Z
699
C3
no. 92-78

VIPER: A 25-MHz, 100-MIPS Peak VLIW Microprocessor

J. Gray, A. Naylor, A. Abnous, N. Bagherzadeh

Department of Electrical and Computer Engineering
Department of Information and Computer Science

University of California, Irvine

Irvine, California 92717

Technical Report No. 92-78

VIPER: A 25-MHz, 100-MIPS Peak VLIW Microprocessor

Jeffrey Gray, Andrew Naylor, Arthur Abnous,
and Nader Bagherzadeh
Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92717
(714) 856-8720

Abstract

This paper describes the design and implementation of a very long instruction word (VLIW) microprocessor. The VIPER (VLIW integer processor) contains four pipelined functional units, and can achieve 100 MIPS peak performance at 25 MHz. The processor is capable of performing multiway branch operations, two load/store operations and up to four ALU operations in each clock cycle, with full register file access to each functional unit. VIPER is the first VLIW microprocessor known that can achieve this level of performance. Designed in twelve months, the processor is integrated with an instruction cache controller and a data cache, requiring 450,000 transistors and a die size of 12.9 by 9.1 mm in a 1.2 μm technology.

1 Introduction

The VLIW architecture is considered to be one of the promising methods of increasing performance beyond standard RISC architectures. While RISC architectures take advantage of only temporal parallelism by using pipelined functional units, VLIW architectures can also take advantage of spatial parallelism by using multiple functional units to execute several operations concurrently. Similar to superscalar architectures, the VLIW architecture can reduce the cycles per instruction (CPI) factor with this added parallelism. Superscalar processors schedule the execution order of the operations at run time. They demand more hardware support in order to manage synchronization among concurrent operations. Since VLIW machines schedule operations at compile time, allowing global code optimization, they tend to have relatively simple control paths–following the RISC methodology.

Although the idea of the VLIW architecture had been known for years, there were no real implementations of such architectures, mainly due to the lack of compilation tools for such machines. Recently, however, with rapid advances in compiler optimization techniques [1], this has changed. These new fine-grain compilation techniques compute a static parallel schedule from an originally sequential program. Percolation scheduling (PS) is one of these promising techniques, and a PS compiler was developed as a parallel effort to the VLSI design of the architecture. In recent years, there have been several efforts to design and develop VLIW architectures [2], [3]. The LIFE processor [4] is the only other VLIW microprocessor that has been implemented; however, the VIPER and LIFE processors differ in the treatment of the register file, the compilation strategy, and the number of ALU and load/store units.

The VIPER processor is well-suited for embedded processing applications, as well as general purpose computing. Due to its efficient VLSI implementation, the processor core could easily be integrated with other subsystems of an application-specific controller on a single chip. This paper presents a four-processor version of the VIPER architecture, integrated with data and instruction caches for general purpose use. However, one could foresee future tools designed to tune the configurational parameters of the VIPER architecture for specific applications, such as embedded controllers for HDTV or multimedia applications.

Section 2 provides an overview of the architecture of the VIPER, describing the basic functional blocks of the processor. Section 3 examines the design of the processor core in greater detail, while the designs of the instruction and data caches are described in Sections 4 and 5. Finally, we provide a brief description of the external interface of the chip in Section 6 and our design verification methodology in Section 7.

2 The Processor Architecture

The VIPER architecture has been designed to reflect the execution model assumed by the PS compiler, which includes support for multi-way branching and speculative execution. Detailed discussion of architectural design issues and analysis of hardware/software trade-offs can be found in [5]. Feasibility and efficiency considerations for a VLSI implementation of the processor architecture led to the current configuration, which includes four functional units and provides an engine that can take advantage of the parallelizing capabilities of the PS compiler.

The organization of the processor was developed to facilitate an efficient VLSI implementation. A key aspect was an emphasis on locality of communication between different blocks of the processor. This is reflected in Figure 1 which illustrates the block diagram of the VIPER. The processor contains four integer functional units connected through a shared eight-port register file. The processor control is pipelined and can issue four operations packed into a very long instruction word in each clock cycle. The pipeline structure of the processor has been designed to reduce the frequency of pipeline hazards and to simplify the bypassing mechanism [5]. The pipeline structure of the VIPER is shown in Figure 2. The control transfer units in FU0 and FU1 control the instruction fetching sequence and can execute multiway branch operations, while the load/store units in FU2 and FU3 establish a connection to a dual-port data cache subsystem. The processor has some of the typical attributes of a pipelined RISC processor, with a simple instruction set that is designed with efficient pipelining and decoding in mind. All instructions have a fixed size with only a few instruction formats, and all follow the register-to-register execution model. Arithmetic, logic, and shift operations can be executed by all of the functional units. In the VIPER architecture, load/store operations use the register indirect addressing mode instead of the more typical displacement addressing mode. Also, to increase execution throughput, the processor performs speculative execution of the operations in the branch and branch delay slots.

3 Processor Core Design

3.1 Operation Timing

The architecture of the VIPER requires overlapped writing and reading of operands to and from the register file within one instruction cycle. This requires a clocking scheme that can support the subdivision of the instruction cycle into distinct read and write phases. In addition, it can be seen in Figure 2 that a register must be able to be written to in the first part of the WB stage for instruction A, so that the same data can be read in the ID stage for instruction C. This will result in the elimination of one level of bypassing, and a more efficient VLSI implementation. For these reasons, the instruction cycle was partitioned into four phases. Table 1 shows the low-level operations of the VIPER and the clock phases to which they are assigned.

The four-phase clock drivers were designed to produce non-overlapping clock signals with a minimal amount of dead time between phases. The design used feedback from the previous phase's buffer chain to qualify assertion of the following phase. By carefully tuning the timing of the buffer chains to the load capacitances, the dead time was reduced to an average of 2 ns in a 10 ns phase. Thus, clocked circuits were designed to meet the criterion of an 8 ns-wide pulse. A total of twenty global clock signals are provided by the clock drivers. These are the four phases plus an additional signal which is active throughout phases 1, 2, and 3, used by many of the dynamic circuits in the data paths. Global complements of all these clock signals are also generated. In addition, separate identical clocks are provided to the processor units and the caches, with the difference that the processor clocks can be stopped in the event of a cache miss.

3.2 Register File Design

The eight-port register file is organized much like a general-purpose RAM with a row decoding scheme, where all address bits are used to specify a single row in the memory array. The core consists of 32-by-32 array of memory elements. Because of the architectural specification that register R0 always contain the value zero, then strictly speaking only 31 of the rows contain functional memory elements, with one row hard-wired to a zero value. The decoders are located on the left and right sides of the memory array and consist of four sets of NAND-type decoders on each side, as a total of eight different register specifiers must be decoded simultaneously during the operand read phases. Located above and below the register array are the bit line drivers, which force values into the registers during a write operation and act as primitive sense amps during the read cycle. The organization of the register file is shown in Figure 3, with two example cell access paths highlighted for register accesses on FU0 and FU2. The highlighted regions show the path from register specifier inputs to the decoders, then to the register cell rows via the word lines and finally out the bit lines and to the functional units.

Each register cell consists of a pair of cross-coupled inverters with four access transistors attaching the output of each inverter to a separate bit line (see Figure 4). The gates of the eight access transistors are each connected to a separate word line; when these lines are driven high by the row decoders, the access transistors are turned on and the bit lines become electrically connected to the inverter pair, allowing a read or write to occur.

A read operation consists of a precharge and decode phase on clock phase 2 followed by a discharge reading phase on phase 3. It is possible that all four access transistors on one side of the register cell could be turned on if four functional units require the same operand. In this worst case, the total capacitive load of four bit lines will be discharging through one n-transistor of the inverter pair, causing a voltage surge which will also be seen at the input of the other inverter. If this spike is large enough, it could cause the inverter pair to change its state, destroying the contents of the register. To prevent this, the n-transistors of the inverters are oversized with the effect that the amplitude of the voltage spike will be smaller due to the increased conductance of the n-transistor. Simulations in SPICE showed that the precharge time of the bit lines is 2.9 ns, and the worst-case discharge of four bit lines through one inverter required 4.2 ns and caused a voltage spike of 0.6 V, which is sufficiently small to prevent disturbance of the state of the cell.

The write cycle begins with the decoding of the destination register specifiers on phase 4. By the end of phase 4, the destination busses have been driven and latched, and the writing commences at the start of phase 1. The data in the destination latch and its complement are driven onto bit lines on opposite sides of the inverter pair, and the decoded word lines are activated, turning on the access transistors. Simulation in SPICE showed that the state of the cell was changed 1.2 ns after the access transistors are turned on. In addition, the decoder arrays are capable of decoding the register specifier in 3.6 ns. Thus the low-level operations of the register file are fast enough to provide a nominal read bandwidth of 12.8 Gb/s, which is more than sufficient for the required read bandwidth of 6.4 Gb/s for the target speed of 25 MHz.

