The Design of a 7-stage Pipelined 143 MHz Microprocessor Implementing a Subset of the x86 ISA

Hyesoon Kim
Onur Mutlu
Santhosh Srinath

Introduction to Microarchitecture, Spring 2002
Submitted: May 10, 2002

Abstract

We have designed a processor that implements a subset of the x86 ISA. The processor is 7-stage pipelined, supports in-order execution with precise exceptions, contains 512-byte, direct-mapped instruction and data caches. It communicates with peripheral devices and main memory using a central system bus that runs at the operating frequency of the processor. The main goal of the design was to attain 200 MHz frequency. The final design runs at 143 MHz (7 ns cycle time).

Contents

1 Introduction **1**

2 Design Overview **1**

2.1 High-Level Connections in the System 1

2.2 Processing Core 2

2.2.1 Fetch Stage 2

2.2.2 Decode Stage 3

2.2.3 Register Read Stage 6

2.2.4 Address Generation Stage 7

2.2.5 D-cache Access Stage 9

2.2.6 Execution Stage 10

2.2.7 Writeback Stage 10

2.2.8 Interrupt/Exception Handling 11

2.2.9 Branch Handling 12

2.2.10 Specialized Hardware to Execute REP MOVS 12

2.3 Memory Subsystem 13

2.3.1 I-cache 13

2.3.2 D-cache 13

2.3.3 TLB 14

2.4 Bus Arbiter, Bus, Memory, and Peripherals 15

2.4.1 Bus Arbiter 15

2.4.2 System Bus 15

2.4.3 Main Memory 16

2.4.4 Keyboard Controller 16

2.4.5 Monitor Controller 16

3 Design Tradeoffs **17**

3.1 Design Goals 17

3.2 Tradeoffs in the Processing Core 17
3.2.1 Fetch Stage Tradeoffs 17
3.2.2 Decode Stage Tradeoffs 18
3.2.3 Register Read Stage Tradeoffs 18
3.2.4 Address Generation Stage Tradeoffs 19
3.2.5 D-cache Access Stage Tradeoffs 19
3.2.6 Execution Stage Tradeoffs 19
3.2.7 Writeback Stage Tradeoffs 19
3.2.8 Optimizations 20
3.3 Tradeoffs in the Memory Subsystem 20
3.4 Tradeoffs in the Bus Arbiter, Bus, Memory, and Peripherals 21

4 Critical Path Analysis **21**

5 Limitations **22**
5.1 Memory Subsystem Limitations 22
5.2 Processing Core Limitations 23
5.3 Limitations in Bus Arbiter, Bus, Memory, Peripherals 24

6 Conclusions **24**

7 Appendix **25**

List of Tables

1 Instruction Set 25

2 Micro-operations 29

3 Control signals as output by the decoder 30

4 Data signals as output by the decoder 31

5 Delay of each pipeline stage 31

List of Figures

1 System Overview 32
2 Processing Unit Overview 33
3 Fetch Stage Block Diagram 34
4 Decode Stage Block Diagram 35
5 Register Read Stage Block Diagram 36
6 Address Generation Stage Block Diagram 37
7 D-cache Access Stage Block Diagram 38
8 Execution Stage Block Diagram 39
9 Writeback Stage Block Diagram 40
10 Special Hardware and State Diagram for REP MOVS 41
11 I-cache Controller State Diagram 42
12 D-cache Controller State Diagram 43
13 Bus Arbitration Logic 44
14 Memory Controller State Diagram 44
15 Keyboard Controller State Diagram 45
16 Monitor Controller State Diagram 45

1 Introduction

This report describes our design and implementation of a computer system implementing a subset of the x86 instruction set architecture (ISA). Our system includes a 7-stage in-order pipeline which overlaps the execution of instructions, a memory system consisting of 512-byte, direct-mapped, write-back instruction and data caches and a 32 KB main memory, two I/O devices (a keyboard and a monitor) and a system bus that connects the processing core to the memory, keyboard, and the monitor. The system runs at a 7-ns cycle time, which is higher than our initial target which was 5 ns. The system is designed as a general-purpose system to execute integer programs.

Our primary design goals were a short cycle time and a simple, streamlined hardware implementation that would work correctly for all cases. We have achieved our second goal: our design works without a glitch. However, we fell a little short of achieving the short cycle time we aimed for. The details of how we could improve our design to fulfill our first goal are provided in this report.

Section 2 of this report describes our design (control and the datapath) in detail. Section 3 describes the design considerations, the tradeoffs we considered in building our system and our design choices along with explanations of why we favored one design choice over another. Section 4 analyzes the critical path of our pipeline and comments on how it can be improved.Section 5 describes what we feel are the major limitations and bottlenecks of our design and discusses how our design can be extended to obtain higher performance. Section 6 contains our concluding remarks.

2 Design Overview

Figure 1 shows a high-level overview of the designed system. There are three main components of our system: the processing core, the memory subsystem, and the bus arbiter/memory/peripherals. In this section, we put these three components in perspective and elaborate on the design of each component.

