clk) read = 1;
@(posedge rdone) if (bdataout != 40) test[6] = 1;
read=0;
baddr = 'hb0;
@(clk) read = 1;
@(posedge rdone) if (bdataout != 30) test[7] = 1;
read=0;
baddr = 'hb0;
@(clk) read = 1;
@(posedge rdone) if (bdataout != 30) test[8] = 1;
read=0;
baddr = 'hb8;
bdatain = 60;
@(clk);
write = 1;
@(posedge wdone) write = 0;
baddr = 'hb8;
@(clk) read = 1;
@(posedge rdone) if (bdataout != 60) test[9] = 1;
read=0;
$display ("invalidating address $b0");
@(clk) ain = 'hb0;
ahold = 1;   //begin cache invalidation cycle
eads = 0;
@(clk) ahold = 0;
eads = 1;
baddr = 'hb0;
@(clk) read = 1;
@(posedge rdone) if (bdataout != 30) test[10] = 1;
```

```
 read=0;
 if (!test) $display ("Cache tests OK");
 else $display ("Cache tests FAILED: %b",test);
 end
endmodule

module read(bread, bdataout, bdav, clk, read, rdone);
output bread, rdone;
input clk, read, bdav;
input [31:0] bdataout;
reg bread, rdone;

initial begin
 bread = 1;
 rdone = 0;
end

always @(posedge read) begin
 bread = 0;   //addr already valid
 @(negedge bdav) rdone = 1;
 @(negedge read) bread = 1;
 rdone = 0;
 end
endmodule

module write(bwrite, bdav, clk, write, wdone);
output bwrite, wdone;
input bdav, write, clk;
reg bwrite, wdone;

initial begin
 bwrite = 1;
 wdone = 0;
 end

always @(posedge write) begin
 bwrite = 0;  //data and addr already ready
 @(negedge bdav) wdone = 1;
 @(negedge write) bwrite = 1;
 wdone = 0;
 end
endmodule
```

Pftest.v

```
module pftest;
reg [31:0] baddr_reg, bdatain, linaddr, pc;
reg write, read, ahold, eads, rdy, a20m, hold, boff, flush, dnext,
   pfflush, page, free;
reg [31:4] ain;
reg [1:0] bsize;
reg [3:0] dpin;
wire [31:0] baddr=baddr_reg;
wire [31:0] cdout, cdin, dout, din, bdataout, ddata, segaddr;
wire cdav, cdread, cdwrite, ads, hlda, breq, blast, brdy, parity,
    bdav, bread, bwrite, clk, iclk, miss, rdone, wdone, ddav, xlate, get;
wire [4:0] bcdef;
wire [3:0] benable, dpout;
wire [31:2] aout, a;
reg [9:0] test;

clock T1 (clk);
ram R1 (aout, dout, din, bcdef, blast, brdy, ads);
pchk P1 (dpin, dpout, parity, din, dout, benable, bsize, bcdef, ads, brdy,
        aout, clk);
biu B1 (ads, rdy, a20m, bcdef, hold, hlda, boff, breq, brdy, blast,
       bsize, benable, aout, ain, dout, din, clk,
       cdout, cdin, a, cdav, cdread, cdwrite, miss);
cache C1 (cdout, cdin, a, cdav, cdread, cdwrite, miss, ahold, eads, ain,
        baddr, bdatain, bdataout, bread, bdav, bwrite, flush, clk);
write WR1 (bwrite, bdav, clk, write, wdone);
prefetch PF1 (ddav, dnext, ddata, segaddr, linaddr, xlate, page, get,
          pfflush, pc, free, bdav, bread, bdataout, baddr, clk);

initial begin
 #300 $finish;
 end

initial begin   //default values
 test = 0;
 free = 0;
 read = 0;
 write = 0;
 ahold = 0;
 eads = 1;
 rdy = 1;
 a20m = 1;
```

```
  hold = 0;
  boff = 1;
  flush = 1;
  bsize = 3;    //32-bit bus
  dpin = 0;     //assumes all even parity
  pfflush = 0;
  dnext = 1;
  page = 1;
  pc = 'ha0;
  baddr_reg = 32'bz;
  end

always @(negedge xlate) begin     //simulated paging unit
  linaddr = segaddr;  //no segmentation translation
  end

always @(get) begin
  if (get) free = 0;
  else free = 1;
end

initial begin   //main test loop
  #3;   //wait for clock to get up to speed
  $display("writing values to cache");
  baddr_reg = 'ha0;   //write ten values into memory starting $a0
  bdatain = 0;
  @(clk) write = 1;
  @(posedge wdone) write = 0;
  baddr_reg = 'ha4;
  bdatain = 1;
  @(clk) write = 1;
  @(posedge wdone) write = 0;
  baddr_reg = 'ha8;
  bdatain = 2;
  @(clk) write = 1;
  @(posedge wdone) write = 0;
  baddr_reg = 'hac;
  bdatain = 3;
  @(clk) write = 1;
  @(posedge wdone) write = 0;
  baddr_reg = 'hb0;
  bdatain = 4;
```