3.3 Data Paths

The VIPER contains four identical 32-bit data paths which are capable of executing all of the arithmetic and logic functions in the instruction set. Each data path consists of three major blocks:

1. an operand unit, which provides an interface between the data path and the register file and handles the switching functions necessary to implement the operand bypassing scheme used by the VIPER architecture;

2. an arithmetic/logic unit, which is capable of integer addition and subtraction, numerical comparison functions, and logic function computation;

3. a barrel shifter, which can execute logical and arithmetic shift operations of an arbitrary number of bit positions in a single cycle.

In addition, the data paths for functional units FU2 and FU3 include the cache interface circuitry, used for load and store operations.

Each data path is implemented using two source busses that provide operands to the data path blocks, and one destination bus that carries the result of the computation back to the register file. As shown in Figure 5, the operand unit physically separates the source busses from the register file. The operand unit latches the output of the register file on each instruction cycle and then conditionally provides this data onto the source busses. In the event that a bypass condition has been detected, indicating that the register file value is not valid, the operand unit will instead provide the data from the destination bus of the functional unit that is the source of the valid data. Because the bypassing scheme in the VIPER restricts bypassing from one functional unit to only itself and its nearest neighbor, the operand unit needs to accept input from two bypassing paths. In addition, it is possible that the instruction requires a 16-bit immediate value as the second operand instead of a register value, hence the operand unit also takes an immediate bus from the control path as an input. The switching function is implemented as n-type pass transistors for reasons of speed and compact physical design (see Figure 6).

The timing of the operand unit is determined by the four-phase nature of the register file. The operand unit must latch the register values on phase 3, and perform the bypass and immediate operand switching on phase 4. The output of this network is driven onto the two source busses and must be stable before phase 1 begins, as many of the data path elements are dynamic circuits that evaluate during phase 123. If the source busses have not stabilized by the rising edge of phase 123, some of the dynamic circuits could falsely discharge and cause erroneous results.

The arithmetic/logic unit is composed of a Weinberger PG generator and a Manchester carry-chain. Individual reusable elements, such as latches and multiplexers, as well as more specific blocks like the PG generator and the carry chain, were all designed in a common bit-sliced style. These circuits were previously designed for a 16-bit prototype of the VIPER's data path [7]. After extending the elements to a 32-bit width, the data paths were assembled by stacking the physical blocks vertically. The block diagram of the data paths is shown in Figure 5, along with the associated physical floorplan of functional units FU0 and FU1.

3.4 Program Counter Unit Design

The program counter unit is responsible for all computation related to the control of the instruction flow. It contains the program counter register, adders to compute branch target addresses, and special trap vector generation hardware. Because the VIPER's long instruction words are aligned on addresses which are multiples of 16, the PC hardware only needs to be 28 bits wide. Control for the PC unit resides in the control for FU0, as all instructions containing a control transfer will have a control transfer operation scheduled on FU0.

The basic organization of the PC unit is shown in the block diagram in Figure 7. In order to implement the VIPER's three-way branch, simultaneous computation of three different target addresses is necessary. These three adders represent the majority of the functionality contained in the PC unit. Three special registers are used to store the current value of the program counter (labeled PC), the address to be returned to following an interrupt (RIPC), and the address to be returned to following a trap exception (RTPC).

Because the addition of branch delay slots has a detrimental effect on performance, the address of the next instruction must be available by the end of the instruction decode (ID) pipeline stage. Hence the computation of the possible target addresses occurs during phase 123 of the ID stage. The branch conditions, however, are not available until phase 4. In the case that the conditions are in the register file at the time of the branch, then evaluation of the branch condition can commence once the source busses have stabilized. In the case that the branch conditions are computed by the previous instruction (and must be bypassed) the source busses will take even longer to resolve. Once the branch conditions are available, then the control logic can begin to arbitrate the branch, requiring additional time before the end of phase 4. This case determines the critical path of the processor, as the pulse width of clock phase 4 must be wide enough to accomplish everything described above, therefore determining the maximum clock frequency. The PC can then be latched on phase 4 and driven to the instruction cache.

Figure 8 shows the simulation of this critical path. Signal `s1_0a` is the branch condition, and after it resolves to zero in the middle of phase 4 (shown as `pcclk`), the `selt0` signal is asserted, indicating a taken branch. By this time, computation of the next PC has completed (`PC + ofa` = 0000006H + FFFFFCH = 000002H), which is latched by the falling edge of `pcclk` and driven to the instruction cache controller.

3.5 Control Design

The control logic for the VIPER is distributed into four blocks, each of which is tightly coupled to the functional unit with which it is associated. Each control path is implemented as three instruction registers, corresponding to the ID, EX, and WB pipeline stages, with combinational logic between registers to perform decoding and other control functions. The control paths were designed in standard cells using the Berkeley Octtools system [8]. The overall approach for designing the control logic was to partition the function into blocks which could be separately described in a higher-level behavioral language, and then synthesized and recombined into one physical block. Partitioning for the instruction decoding was performed by grouping together control signals with common functions (for example, all ALU-related control signals), and assigning them as outputs of one functional block. In essence this partitioning also divides decoding by instruction class (computation, control transfer, or load/store), which allows the blocks to be included in the control for only the functional units that require it. The instruction bits then serve as inputs to many of these blocks. Some advantages of this partitioning is that it reduces the amount of logic that must be considered simultaneously by the logic synthesis tool, and that each block may be individually tested for correctness, reducing the complexity of debugging. A possible disadvantage of this system is that the logic synthesizer might be capable of minimizing the gate count further if all behavior were described and synthesized together. Each control path was assembled by including the synthesized blocks needed for that functional unit and implementing the instruction registers as arrays of latches. The four blocks were then placed and routed into four rows of standard cells each in order to achieve an aspect ratio consistent with the floorplan shown in Figure 1. It is worth noting that over 40% of all cells used are dedicated to bypass detection.

4 Instruction Cache

One problem common to VLIW processors is that of code expansion. Since the compiler can not always schedule four operations in every cycle, then the unused operation slots will have to be filled with NOP's. In addition, many of the techniques used by the compiler to exploit parallelism, such as loop unrolling and software pipelining, will also cause the code size to grow. This code expansion could have a detrimental effect on the locality of the code, which would affect the cache performance. When combined with the increased width of the instruction word, this effect requires that the instruction cache be somewhat larger than a comparable uniprocessor cache. For these reasons, the instruction cache controller was included on the same die as the processor, with the intent that the actual instruction cache RAM be implemented in external memory.

4.1 Organization

The largest physical burden that the VIPER puts on its instruction memory is the 128-bit width of the instruction word. In order to fetch an instruction in a single cycle, the read bandwidth of the cache must be four times greater than that of a single processor machine. Implementing this wider memory with an on-chip instruction cache does greatly impact the die area, as a cache containing the same number of words as a uniprocessor implementation will have to be four times as large. Preliminary area estimates showed that the largest possible instruction cache feasible for a reasonable chip size would be about 8K bytes. The area required for this size of a cache would be about twice that of all four functional units combined, and a cache of this size would only hold 512 long instruction words. Studies have shown that the expected hit ratio for a cache of this size would be approximately 75% [9, 10]. This would have a serious impact on the peak performance of the VIPER. Recent research has shown that much larger off-chip instruction caches can be built out of standard external SRAM that will meet the same cycle time as an on-chip cache [11]. The above justification for external RAM does not apply to the tag RAM or the cache controller, however. After the tag is read out of the tag memory, it still has to be compared with the tag of the pending instruction. The result of this comparison is then used by the cache controller to decide what the next state of the processor will be. The time required for these additional operations (along with extra chip boundary crossings) makes it difficult to use off-chip RAM for the tag memory without violating the target cycle time. Furthermore, since the tag RAM is small compared to the instruction memory, it can easily be included on the die with the processor.

The configuration of the cache is direct-mapped with 512 blocks of 32 words each. The direct-mapped approach was chosen for several reasons. First, the hit rate for direct-mapped caches approaches that of caches with more associativity when the size of the cache becomes large [12]. Second, one side effect of higher degrees of associativity is that the memory must be made wider, since N banks of memory are required to implement N-way set associativity. Since the VLIW already requires a fairly wide memory structure, this would merely compound the problem. Finally, the outputs of the separate sets would have to be multiplexed together in order to supply the correct data to the processor. The choice of 32 words for a block size does yield a slightly lower hit rate than a smaller block size would, but was chosen in order to keep the tag RAM at a reasonable size. For a block size of 32 words, the tag RAM is fairly large at almost 1K bytes. The blocks that make up the instruction cache are illustrated in Figure 1.

The instruction tag RAM is a 512 word X 15 bit static ram with a built-in comparator function on fourteen of the bits (the tag bits) and one resettable bit (the valid bit). The RAM is partitioned into 64 rows and 120 columns. The global layout has the memory core split vertically into two halves, with the row decoders and drivers placed between the halves. The bit line prechargers are placed on top of the memory cores, and the column multiplexors, tag latches, write drivers, and comparator circuitry are situated on the bottom. The global control logic is located under the row decoders in the center of the RAM.