2.1 High-Level Connections in the System

In an implementation of our system, we plan to have the processing core and the memory subsystem placed on-chip. The chip will be connected to the main memory and peripherals using the system bus. Bus arbitration logic will also be placed on chip to reduce the main memory access latency experienced by the processing core.

The processing core communicates with the memory subsystem using dedicated address and data buses for instruction and data accesses. Memory subsystem is the on-chip portion

of the memory hierarchy (and controls associated with it). It supplies instructions to be executed by the processing core. It also supplies the data on which those instructions will operate.

Memory subsystem is connected to the off-chip main memory (the main supply of instructions and data) and the peripheral I/O devices via the system bus. Memory subsystem is also internally connected to the bus arbiter. The transfer of data on the bus is mediated centrally by the bus arbiter, whose job is to decide which device controls the bus at any given time.

The last connection between the high-level components of the system is the one between the processing core and the I/O devices. I/O devices have dedicated interrupts lines coming into the processing core which can interrupt the execution of instructions in the pipeline. In response to an interrupt, however, the processing core needs to go through the memory subsystem to communicate with the I/O devices.

2.2 Processing Core

The processing core is the part of the system where instructions get executed. We have pipelined the processing core into 7 stages (Fetch, Decode, Register Read, Address Generation, D-cache access, Execution, and Writeback) to increase the throughput of instruction execution. The high level overview of these seven stages is shown in Figure 2. We discuss the purpose and design of each stage in the next few sections.

2.2.1 Fetch Stage

The main purpose of this stage is to bring instructions into the processing core. To accomplish this, the fetch stage needs to access I-cache and figure out which address to fetch from in the next cycle. Due to the variable-length nature of instructions in the x86 ISA, we do not exactly know how much data we should bring into the processing core from the I-cache. Worse, we do not know where the next instruction starts until the current instruction is decoded. To overcome these problems in an efficient and effective way, the fetch stage buffers the bytes fetched from the I-cache in the instruction register (which is a 256-bit wide latch between fetch stage and decode stage). Moreover, fetch stage always fetches one cache line ahead of the current cache line that is being processed by the decoder. By such a design, we aim to overlap the instruction fetch latency with instruction processing latency.

A detailed schematic of the fetch stage is given in Figure 3. I-cache provides 128 bits in each cache line. This size is chosen, because the maximum size of an instruction cannot be more than 128 bits. The most interesting part of the fetch stage is the logic that handles the shifting of the instruction register (IR) and the logic that determines when

to latch new data from I-cache into the instruction register. Our design was based on our initial finding that the number of bytes consumed by the decoder is known late in the cycle (assuming a 5 ns cycle time). To prevent this information from stretching the cycle time, we decided to use this information in the fetch stage as late as possible.

When the IR is more than half full, fetch stage does not latch the instructions from I-cache into the IR, even though instructions may be available. Whether or not IR is more than half full is determined based on stale information. In other words, this decision is not based on how many bytes are consumed by the decoder in the current cycle. If the IR is less than half full and I-cache access results in a hit, the contents of IR are concatenated with the newly-fetched 128 bits from the I-cache. The concatenated instruction bits are eventually shifted again based on the information given by the decoder as to how many bytes it consumed during the current cycle. This design makes use of the slow-coming information from decode as late as possible.

To make such a design possible, we maintain a 5-bit unsigned register that denotes how many bytes in the instruction register are valid. Ideally, we would like the value of this register to be always greater than or equal to 16. The update of this register is performed using a 5-bit adder that subtracts the number of bytes consumed by the decoder from the (old value of this register + number of useful bytes from I-cache). This update is one of the critical loops in our design which we will touch upon later, but it is not on the critical path.

We maintain three different latches that contain addresses. EIP contains the correct starting address of the instruction that is currently being decoded. EIP+CS contains the sum of the EIP value and the CS value. The purpose of this register is to remove the EIP+CS addition in I-cache access out of the critical path. EIP+CS value needs to be calculated on every operation that changes either of these architectural registers. The third latch, VIP+CS, is the actual latch used to access I-cache. This latch is 28 bits and addresses only the cache lines. It runs ahead by 1 cache line most of the time to keep supplying instructions at a rate greater than the consumption rate of the rest of the pipeline.

2.2.2 Decode Stage

Our design includes a single-cycle decode stage that is designed to decode instructions in less than 5 ns. The decode stage conatins the most complicated logic in the whole processing core. Due to the variable length and non-uniform-decode property of x86 instructions, a lot of hardware is needed to make this stage faster. Decode stage performs other functions than only decoding instructions and generating control signals. The following is a comprehensive list of the functions performed by the decode stage:

1. Decode of one instruction per cycle and generation of control signals for that instruc-

tion.

2. Generation of the "number of instructions decoded" signal and communication of this information to the fetch stage.

3. Generation of micro-operations for complex instructions whose execution requirements do not comply with the pipeline organization.