```
@(clk) write = 1;
@(posedge wdone) write = 0;
baddr_reg = 'hb4;
bdatain = 5;
@(clk) write = 1;
@(posedge wdone) write = 0;
baddr_reg = 'hb8;
bdatain = 6;
@(clk) write = 1;
@(posedge wdone) write = 0;
baddr_reg = 'hbc;
bdatain = 7;
@(clk) write = 1;
@(posedge wdone) write = 0;
baddr_reg = 'hc0;
bdatain = 8;
@(clk) write = 1;
@(posedge wdone) write = 0;
baddr_reg = 'hc4;
bdatain = 9;
@(clk) write = 1;
@(posedge wdone) write = 0;
baddr_reg = 32'bz;
pfflush = 1; //flush prefetch so as to reset program counter
@(posedge clk) pfflush = 0;
free = 1;
#50; //wait for prefetch to work
@(clk) dnext = 0;
@(negedge ddav) if (ddata != 0) test[0] = 1;
dnext = 1;
@(clk) dnext = 0;
@(negedge ddav) if (ddata != 1) test[1] = 1;
dnext = 1;
@(clk) dnext = 0;
@(negedge ddav) if (ddata != 2) test[2] = 1;
dnext = 1;
@(clk) dnext = 0;
@(negedge ddav) if (ddata != 3) test[3] = 1;
dnext = 1;
@(clk) dnext = 0;
@(negedge ddav) if (ddata != 4) test[4] = 1;
dnext = 1;
```

```
  @(clk) dnext = 0;
  @(negedge ddav) if (ddata != 5) test[5] = 1;
  dnext = 1;
  @(clk) dnext = 0;
  @(negedge ddav) if (ddata != 6) test[6] = 1;
  dnext = 1;
  @(clk) dnext = 0;
  @(negedge ddav) if (ddata != 7) test[7] = 1;
  dnext = 1;
  @(clk) dnext = 0;
  @(negedge ddav) if (ddata != 8) test[8] = 1;
  dnext = 1;
  @(clk) dnext = 0;
  @(negedge ddav) if (ddata != 9) test[9] = 1;
  dnext = 1;
  #10 free = 0;
  if (!test) $display ("Prefetch tests OK");
  else $display ("Prefetch tests FAILED: %b", test);
  end
endmodule

module write(bwrite, bdav, clk, write, wdone);
output bwrite, wdone;
input bdav, write, clk;
reg bwrite, wdone;

initial begin
  bwrite = 1;
  wdone = 0;
  end

always @(posedge write) begin
  bwrite = 0;  //data and addr already ready
  @(negedge bdav) wdone = 1;
  @(negedge write) bwrite = 1;
  wdone = 0;
  end
endmodule
```

Analysis

```
cp rmake Makefile
make
cp pmake Makefile
make
cp pagemake Makefile
make
cp segmake Makefile
make
cp flagmake Makefile
make
cp addrmake Makefile
make
cp shiftmake Makefile
make
cp alumake Makefile
make
cp bmake Makefile
make
cp cmake Makefile
make
cp pfmake Makefile
make
cp totalmake Makefile
make
```

Appendix 5

References

1. **Intel486 DX Microprocessor Data Book**, Intel Corporation, 1991

2. **Computer Architecture: A Quantitative Approach**, John Hennessy and David Patterson, Morgan Kaufmann Publishers, San Mateo, CA, 1990

3. **Digital Design with Verilog HDL**, Eliezer Sternheim, Rajvir Singh, and Yatin Trivedi, Automata Publishing Company, Cupertino, CA, 1990

4. **Microsoft's 80386/ 80486 Programming Guide**, Ross Nelson, Microsoft Press, Redmond, WA, 1991

5. **Computation Structures**, Stephen Ward and Robert Halstead Jr., MIT Press, Cambridge, MA, 1990

6. **Intel486 Microprocessor Family: Programmer's Reference Manual**, Intel Corporation, 1992

7. **PC Magazine, The Programmer's Technical Reference: The Processor and Coprocessor**, Robert Hummel, Ziff-Davis Press, Emeryville, CA 1992