4.2 Cache Controller and State Machine

A state machine containing six states was used to control the different modes of cache operation. The state diagram appears in Figure 9. A brief description of each of the states follows:

- *NORM*: Normal mode. The address on the PC bus is driven off-chip, while the tag is compared with that in the tag memory.
- *HOLD*: Hold state. The instruction cache will cycle in this state while waiting for the data cache to service a miss.
- *MS*: Miss-Start state. The controller enters this state for one cycle during a cache miss when it sends an instruction address to main memory.
- *MW*: Miss-Wait state. The controller cycles in this state while waiting for the instruction to return from main memory.
- *NCS*: No-Cache-Start. Similar to the *MS* state, only occurring during a cache bypass operation. (The bypass operation is described below.)
- *NCW*: No-Cache-Wait. Similar to the *MW* state, only occurring during a cache bypass operation.

If the controller detects a cache miss while in the *NORM* state, then it will send a stall signal to the clock controller and enter the *MS* state. The block-fill procedure was kept simple to ease the synchronization with the data paths and the data cache. The state machine will cycle between *MS* and *MW* 32 times while the block addresses are sequenced. The instructions from main memory are written into the cache as they return. When the counter matches the instruction requested by the processor, then the instruction pipeline latches the returning data, although the rest of the processor is stalled. Once the block replacement has concluded, the stall signal will be deasserted and the processor will continue operation.

The *NCS* and *NCW* states were added in order to bypass cache replacement when necessary. This will occur at the initiation of an external interrupt servicing routine or a trap instruction, so as to avoid reading unnecessary sections of the operating system's jump table into the cache. This improves processor performance in both of these events. The operation in these two states is similar to that of *MS* and *MW*, except they fetch only one instruction and do not write into the cache. The state machine is updated every cycle on the falling edge of phase 4, but the next state must be determined by phase 3 in order to synchronize the instruction cache controller with the data cache controller and the stall mechanism for the data paths.

5 Data Cache

The data memory subsystem of the VIPER processor must meet several requirements. Since VIPER has two parallel data ports, then the data memory subsystem must be able to service two requests per cycle. The two ports are independent, so both ports are able to load or store data regardless of the operation of the other port. Since load/store instructions are pipelined like any other computation operation, the data cache was designed to perform a read or write in a single processor cycle during the EX pipeline stage.

5.1 Organization

The decision was made to use a dual-port memory scheme in order to service both data ports of the processor independently. This scheme has a performance advantage over other arrangements that could satisfy the multiple data ports. Two other possible schemes would be autonomous memories (each data port is connected to a separate cache) or interleaved caches (one bank of cache holds even memory addresses, while another holds odd addresses). The autonomous scheme suffers from coherency problems, while the performance of the interleaved approach is degraded by address conflicts between the two ports. The dual-port scheme does require special RAM hardware of greater size and complexity. Since the two load-store units are independent, the data RAM must support mixed read and write operations concurrently. The tag RAM must also be dual-ported, and must be able to compare the tags from both ports simultaneously.

The decision to use a dual-ported data cache constrained the cache to be included on-chip for several reasons. While there are dual-port RAM chips available, none have been found that operate at the speed required to meet the target cycle time. In addition, an off-chip cache would require a minimum of two external memory ports, which was unacceptable since the VIPER already requires 160 pins for the instruction cache interface. The fact that the VIPER is pad limited, when coupled with the decision to use an off-chip instruction cache, does leave enough core area for 4K bytes of dual-port RAM on chip. One difficulty with implementing the data cache on-chip was the lack of available RAM generators for dual-ported memory, thus both the data and tag RAM were implemented using full-custom layout. Using the custom on-chip cache allowed use of a single port to main memory. However, in the event that both load/store units have cache misses in the same cycle, then the replacements must be serialized through the single port. This will result in some loss of performance, and an increase in the complexity of the cache controller, but this loss is offset by the smaller cost of the simpler single-port external interface.

Unlike the direct-mapped instruction cache, a two-way set associative scheme was chosen for the data cache. Most of the arguments given for the direct-mapped approach used in the instruction cache don't apply to the data cache. First, while the performance gap between direct-mapped and set-associative caches may be small for large cache sizes, the difference can be substantial for smaller caches. In fact, studies have shown that for caches of the size of the VIPER's data cache, the performance gain of doubling the associativity can be about the same as the performance that would be gained by doubling the cache size [9]. Second, since the cache is on-chip, there is no extra chip boundary crossing necessary to choose between the memory banks. In fact, the multiplexing function between the two sets of data is achieved with tristate drivers on the outputs of the two RAM. Finally, since the data word is only 32 bits wide, then moving to multiple sets does not result in the excessive width problem that would occur with the instruction cache. The cost of this performance is added complexity in both the tag RAM and the cache controller. One added advantage that two-way set associativity gives VIPER is that it makes it easier to resolve collisions between the two ports. These collisions occur when both ports try to access the same cache block, but with different tag values. With a direct-mapped cache, only one of the accesses would be able to go into the cache, and the other would have to be serviced externally. With a two-way set associative cache, each port can access a different bank of data, thus possibly avoiding the external memory reference.

Since all external memory references are made through a single port, it becomes desirable to reduce the traffic on this port as much as possible. Since this port is only accessed during cache misses, then this traffic can be reduced by decreasing either the miss rate or the miss penalty. One way to reduce the miss penalty is to allow sub-block replacement. This was the main motivation for the decision to keep a valid bit for each word in the cache. This allows only one word to be fetched from memory at a time, thus ensuring that unwanted data is never fetched. Furthermore, the only time that a write operation results in an external reference is when a cache miss occurs in a dirty block. In this case, only the valid words in the dirty bit will be written back to main memory. The main cost of this feature is an extensive 64 word by 16 bit status RAM. A write-back replacement scheme was also chosen to reduce the traffic on the external port. This requires a dirty bit for each block in the cache to signify if the block has been modified since it was last fetched. A standard cell block called STATREG (status register) was included to support this. The STATREG is loaded with a word out of the status RAM in the case of a dirty miss. The STATREG then encodes the word to get the block offset of the highest valid word. Once that word is read from the data RAM and sent to main memory, then STATREG is decremented such that encoded output will be the block offset of the next valid word. This continues until the STATREG detects that the current word is the last valid word in the block. Since a two-way set associative configuration is being used, then some method must be provided to decide which set entry will be replaced when a cache miss occurs. A least-recently-used replacement strategy was chosen to improve the hit-rate of the cache. The hardware implementation for this function requires a 32 bit register and control logic for reading and updating the register. The LRU was constructed from standard cells, and is merged with the cache controller block.

5.2 Data RAM

The main data RAM for the VIPER is duplicated twice to support the two-way set associativity. Each bank of RAM contains 512 words of 32 bits each, configured with 128 columns and 128 rows. The global layout of the RAM is the similar to that of the instruction tag memory. Many of the circuits in this RAM are identical to their single-port equivalents found in the instruction tag RAM, with some logic duplicated due to the extra port. The memory cell used in this RAM is an eight-transistor dual-port RAM cell. This circuit is an extension of the single-port cell, with two added read/write transistors and their associated bit lines. The four bit lines are necessary because the RAM must be able to perform two simultaneous differential writes to different locations in the same memory column. The size of this cell is about 50% larger than the corresponding single-port cell. Since the array of these cells will consume the majority of the area in a large RAM, then the same 50% factor roughly applies to the total size of the RAM as well.

The addition of the extra port creates several special cases that the cell must be able to handle. First, the cell must be robust enough that the stored value will not be disturbed when both ports perform a read from the same location at the same time. While it is an illegal operation for both ports to write to the same memory location in the same cycle, it is possible for the RAM to write to different column locations on the same row. This will result in one memory cell having three bit lines charged high and one bit line driven low. This situation is illustrated in Figure 10. If both ports do attempt to write to the same location at the same time, then the value stored at that location will be indeterminate. Fortunately, this case should not occur, since the PS compiler will not schedule such a conflict. SPICE simulations were run to verify that this cell could handle the above situations correctly.

5.3 Cache Controller and State Machine

The state machine for the data cache (DCSM) is much more complex than the corresponding state machine for the instruction cache. Most of the added complexity stems from having multiple memory ports to contend with. The state machine has fourteen separate states to control the different modes of cache operation. A state diagram of the of the DCSM appears in Figure 11. A brief description of each of the states follows:

- *NORM*: Normal mode. All load/store operations will begin by entering this state for a single cycle.
- *HOLD*: Hold state. The data cache will cycle in this state whenever it is not servicing a memory request. This includes waiting for the instruction cache to finish, or waiting on a pipeline stall.
- *DM2F*: Dirty-Miss-First state for FU2. First state in a dirty-miss service. One cycle is necessary here to read the first dirty word out of the data cache.
- *DM2D*: Dirty-Miss-Drive state for FU2. The controller enters this state for one cycle when the address and then the data for the dirty word are driven onto the external port.
- *DM2W*: Dirty-Miss-Wait state for FU2. The controller cycles in this state until it receives an external signal showing that the main memory has finished its write cycle. During this period, STATREG is updated and the next valid word in the dirty block is read out of the data cache.
- *DM3F, DM3D*, and *DM3W*: Same as *DM2F*, *DM2D*, and *DM2W* respectively except that the external port is servicing the memory request from FU3 instead of FU2.
- *M2F*: Miss-First state for FU2. The controller enters this state one cycle during a cache miss when it sends an address and a read request to main memory.
- *M2W*: Miss-Wait state for FU2. The controller cycles in this state waiting for data to return from main memory.
- *M2D*: Miss-Done state for FU2. The controller enters this state for one cycle to write the fetched word into the data cache.
- *M3F, M3W*, and *M3D*: Same as *M2F*, *M2W*, and *M2D* respectively except that the external port is servicing the memory request from FU3 instead of FU2.