4. Detection of REP MOVS instruction and control of its execution.

5. Injection of micro-operations that save the state of the machine and jump to the correct interrupt service routine on detection of an exception or interrupt.

6. Assignment of an 8-bit tag for each decoded instruction.

The main function of the decode stage is the decode and generation of a single instruction in a single cycle. To achieve this purpose we dedicate a lot of hardware that decodes different portions of the instruction register in parallel. Eventually the correct control signals are selected based on the result of the decode of opcode and modR/M bytes.

The prefix decoder decodes first 3 bytes in the instruction register and determines whether each of them is a prefix or not. The opcode decoder looks at the first 4 bytes of the IR and tries to find an opcode. The correct opcode is selected based on the result of the prefix decoder (but the actual decoding of the opcode is done in parallel with the decoding of the prefix). For simple instructions that do not require more than one micro-operations, the opcode byte (or the second byte of the opcode if it is a 2-byte opcode) is used to access the control store ROM which provides some preliminary control signals and information required for the rest of the decode stage. If the instruction requires a ModR/M byte, the work done by the ModR/M and possibly SIB decoders would be useful for later stages. Register numbers are also selected based on whether or not the instruction requires ModR/M and/or SIB bytes. There is also another rom that is used to generate the control signals for selecting the inputs of the adders used in address generation stage. This ROM is accessed using the ModR/M and SIB bytes.

Instruction Size Calculation

The calculation of the instruction size is one of the important loops in the processing core. The size of the decoded instruction is determined by the decoder and then conveyed to the fetch stage so that fetch stage correctly shifts the IR. Last piece of information that makes the instruction size stable comes from the ModR/M decoder. We kept this loop under 5 ns by optimizing the design and the logic. The shifted value of IR is latched before 5 ns, which could have been our critical path.

Control Signals

The list of the control signals generated by the decoder is provided in Table 3 in Appendix. These signals determine how the data flows in the pipeline. Table 4 includes the data signals that are output by the decoder.

Micro-operations

The execution requirements of some the implemented instructions do not comply with the organization of our pipeline and allocation of resources. For example, the IRETD instruction performs three pop operations, where each pop operation reads from the stack (which is in memory) and increments the stack pointer. The first two of these pop operations are in effect branches in that they modify the EIP and CS registers. Our pipeline organization does not allow so many cache accesses and serial updates to the state at the same time. To avoid stalling at many different parts of the pipeline for the execution of the IRETD instruction, we decided to break this instruction into three different and distinct micro-operations: POP EIP, POP CS, POP EFLAGS. All three of these are exactly the same as a normal POP instruction with destination being the EIP, CS, or EFLAGS.

Other instructions that were broken into micro-operations are far CALL (opcode 9A), far RET (opcode CB), far RET imm16 (opcode CA), and REP MOVS (opcodes F3A4 and F3A5).

For interrupt/exception handling, the decoder also generates micro-ops to push EFLAGS, CS, and EIP, and load the target CS and EIP from the interrupt descriptor table.

A list of all micro-operations are given in Table 2 in the Appendix.

To inject micro-operations into the pipeline, decoder generates a stall signal that stalls the fetch stage and inhibits the update of the instruction register. While that stall signal is high, decoder sequences in the control store ROM every cycle. Once the last micro-operation is inserted into the pipeline, the decoder removes the stall signal and starts accessing the control store ROM based on the contents of the IR (opcode). The sequencer used to switch back and forth from and to the micro-operation insertion mode is shown as the muxes that are above the control store ROM in Figure 4.

Instruction Tagging

As limited support for out-of-order execution, the decoder tags each instruction with an 8-bit sequence number. At any given time, different instructions in the pipeline will have unique tags. These tags are used in forwarding and updating of the scoreboard bits. Main function of tags in our design is to let the pipeline not stall in case of write-after-write hazards.

Decoder Stall Signals

The decode stage generates two stall signals. One is the stall signal generated due to micro-operation insertion. The other is the stall signal due to not enough bytes being available in the instruction register. Both stall signals are propagated to the fetch stage, but are treated differently. In case of the stall due to micro-operation insertion, the IR is write-disabled, but in case of not enough bytes being available IR needs to be enabled so that new bytes from I-cache can be latched into IR. However, if the decoder does not

have enough instructions, the IR should not be shifted.

Other Decoder Functions

We will touch upon the control for interrupt/exception handling and REP MOVS in later sections.

2.2.3 Register Read Stage

The block diagram of the Register Read Stage is shown in Figure 5. This stage takes as input the signals generated by the decoder and performs the following functions:

1. Reads the general purpose and segment register files and necessary flags.

2. Generates a stall signal if any of the required registers/flags is not available in the register files or is not forwarded.

3. Calculates the target EIP and target EIP+CS for relative and far branches.

4. Determines whether conditional branches are taken.

5. Generates the inputs of address generation adders that will be used in the next stage.

Scoreboarding

Main function of this stage is the fetch of register operands. In a pipelined microarchitecture, this fetch needs to be supported with hardware interlocks so that stale values are not supplied to incoming instructions. We implement this hardware interlock using a scoreboard. When an instruction is known to write into a register, it sets the valid bit of that register to 0. The instruction also writes its tag in the tag store associated with the register. If an instruction later tries to read that register, it will need to stall since the valid bit associated with that register is cleared. When the first instruction eventually reaches the writeback stage, it writes into the destination register. The valid bit is set back to 1 only if the tag of the instruction in the writeback stage matches the tag of the register. This way we can have multiple instructions in the pipeline writing into the same register,

Forwarding

Register values can be forwarded from either the execution stage or the writeback stage. Forwarding is only done for the register designated as the destination register for the instruction. A value is fowarded if the required register id and tag matches with the destination register id and tag of the instruction in the execution or writeback stages.

Register File Organization

The general purpose register file contains 8 32-bit architectural general-purpose registers. It has 4 read ports and 3 write ports. The maximum number of read ports required by an

instruction is 3. 4th port is added for convenience in our design. It was easier to allocate ports to instructions using a 4-ported register file. The maximum number of write ports required by an instruction is 3 (REP MOVS).

Reads and writes to the GPR file are done on a 32-bit granularity. Hence, if an instruction only requires an 8-bit register, the whole 32-bit value is read, but the control signals are used to manipulate only the required 8 bits in the register. When an instruction needs to write only a portion of the 32-bit register, the control signals generated by the decoder for that instruction align the value such that the correct portion of the register is written. This means that an instruction needs to carry the whole 32-bit value although it may not operate on all 32 bits. The validation and invalidation of registers are also done on a 32-bit granularity which likely introduces unnecessary stalls in the pipeline, but simplifies the design.

Segment register file is also an 8-entry, 4-read-port, 3-write-port structure where each register is 32 bits. Only 6 of the entries are used, because there are only 6 architectural segment registers in the x86 architecture. Every instruction is required to read the CS value because there might be an exception/interrupt which may cause that instruction to save the context. If the CS value is not valid, then the register read stage generates a stall. Only stack operations, instructions that require modR/M memory mode, and REP MOVS need to source the segment register file. The segment scoreboard is organized the same way as the GPR scoreboard.

Flags are not stored in a register file. Instead each flag is treated as a separate entity with exclusive read or write signals. Whether or not an instruction needs to source a flag is determined based on the operation id produced by the decoder.

2.2.4 Address Generation Stage

The block diagram of this stage is shown in Figure 6. The functions of the address generation stage are as follows:

1. Calculation of the read/write address required for instructions sourcing and writing into memory.

2. Calculation of the highest and lowest possible addresses to be accessed by the instruction.

3. Calculation of the offset needed for segment limit checks.

4. Generation of addresses needed by the REP MOVS instruction.

5. Generation of IDTR address for exception handling.

6. Segment limit check for the offset generated by a relative branch (JMP/Jcc) in the register file stage and redirection of the fetch stage based on the result of this limit check.

ModR/M/SIB Address Generation

To generate the ModR/M address required by an instruction, we use a 4-input adder that is built using three 2-input adders. Our adder is optimized for latency (at the expense of area) and a 2-input adder operates at 2 ns. The 4-input address generation adder generates results at 3.8 ns. After the generation of the ModR/M address, we use muxes to select whether it will be used as a destination or a source.

The inputs of the 4-input adder are determined in the register read stage based on the control signals generated by the decoder. The following lists what those four inputs can possibly be:

1. Segment register value
2. Base register value
3. Displacement value (or 0)
4. Scaled index register value (or 0)

This 4-input adder thus generates a modR/M or SIB address based on the inputs propagated to it from the register read stage. Other than the ModR/M address that will be sent to the memory subsystem, this stage generates the highest possible ModR/M offset and the highest possible ModR/M address that can be accessed by the instruction. The ModR/M offset is used to check the segment limit in the D-cache access stage. The highest possible ModR/M address is generated to check for possible page faults that may be due to a data access on the page boundary. To facilitate the generation of these two addresses we make use of two other adders.

Stack Address Generation

We use a three-input stack adder (2.6 ns delay) to generate the address to push into or to pop from. The three inputs to the stack adder are:

1. Stack segment register value.
2. ESP value.
3. An immediate value based on the operation type and operand size (0 for POP-type stack operations, -2 for 16-bit PUSH-type operations, -4 for 32-bit PUSH-type operations).

In addition to the normal stack address that will be sent to the memory, this stage generates two more stack addresses: high stack address and low stack address. These are respectively highest and lowest possible locations in the stack that the instruction can access. They are used to check for page faults in the dcache read stage.

IDTR and REP MOVS Address Generation

The address generation stage contains two special modules to generate addresses needed by interrupt/exception handling mechanism and REP MOVS instruction. The IDTR address

is used by LIDT1 and LIDT2 micro-operations, which access the IDTR to redirect the fetch into the correct insterrupt service routine. REP MOVS instruction requirtes two addresses (source and destination) and each address can cause a page fault or segment limit violation. Hence, there are special adders that to calculate the highest possible source and destination addresses.