6 Global Routing and External Interface

The external interface of the VIPER is dominated by the 128 input pins for the instruction bus, which are necessary to fetch one instruction per cycle. Another 28 output pins are dedicated to the program counter output bus, which connects externally to the instruction cache memory. Thirty-two bidirectional pins are used by the data cache to interface with the main memory, using these pins as a multiplexed data and address bus. In addition, twelve more pins are dedicated to various control signals, for a total of 200 signal pins. 32 pins were allocated for powering the pad frame and an additional 20 pins were allocated for powering the core blocks. Thus the total number of pins required is 252, which can be packaged in a standard 256-pin PGA. The pins were assigned to die edges in order to reflect the approximate aspect ratio of the placed but unrouted core macrocells.

Once the pin order was assigned, the task of routing all the macrocells together and to the pad frame was accomplished by the Octtools system. After several iterations of the final routing stage were performed, altering the pad ordering and macrocell placement to try to improve the routing quality each time, the layout was within 3 microns of the minimum size horizontally, and exactly minimum vertically. The routing is compact enough that the pad frame can not be made any smaller, thus the chip is perfectly balanced between being core-limited and pad-limited. With 450,000 transistors, the final die size is 12.9 by 9.1 mm, for a total area of 116.5 mm^2. Figure 12 shows the physical design of the VIPER processor die.

7 Simulation and Verification

The basic VIPER architecture was developed by extensive simulations in the C language at the register-transfer level, in order to decide which choices of configuration parameters provided the best overall performance [5]. A structural model was then developed in the VHDL hardware description language in order to understand the large-scale structures that would comprise the physical design and to determine the nature of the control signals needed by each major block. Many of the synthesis descriptions of the control paths were based on the VHDL model. However, this preliminary model did not include a four-phase clocking scheme, thus a second RTL simulator was written in C in order to simulate the final architecture on a phase-by-phase basis. This simulator was used to create test vectors for use by Stanford's switch-level simulator IRSIM. IRSIM is capable of simulating circuit element delay based on the capacitive loading of a node and the strength of the driving transistors, and was used for simulating large blocks as well as the entire processor core. When more accurate timing or electrical information was needed, SPICE simulations were performed on small subcircuits such as the register cells and some of the more critical data path elements, as mentioned previously.

Verification of the design proceeded by writing small test programs which were specifically designed to test various operational conditions, such as data hazard handling and branch condition bypassing, as well as general testing of the data path elements and the full range of the instruction set. The C simulator acted as an assembler for these test programs, creating the binary instruction traces that were used as stimulus for the switch-level simulations, as well as the expected values for all important registers and busses. The size of the data cache RAM prevented integrated testing of the processor and cache blocks, so processor simulation assumed a perfect memory system with no cache misses. To verify proper operation of the stalling mechanism, a separate test was performed where cache-miss cycles were manually inserted at various points in the simulation.

8 Final Thoughts

The VIPER VLIW microprocessor we have presented is capable of executing four 32-bit integer operations concurrently. The short design time of twelve months demonstrates the power of designing hardware with relatively simple, repeatable components. The small, simple control paths are a distinct advantage over the significantly more complex control found in superscalar architectures due to run-time scheduling hardware. With an operating speed of 25 MHz, the processor can achieve a peak throughput of 100 MIPS. We hope that the VIPER will prove to be a useful example of a general-purpose architecture with strong promise for use in other applications that demand high-performance computation.

References

[1] A. S. Aiken, "Compaction-Based Parallelization," Ph.D. Dissertation, Cornell University, August 1988.

[2] K. Ebcioglu, "Some Design Ideas for a VLIW Architecture for Sequential-Natured Software," *Proceedings IFIP*, 1988.

[3] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. P., Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Transactions on Computers*, Vol. 37, 1988, pp. 967-979.

[4] J. Labrousse and G. A. Slavenburg, "CREATE-LIFE: A Modular Design Approach for High Performance ASIC's," *COMPCON* 1990.

[5] A. Abnous, "Architectural Design and Analysis of a VLIW Processor," M.S. Thesis, University of California, Irvine, 1991.

[6] P. Chow and M. Horowitz, Architectural Tradeoffs in the Design of MIPS-X, *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 300-308, Pittsburgh, Pennsylvania, June 1987.

[7] A. Abnous, C. Christensen, J. Gray, J. Lenell, A. Naylor, and N. Bagherzadeh, "VLSI Design of the TinyRISC Microprocessor," *Proceedings of the 1992 Custom Integrated Circuits Conference*, pp. 30.4.1-30.4.5, Boston, May 1992.

[8] A. Casotto, editor. "Octtools 5.1 User's Guide," University of California, Berkeley, 1991.

[9] J. Hennessey and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA: 1990.

[10] A. Naylor, "Cache and Memory System Design for a VLIW Processor," M.S. Thesis, University of California, Irvine, 1992.

[11] J. Lotz, B. Miller, E. Delano, J. Lamb, M. Forsyth, and T. Hotchkiss, "A CMOS RISC CPU Designed for High Performance on Large Applications," *IEEE Journal of Solid-State Circuits*, vol. SC-25, pp. 1190-1198, October 1990.

[12] M.D. Hill, "The Case For Direct-Mapped Caches," *IEEE Computer*, vol. 21, no. 12, pp. 25-41, December 1988.

A Captions

Captions for Gray, Naylor, Abnous, Bagherzadeh; **VIPER–A-25 MHz, 100-MIPS Peak VLIW Processor**:

Table 1. Phase-by-phase operation of the VIPER.

Table 2. Die area consumed by various processor elements.

Fig. 1. Block diagram of the processor.

Fig. 2. Pipeline structure of the VIPER.

Fig. 3. Organization of the eight-port register file.

Fig. 4. Transistor-level design of the eight-port register cell.

Fig. 5. Organization of the data path (top) and corresponding floor plan.

Fig. 6. Design of the operand unit and bypassing switches.

Fig. 7. Block diagram of the program counter unit.

Fig. 8. Simulation of the processor's critical path.

Fig. 9. State machine diagram for instruction cache.

Fig. 10. Write conflicts in the dual-port memory cell.

Fig. 11. State machine diagram for data cache.

Fig. 12. Physical design of the VIPER processor.

Clock phase	Pipeline stage	Operation
Phase 1	IF	Program counter driven to instruction cache
	ID	Instruction decoding begins Bypass condition detection begins
	EX	Evaluation of ALU elements begins Data cache RAM decoders evaluate Data tag and status RAMs read and compared
	WB	Destination busses written into register file
Phase 2	IF	Instruction cache tags read and compared
	ID	Source register specifiers are decoded Register file precharges
	EX	Evaluation of ALU elements in progress Data RAM is read
Phase 3	IF	Stall signal sent to clock generator in event of instruction cache miss
	ID	Source operands read from register file Outputs of instruction decoders latched Computation of possible PC targets completes
	EX	ALU outputs latched Stall signal sent to clock generator in event of data cache miss
Phase 4	IF	Instruction latched in instruction pipeline
	ID	Bypassing paths activated, source operands latched Source busses driven Data address driven to cache Next PC selected
	EX	Destination busses driven and latched Destination register specifiers decoded

Table 1: Phase-by-phase operation of the VIPER (Gray, Naylor, Abnous, Bagherzadeh)

Cell	**Percentage of total chip area**
Register file	5.1
Branch units	2.8
Load/store units	3.2
Program counter unit	0.9
Control	6.7
Local routing	9.4
Processing core total	28.1
Data cache RAM	25.7
Data cache control	7.1
Instruction cache control	4.4
Cache total	37.2
Clock generator	1.8
Pads	7.3
Global routing	25.6

Table 2: Die area consumed by various processor elements (Gray, Naylor, Abnous, Bagherzadeh)

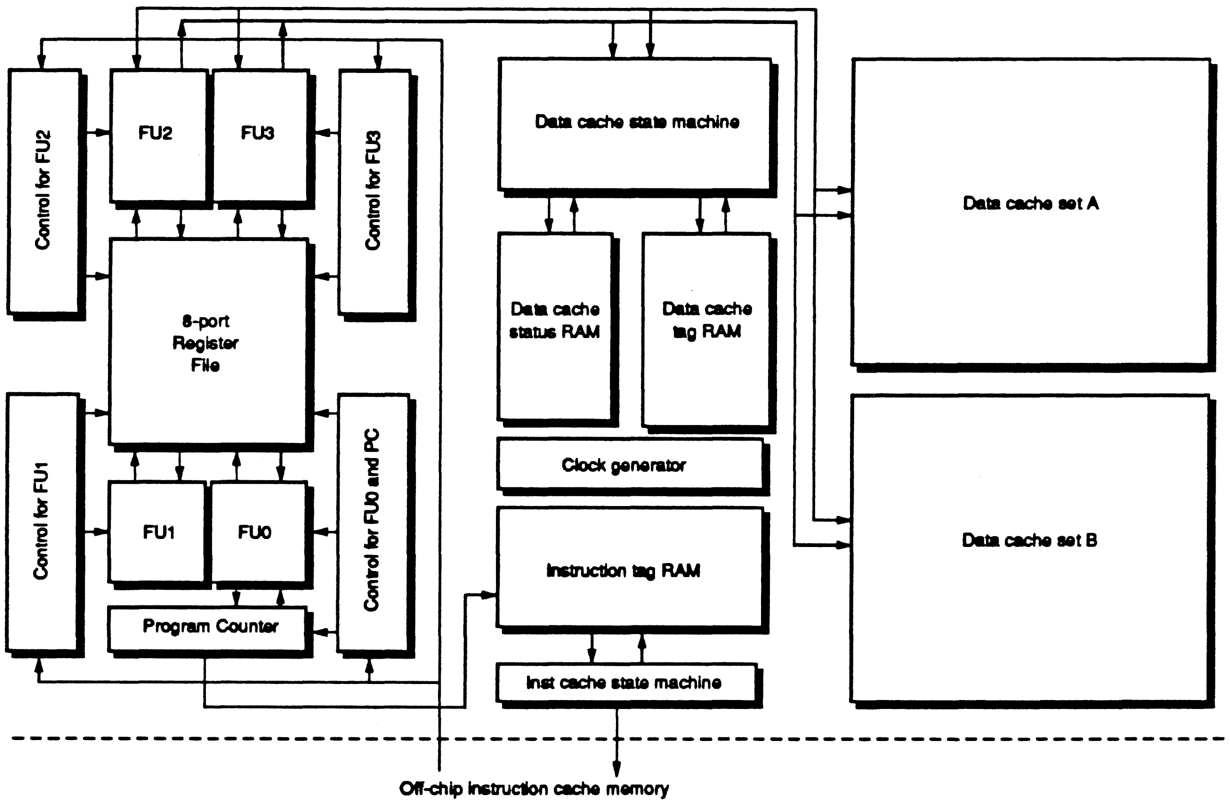


Figure 1: Block diagram of the processor (Gray, Naylor, Abnous, Bagherzadeh)

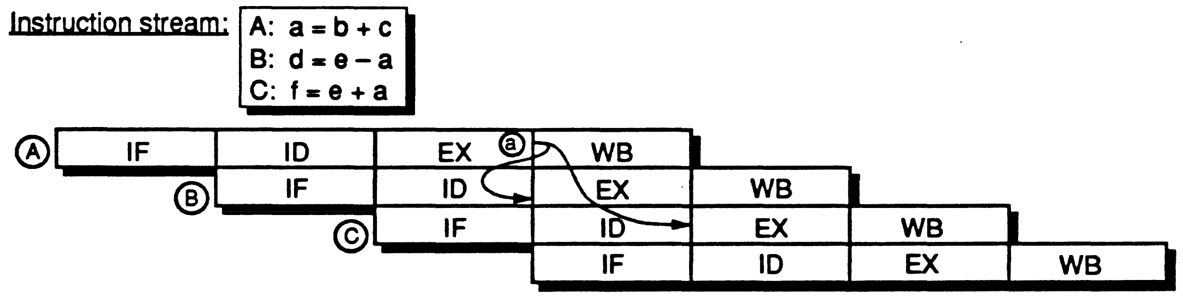


Figure 2: The pipeline structure of VIPER (Gray, Naylor, Abnous, Bagherzadeh)

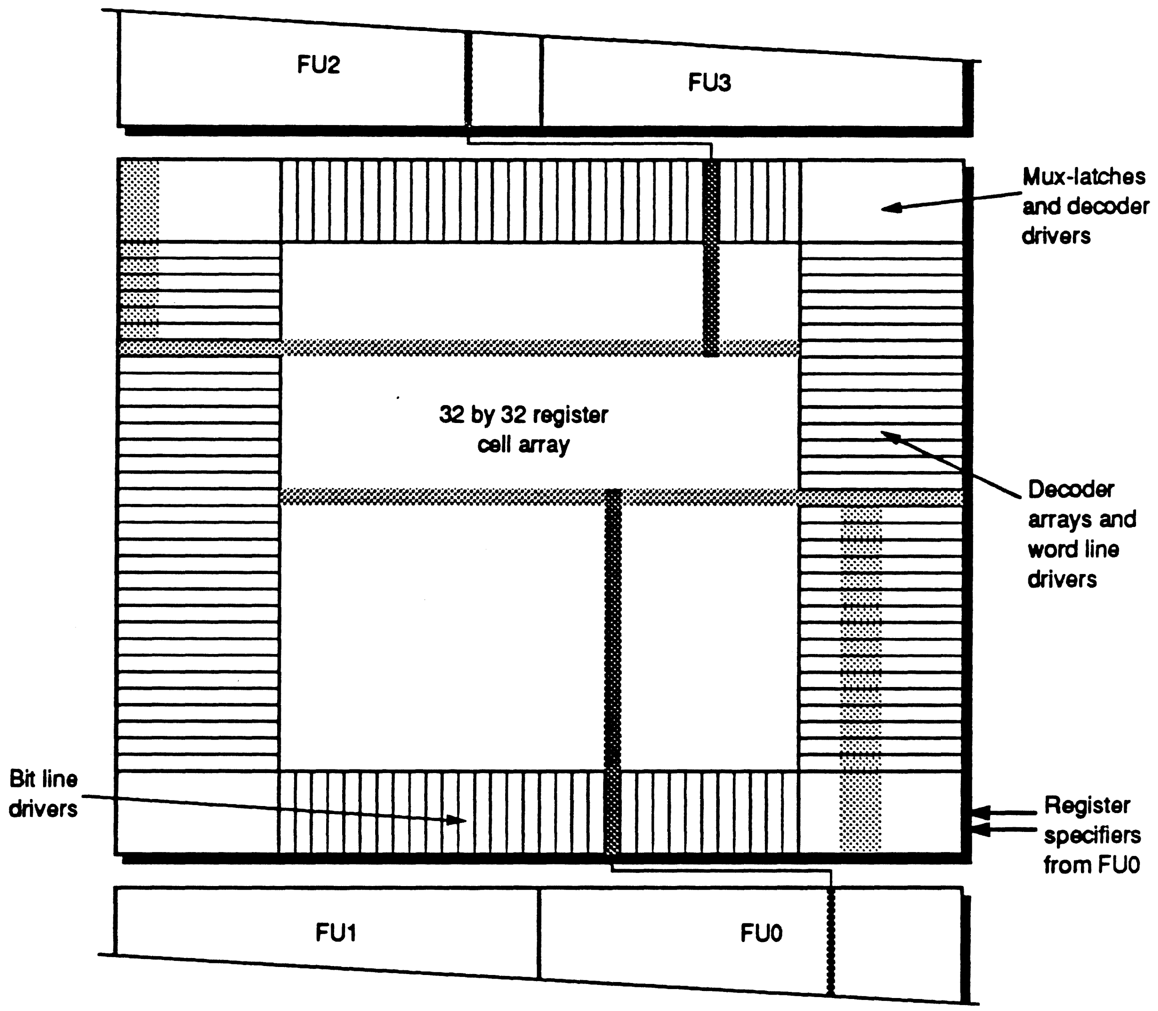

FU2
FU3
Mux-latches and decoder drivers
32 by 32 register cell array
Decoder arrays and word line drivers
Bit line drivers
Register specifiers from FU0
FU1
FU0

Figure 3: Organization of the eight-port register file (Gray, Naylor, Abnous, Bagherzadeh)