Early Branch Resolution

All Jcc and JMP instructions that do not require register or memory access are resolved in this stage. A redirect signal is sent to the fetch stage to latch the new EIP, EIP+CS, and VIP+CS values into their correponding registers.

2.2.5 D-cache Access Stage

The block diagram of this stage is given in Figure 7. Main functions of this stage are:

1. Read data from the memory subsystem.
2. Stall in case an older instruction in the pipeline writes to memory.
3. Perform TLB accesses and segment limit checks and generate exception signals.

We elaborate on the D-cache organization when we discuss the memory subsystem. Here, we note that D-cache is a shared resource between the D-cache Access Stage and the Writeback Stage. The D-cache access stage needs to read data from the cache and write-back stage needs to write data into the cache. These two accesses need to be mediated. Due to the out-of-date technology of the RAM parts given in the library, our circuit technology cannot support reads and writes at the same time, even to different RAM locations. To simplify the design, we give priority to cache writes over cache reads. The main reason for this design choice is the guess that many of the writes will hit in the cache (because they were read two stages back). If this guess is correct, the pipeline needs to stall only for one cycle. In the worst case, the write and read accesses may conflict in the cache which will cause a stall that can last hundreds of cycles (due to bus access). We discuss other possible solutions in the section where we discuss our design tradeoffs.

Memory Ordering

This stage ensures correct memory ordering by stalling the previous stages in case the instruction in this stage needs to read memory while an instruction in one of the later stages needs to write memory. The stall signal is generated by comparing the read_memory signal of the instruction in the D-cache stage with the write_memory signals of the instructions in Execution and Writeback stages.

Exception Signal Generation

Two different hardware modules are used in this stage for exception checking. One is

the segment limit checker which compares the ModR/M offset to the segment limit. The other is the TLB which generates either a page-fault or general protection exception. The TLB is a heavily-ported structure which takes as input 8 addresses and the operation type and checks if any of those addresses cause an exception.

2.2.6 Execution Stage

The high-level block diagram of the Execution Stage is shown in Figure 8. This is the stage where bulk of the work gets done. Almost all instructions (other than some branches) calculate their results to be committed to architectural state in this stage.

The execution stage consists of two main parts:

1. Modules to generate results to be written into GPR register file, memory, and segment registers.

2. Modules to generate target EIP and EIP+CS.

All branch-type operations that are not resolved in the Address Generation Stage (CALL, indirect JMP, IRET micro-operations, RET) determine the target EIP and EIP+CS in this stage using the branch address generation logic. The target EIP is compared with the CS limit to detect the possible general protection exceptions.

Other operations each all have their specialized and tailored modules to perform the execution of each operation as fast as possible. Each of these modules generate results for memory, destination registers, and flags. Eventually, the correct value is selected using the operation type signal that is provided by the decoder.

2.2.7 Writeback Stage

This is the stage where all updates to the architectural state is performed. Architectural state consists of the general purpose registers, segment registers, flags, memory subsystem, and main memory. An instruction in this stage updates the architectural state only if there are no exceptions generated by this instruction and there are no pending interrupts. If there is an exception or an interrupt, the instruction is inhibited from updating the architectural state and the writeback stage sends an exception_handle signal to the decoder, which will start inserting micro-operations that will facilitate branching to the correct interrupt service routine.

One function that is performed by the writeback stage is the alignment of values to be written into the general purpose register file. This alignment is done based on control signals from the decoder (whether this instruction writes 8, 16, 32 bits and the register id to be written into). As discussed in Section 2.2.3, this alignment is required because the granularity of writes into the register file is 32-bit.

Writeback stage is the other stage that can redirect the fetch stage. It supplies the target EIP, EIP+CS values to the fetch stage along with a signal that redirects the fetch engine. If a branch-type instruction needs to wait until writeback stage to be resolved, a 7-cycle bubble will be introduced into the pipeline which degrades performance in the presence of such branches. We note that this penalty is only 4 cycles for Jcc and non-indirect JMPs which are relatively more common.

A high-level block diagram of the Writeback Stage is provided in Figure 9.

2.2.8 Interrupt/Exception Handling

Our approach to handling of interrupts and exceptions is broken into several tasks:

1. Detection of interrupts and exceptions.
2. Determination of which interrupt/exception to handle.
3. Branching to the correct interrupt service routine.
4. Return from interrupt service routine using the IRETD instruction.

We have discussed the detection mechanism for exceptions in the previous sections. Each pipeline stage has some logic to generate and/or propagate the exception signals for Page Fault exception and General Protection exception. These signals eventually reach the writeback stage and are checked in that stage. Interrupts are events that are external to the processing core. They can come at any time from a peripheral device. To ensure correct handling when the processing core detects that the interrupt signal is high in one of its interrupt pins, it latches the signal. The latched interrupt signals are checked in the writeback stage.

Multiple exceptions or interrupts may be present when the writeback stage checks the existence of an exception or an interrupt. We prioritize exceptions over interrupts for handling. Page Fault is prioritized over General Protetcion exception. One of the interrupt signals (INT1) has priority over the other (INT2). Interrupt handling logic at the writeback stage conveys the information as to which interrupt/exception to be handled to the decoder based on this priority mechanism.

Once the writeback stage detects that an exception or interrupt is present it sends a flush signal to all pipeline latches which invalidates all the older instructions. Each pipeline latch has a valid bit associated with it to support this flush operation (and also to support other stalls). The current instruction in the writeback stage (excepting instruction) is not allowed to update the architectural state. The EIP and CS carried along with the excepting instruction is saved in the latches in the interrupt/exception handling logic. The flush signal changes the control in the decoder. The decoder starts inserting micro-operations into the pipeline which will save EFLAGS, EIP, and CS on the stack and load

the new EIP and CS values from the IDTR entry associated with the exception/interrupt type. These micro-operations, in the order of their insertion, are:

1. PUSH EFLAGS (Pushes the EFLAGS register on the stack)

2. PUSH CS (Pushes the CS register on the stack)

3. PUSH EIP (Pushes the EIP register on the stack)

4. LIDT1 (Performs a doubleword access to get the first doubleword of the IDTR associated with the exception/interrupt)

5. LIDT2 (Performs a doubleword access to get the second doubleword of the IDTR associated with the exception/interrupt - writes into EIP, CS, EIP+CS registers, and redirects fetch stage)

Once LIDT2 is inserted, the decoder stops stalling the front end and returns to normal mode where it inserts a single operation into the pipeline based on the instruction in the instruction register. All exceptions and interrupts are disabled until IRETD is executed.

Returning from the interrupt service routine is initiated using the IRETD instruction. As discussed in Section 2.2.2, this instruction is broken into three micro-operations, last of which is POP EFLAGS. The execution of POP EFLAGS enables interrupts and exceptions again.

2.2.9 Branch Handling

If a branch instruction is decoded and as it flows through the pipeline, fetch stage is stalled until the branch instruction is resolved. This is accomplished using stall signals from all stages except for writeback indicating that an unresolved branch is currently in that stage. Fetch stage is stalled for all operations that change either the EIP or the CS value.

2.2.10 Specialized Hardware to Execute REP MOVS

Figure 10 shows the special hardware located in Address Generation stage and used to execute REP MOVS without stalling between iterations (due to each iteration writing and sourcing the same registers - ESI, EDI, ECX). The intermediate values of EDI, ESI, and ECX are calculated using extra adders in the Address Generation stage and they are latched in temporary registers. Incoming iterations of REP MOVS grab the values of these registers from the temporary registers and can execute without stalling.

This figure also shows the state diagram for REP MOVS execution. We use the decode of the REP MOVS instruction as an entry point into this state diagram. Decoder remains in these special states until ECX becomes zero.

The decoder generates two different micro-operations for REP MOVS. One is the REP_FIRST micro-operation, which denotes the beginning of REP MOVS. The second is the REP_ITER micro-operation used for all other iterations of REP MOVS. And the last one is the REP_DONE micro-operation that signals that REP MOVS is done and clears the pipeline.

2.3 Memory Subsystem

Memory Subsystem is the part of our system which resides on-chip and caches data. It consists of two single-ported (read or write), 512-byte, direct-mapped caches: Instruction cache (I-cache) and the Data cache (D-cache). It also consists of an 8-entry TLB. We discuss the implementation and control of each of these units in the following sections.

2.3.1 I-cache

The Instruction cache supplies instructions to the fetch unit. It contains 16 bytes in each line, which is the maximum size of an instruction. We chose 16 bytes as the block size for the I-cache because we thought a larger block size would require a wider bus, a wider memory or a longer time to retrieve data from memory.

The instruction cache has a simple organization with 32 blocks. The cache is virtually-indexed and physically-tagged. Both the data store and the tag store are indexed using bits 8 through 4 of the virtual address. Top 6 bits of the physical address are used as the tag and are compared after the physical frame number corresponing to the virtual address is available. Tag store also contains 1 bit indicating whether or not the block is valid.

The state diagram for the instruction cache is shown in Figure 11. On a cache miss, the I-cache controller requests the bus and waits for a BG (bus grant) signal. When bus is granted to the I-cache controller, it asserts BBSY (Bus Busy) and sends the address to the memory and starts waiting for the data and acknowledgement from memory. Once the memory controller responds with the "Done" signal, the I-cache controller, grabs the 128 bits from the bus and writes them into the cache and sets the valid bit of the entry.

Attached to the I-cache controller is an alignment logic which alignes the supplied instructions if the access address is not aligned within cache boundaries.

2.3.2 D-cache

D-cache has a similar physical structure to I-cache in that its block size is 128 bits, it is direct-mapped, and indexing and tag match occur the same way as in the I-cache. However, the control of D-cache is more complicated than the I-cache due to three reasons:

1. D-cache needs to support unaligned accesses.
2. D-cache is writable by the processing core.
3. D-cache controller needs to support accesses to non-cacheable data.