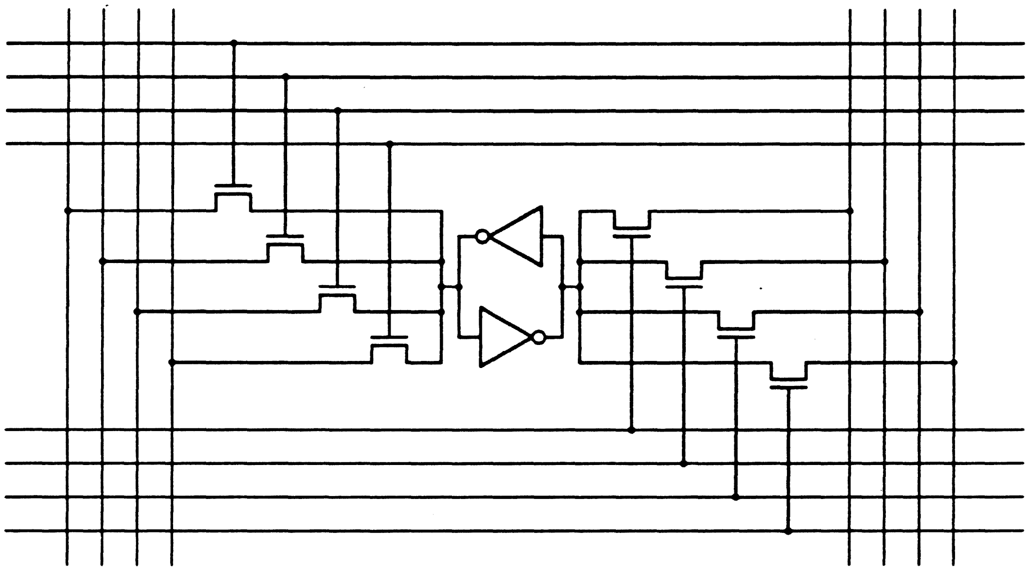
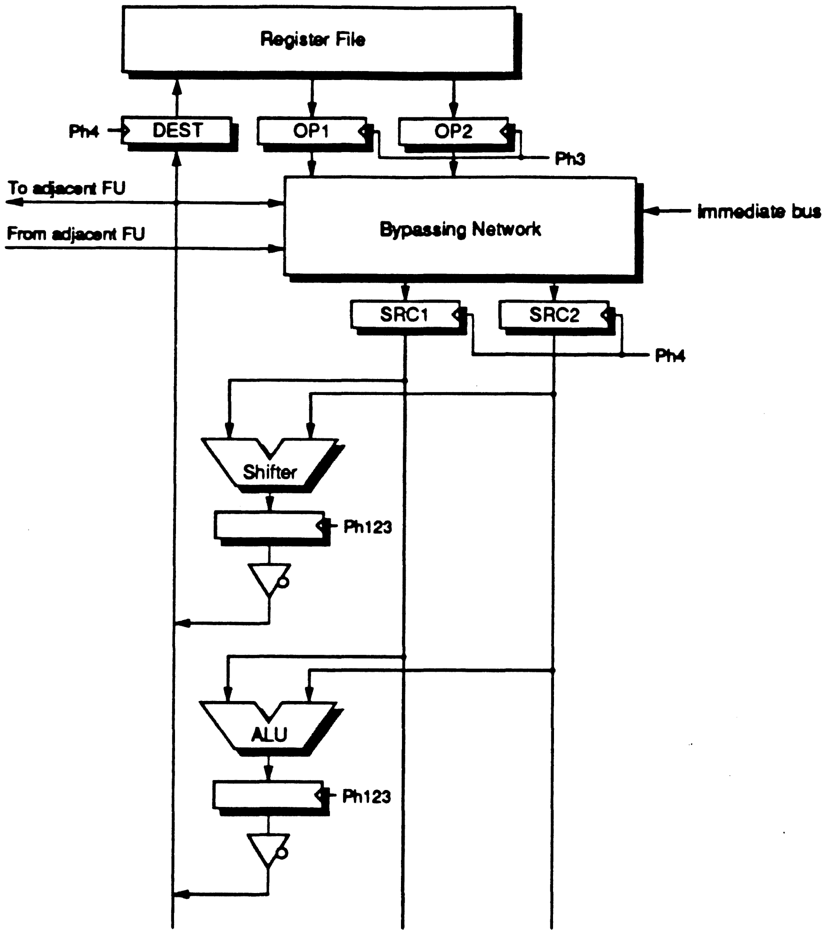


Figure 4: Transistor-level design of the eight-port register cell (Gray, Naylor, Abnous, Bagherzadeh)



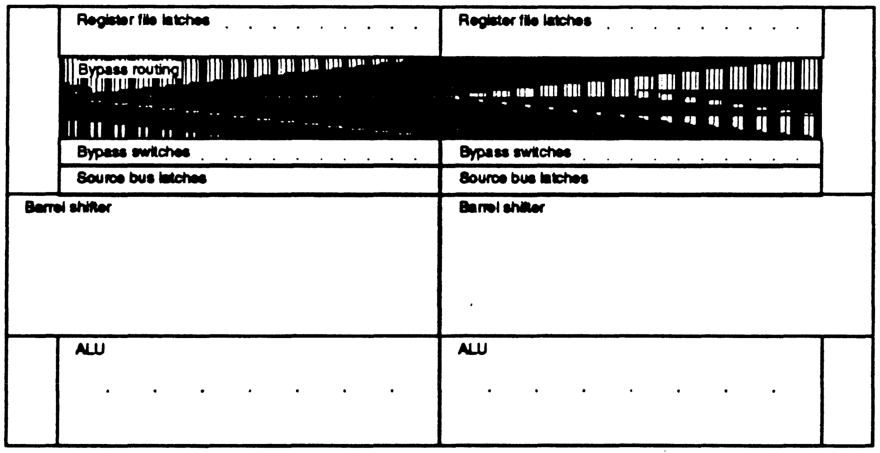


Figure 5: Organization of the data path (top) and corresponding floor plan (Gray, Naylor, Abnous, Bagherzadeh)

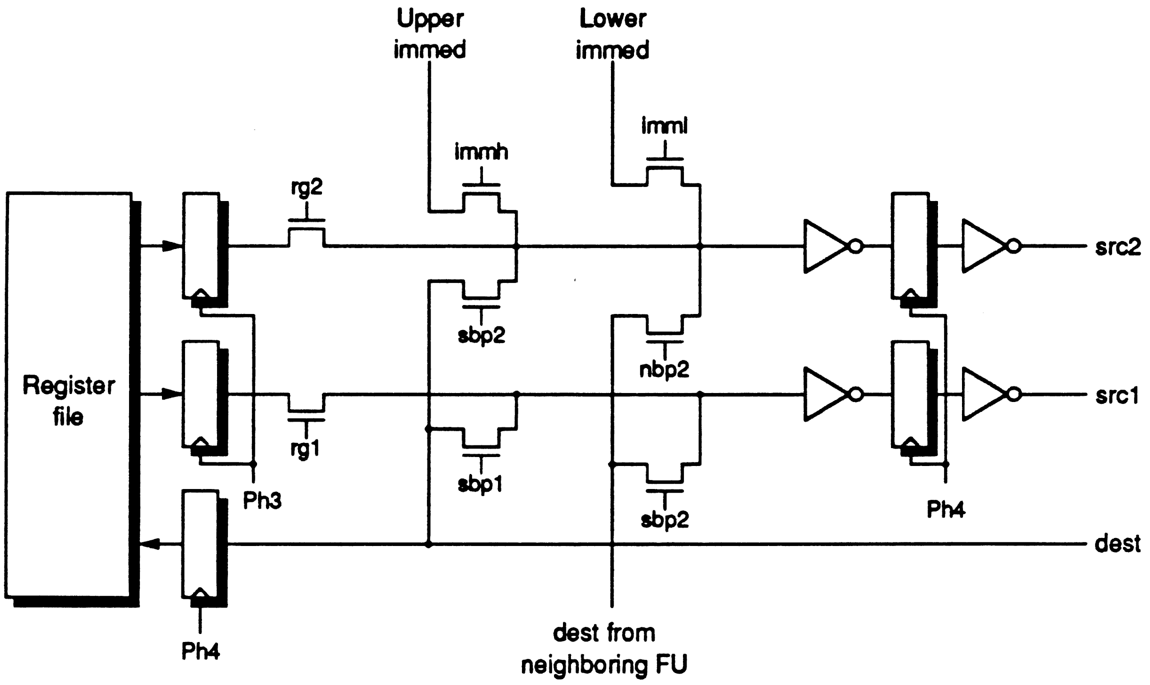


Figure 6: Design of the operand unit and bypassing switches (Gray, Naylor, Abnous, Bagherzadeh)