Figure 12 shows the state diagram of the D-cache controller. Unaligned accesses to the D-cache can occur on word (16-bit) or doubleword (32-bit) accesses. In case of an unaligned access, the D-cache controller makes two accesses to the D-cache.

Writes to the D-cache cause the cache line to become dirty. A dirty bit is added to the tag store of the D-cache for this purpose. When a cache line is allocated on a write, the dirty bit of the block is set automatically on allocation.

Non-cacheable writes or reads do not write the data into the D-cache. Instead, the D-cache controller provides some states to bypass the write to D-cache on any kind of non-cacheable access.

D-cache interfaces to the Bus Arbiter the same way I-cache does. However, accesses of the D-cache to the bus arbiter are prioritized over an I-cache bus request. This design choice was made to decrease the number of stalls in the pipeline due to instructions needing data to operate on.

The D-cache controller aligns the data accesses based on the access size. For read accesses it provides the data at the least significant bytes of the 128 bits. For write accesses, the alignment is done by accessing a ROM indicating where the byte(s) written should go in the cache line(s).

2.3.3 TLB

The TLB is the on-chip cache of the most recently used 8 page table entries. 6 of these entries are cacheable entries. 2 of them are non-cacheable and are used to perform memory-mapped I/O. One of the non-cacheable entries is allocated for the keyboard and the other is allocated for the monitor.

Each entry in the TLB contains a valid bit, a present bit, a read-only bit, a non-cacheable bit, and two bits indicating which device the address is mapped to if the entry is non-cacheable.

The TLB in our design is an 8-ported structure which is accessed by many addresses many of which are used to check exceptions.

2.4 Bus Arbiter, Bus, Memory, and Peripherals

The third and the most distributed component of our system consists of the bus arbiter, the bus, the main memory, and attached peripheral I/O devices. The bus arbiter resides on-chip and accepts requests from the I-cache or the D-cache. All the other components reside off chip and communicate with the processing core using the bus or interrupt lines. At any given time, only the processor core (through the I-cache or the D-cache) can be the master of the bus. We have implemented two devices that can act only as slaves. When these devices require to communicate with the processing core, they raise their dedicated interrupt line and the processor communicates with the devices through the system bus. Memory controller is also similar to an I/O device except that it cannot interrupt the processing core. The communication paths between these components are clearly sketched in Figure 1.

We examine each of these components in more detail.

2.4.1 Bus Arbiter

We use a centralize, synchronous arbitration scheme to determine which device controls the bus.The logic used to perform bus arbitration is shown in Figure 13. This logic takes two bus request signals, one from the I-cache, the other from the D-cache. It also takes as input the BBSY signal inditcating whether or not the bus is busy in the current cycle. Based on these signals it determines whether or not to grant the bust to the I-cache or the D-cache. If both I-cache and D-cache request the bus at the same time, the arbiter gives priority to the D-cache request to prevent the instruction from starving and pipeline from stalling.

Once the I-cache or the D-cache receive the BG (Bus grant) signal, they assert the BBSY signal until the bus transaction is complete.

2.4.2 System Bus

The bus itself runs at the same speed at the processor. It is divided into three logical parts: Data bus, Control bus, and Address bus.

The Data bus is 128 bits wide and is used to transfer data bidirectionally between memory and the processing core or unidirectionally between the processing core and I/O devices.

The Address bus is only 15 bits. The largest storage structure connected to the bus is the main memory, which is 32 KB and can be fully addressed using 15 bits.

The Control bus is 7-bit wide contains the following signals:

1. BBSY (indicates whether a transaction is in progress)

2. Device ID (2-bit signal indicating which device needs to access the data)
3. Interrupt lines (2 bits for each I/O device)
4. ACK (Acknowledgement signal which tells that the data is loaded onto the bus)
5. BRW (indicates whether this is a read access or a write access)

2.4.3 Main Memory

The main memory of our system is 32 KB. It is organized into 8 banks each of which has the capability of supplying 128 bits. The granularity of data supplied by memory is 128 bits. Top 3 bits of the address is used to select the bank to access.

The memory controller state diagram is shown in Figure 14. Memory controller initiates an access to the memory when the device id on the bus is equal to the id of the memory controller. When the access to the memory is complete, the memory controller sends an ACK signal to the processing core and loads the 128 bits onto the bus. The memory controller decides that the access is complete based on the firing of a timer. The timer is set to its maximum value when the access to the memory is initiated. When the timer counts down to 0, the access must have been complete and the memory controller deduces that the main memory is ready and asserts the ACK signal on the Control bus.

2.4.4 Keyboard Controller

The keyboard controller state machine is shown in Figure 15. Keyboard controller contains a 256 entry buffer where each entry is 8 bits. This buffer holds input data read from the keyboard. When new data is read into the buffer, the keyboard controller raises its interrupt signal. The processor, using an interrupt service routine (number 0), performs a non-cacheable MOV from the buffer of the keyboard controller into the processor. The keyboard controller lowers the interrupt signal when the buffer is read by the processor.