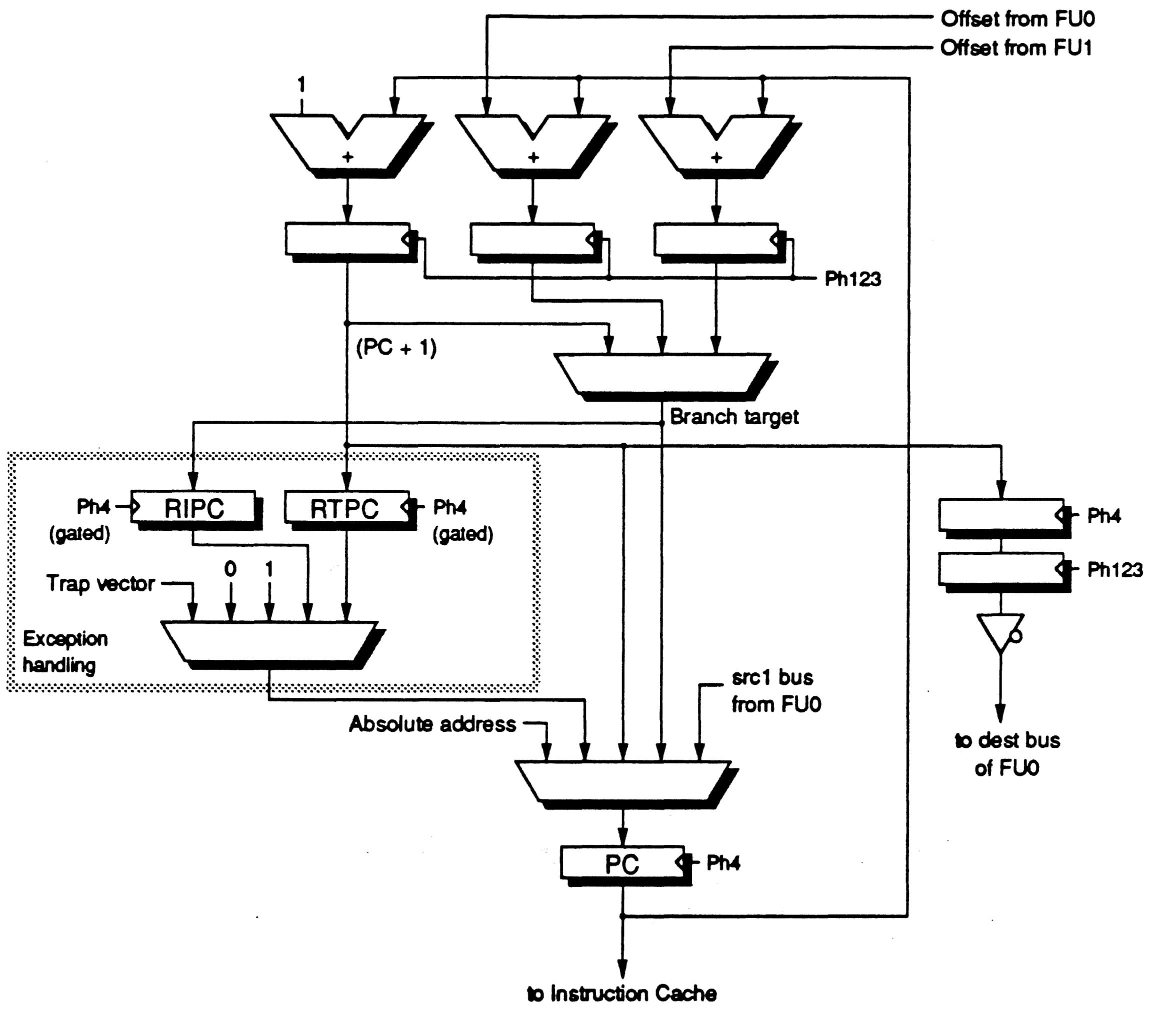


Figure 7: Block diagram of the program counter unit (Gray, Naylor, Abnous, Bagherzadeh)

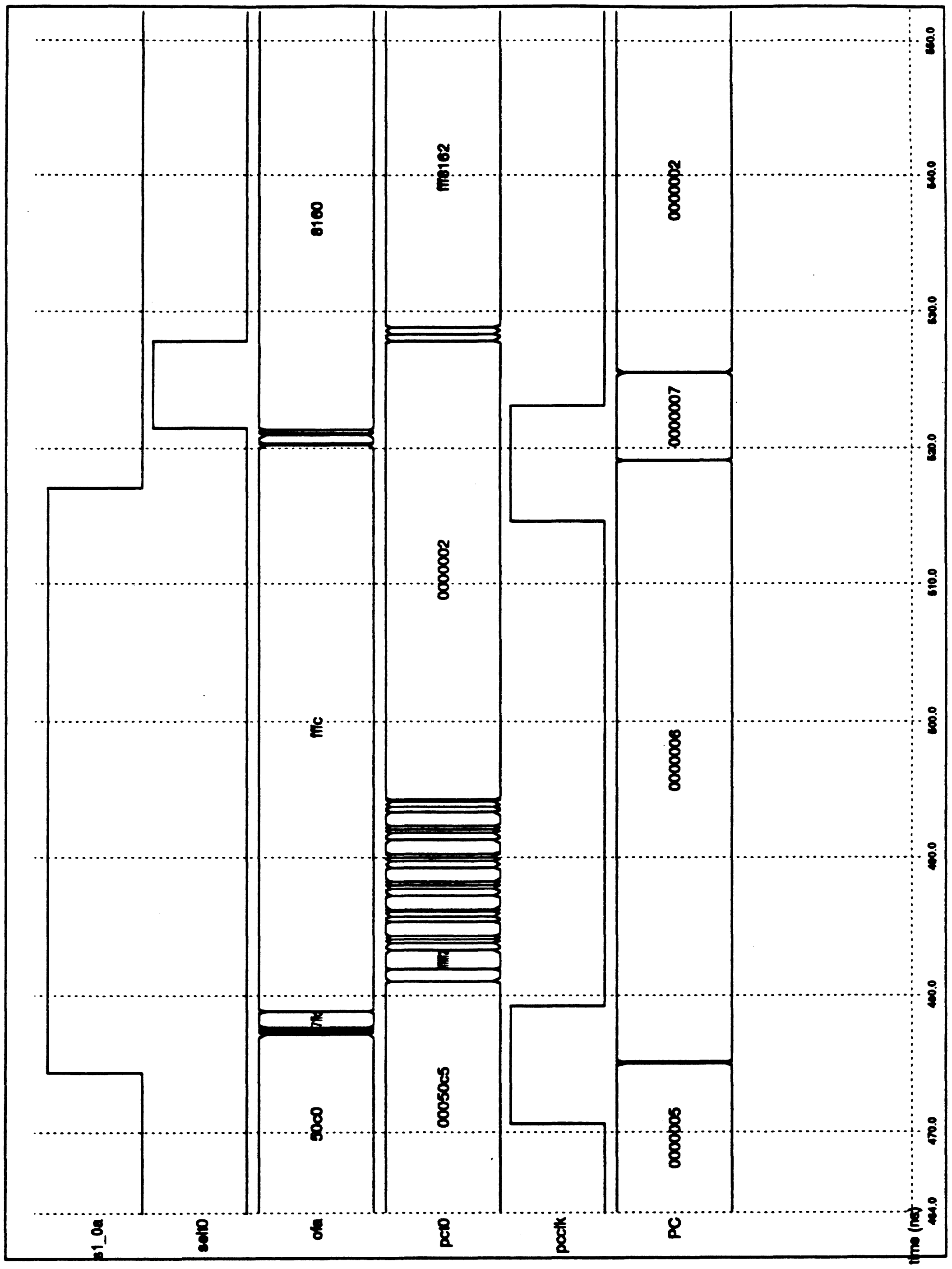


Figure 8: Simulation of the processor's critical path (Gray, Naylor, Abnous, Bagherzadeh)

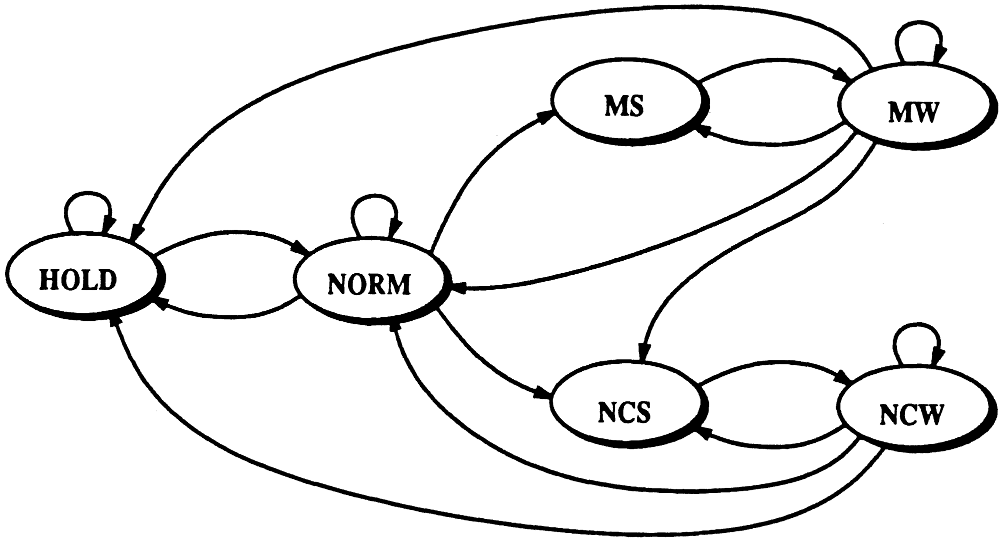


Figure 9: State machine diagram for instruction cache (Gray, Naylor, Abnous, Bagherzadeh)

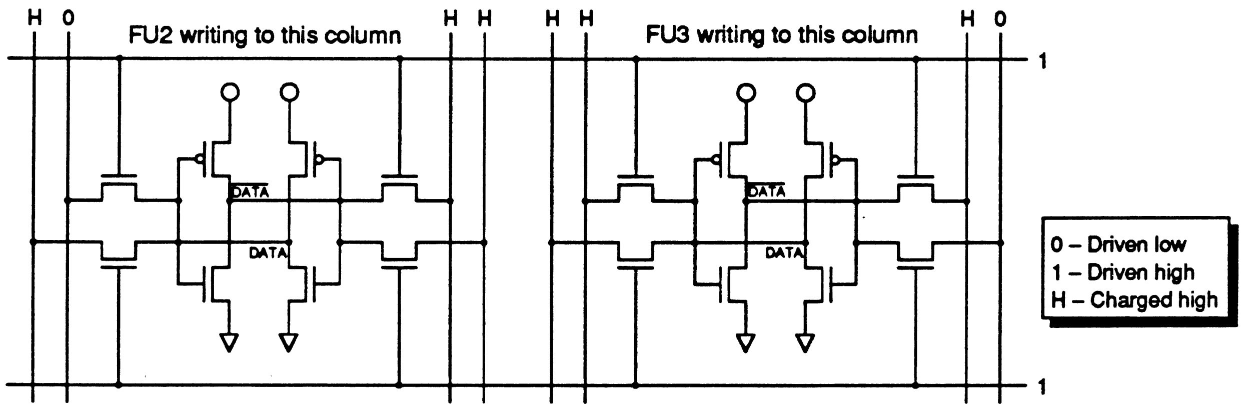


Figure 10: Write conflicts in the dual-port memory cell (Gray, Naylor, Abnous, Bagherzadeh)

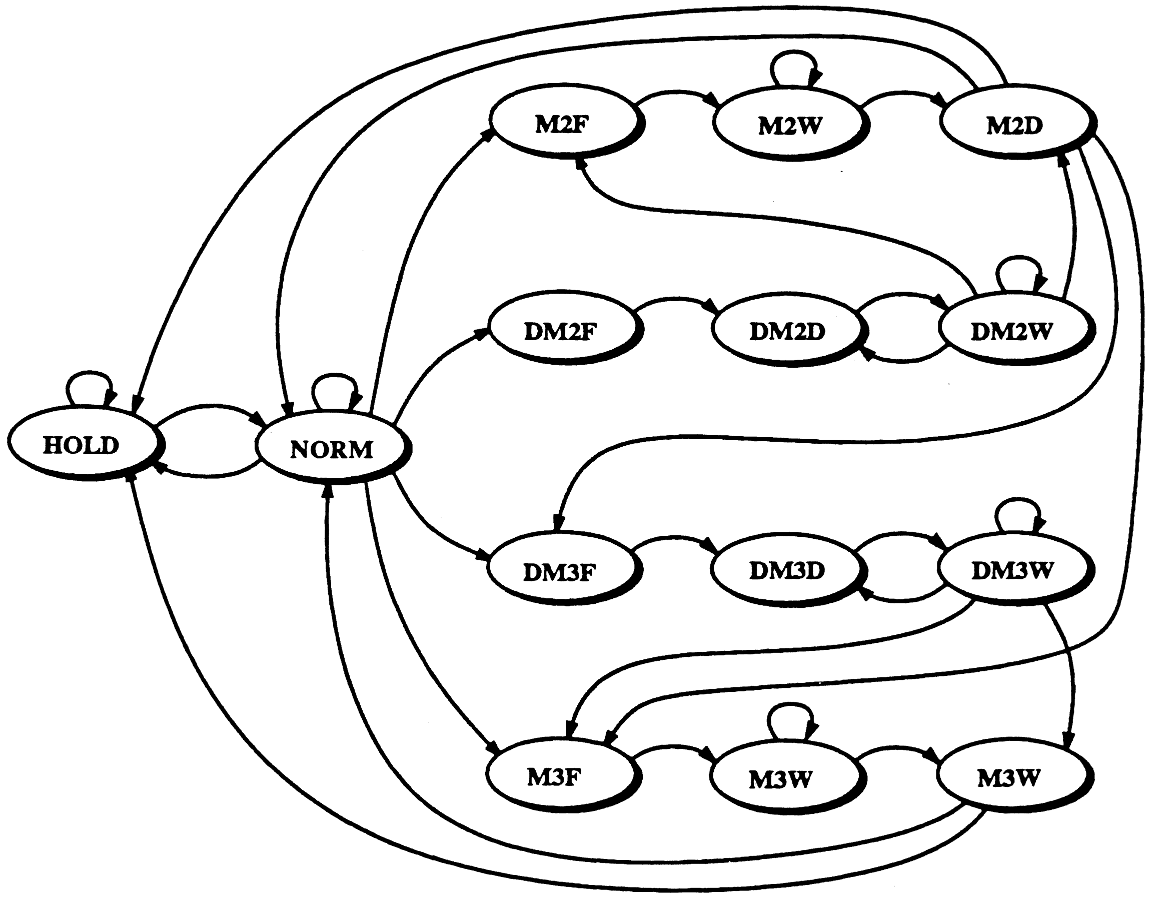


Figure 11: State machine diagram for data cache (Gray, Naylor, Abnous, Bagherzadeh)

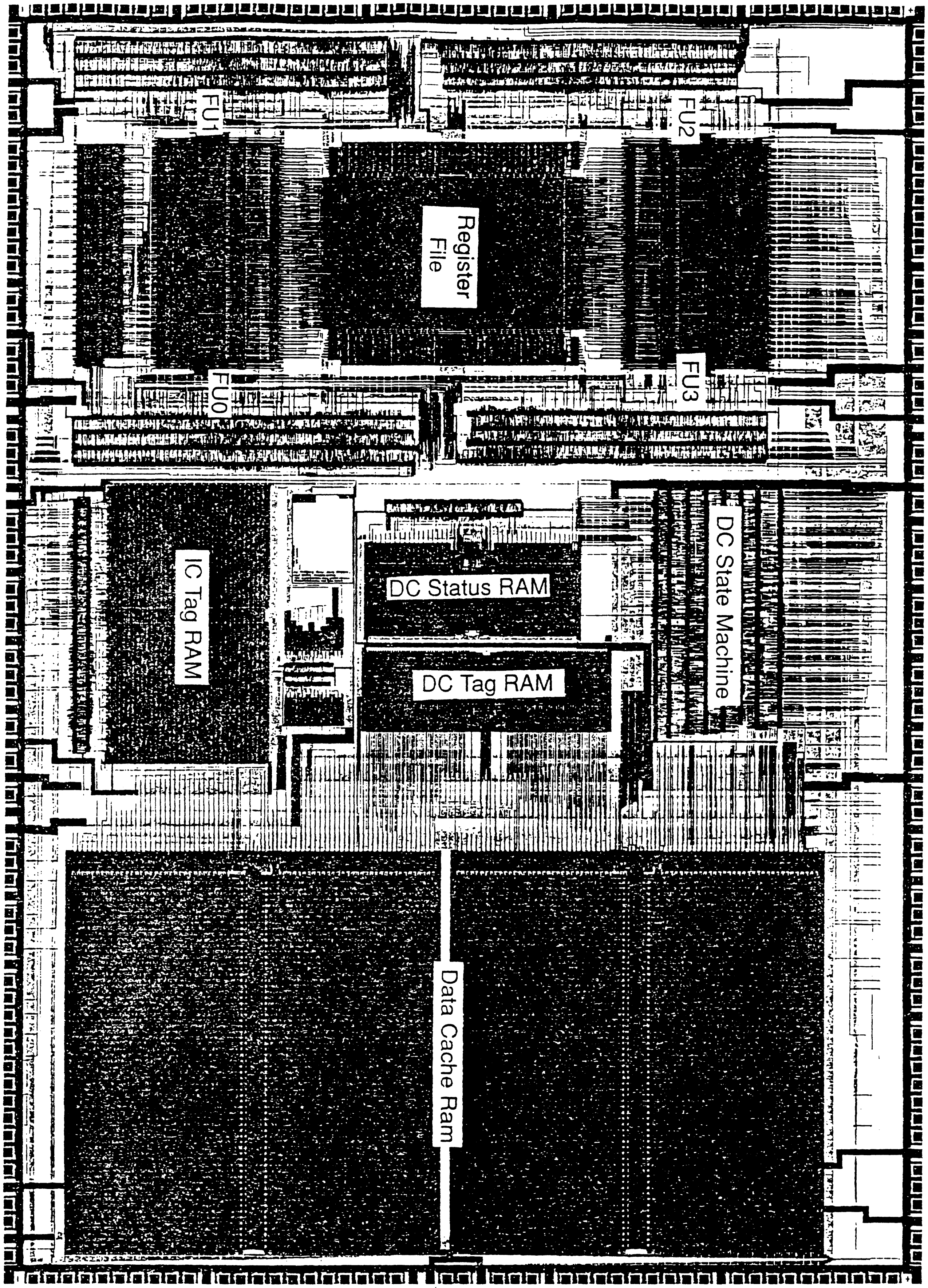


Figure 12: Physical design of the VIPER processor (Gray, Naylor, Abnous, Bagherzadeh)