2.4.5 Monitor Controller

The state machine for the monitor controller is shown in Figure 16. The monitor controller also holds a buffer containing the data sent by the processing core. The controller buffers the data when the device id on the control bus matches the device id of the monitor controller (id = 3). If the processor overruns the mnitor controller buffer, older data in the buffer is discarded.

3 Design Tradeoffs

In this section, we describe the important tradeoffs we made during our design and why we made those tradeoffs. Some of these tradeoffs turned out to be unnecessary due to the unanticipated critical path, but they were applicable to a 5-ns cycle time target.

3.1 Design Goals

Of course, our main design goal was to design the system such that it works correctly for all instructions and conditions. One secondary goal was to achieve a 5-ns cycle time. Another important goal was to make the hardware as simple, streamlined, and understandable as possible.

3.2 Tradeoffs in the Processing Core

We spent most of our time to optimize the architecture and logic of the processing core. We will outline the main tradeoffs we made in each stage.

3.2.1 Fetch Stage Tradeoffs

The main tradeoff we faced in the design of the fetch stage was when to shift the instruction register. As we discussed earlier, we decided to shift this register based on old information first and use the late coming signals from the decoder latest so that we do not unnecessarily stretch the cycle time. This design choice was favoring cycle time over a one-cycle stall in a design oriented for 5-ns cycle time. When the IR is more than half full, the incoming bytes from the I-cache are not concatenated with the bytes in IR to be shifted. The determination of how many bytes are inside the IR is made based on stale information, which can possibly cause stalls due to not latching valid data from the I-cache into the IR. However, now that we know this loop is not on the critical path, we believe we could have used the "number of instructions consumed" signal from the decoder to decide whether or not to incorporate the incoming bytes from the I-cache without stretching the 7-ns cycle time.

Another design decision we made was to keep the size of the instruction register at 32 bytes (256 bits). We could have increased the size of the IR so that it could buffer more bytes. This would be more useful if we increased the block size of the I-cache. Such a decision would definitely increase the delay because we would be shifting more data. We chose not to have an IR larger than 32 bytes so as not to increase the cycle time. However, with the 7-ns cycle time we have, we could increase the size of the IR concurrently with the size of the I-cache block.

3.2.2 Decode Stage Tradeoffs

Our main goal in the decode stage was to optimize the logic that generates the number of bytes consumed signal such that the loop between fetch and decode that uses this signal would be kept under 5 ns. We achieved our main purpose by doing a lot of decoding in parallel and selecting the result of the parallel decoders. The hardware complexity of the decode stage is therefore quite high.

Another signal we optimized for in the decode stage was the stall signal, which was discussed in Section 2.2.2. This signal turned out to be the latest signal generated by the decoder (with a delay of 4.73 ns as shown in Figure 4).

Our choice of breaking hard-to-handle instructions into micro-operations turned out to be a very good design decision in that it simplified the control of the pipeline and made the pipeline more regular. Some of the complicated instruction share the micro-operations, which relieved us from generating new control signals. Especially handling of exceptions/interrupts using micro-operations made the control of the pipeline much simpler than how it would be otherwise.

3.2.3 Register Read Stage Tradeoffs

We decided to dedicate one full stage for register access to be able to do forwarding without stretching the cycle time. Excluding the signals coming into this stage from other pipeline stages, the register file read stage takes shortest amount of time on its critical path.

In hindsight, we could perhaps have eliminated this stage by optimizing the logic for a 7-ns cycle time and by merging the logic in this stage with the logic in address generation stage. This would have resulted in a smaller branch misprediction penalty.

Another tradeoff we made in this stage was the granularity of reads and writes for the register file. We chose this granularity to be 32 bits, which would cause some unnecessary stalls if the instructions are sourcing exclusive parts of the 32-bit register. We believe that this case does not happen frequently. Our design choice favors simplicity over performance for rare cases.

As to the forwarding paths, we decided not to forward the flag and segment register values from other stages. We also decided to forward destination register values only from the execution and writeback stages. These design choices definitely effect the CPI (cycles per instruction) of the processing core, but by how much they affect the CPI depends on the program characteristics.

We chose to perform limited register renaming in this stage by using the instruction tags generated by the decoder. This let us have multiple instructions in the pipeline writing into the same register. Hence, our pipeline does not stall n a WAW (Write-after-write)

dependency.

3.2.4 Address Generation Stage Tradeoffs

The main reason that made us keep address generation in one stage was the fact that we had an optimized 2-input adder whose critical path was 2 ns. If we had a slower adder, we would have thought of breaking address generation into two stages.

3.2.5 D-cache Access Stage Tradeoffs

We followed a conservative approach in memory ordering. If there is an older store in the pipeline that is not yet committed, the younger load stalls the pipeline. A less conservative solution would be to check if the address the store is writing to matches with the address the load is reading from. Such a check poses some difficulties due to the different granularity of writes. Also, in our pipeline, a more aggressive memory ordering mechanism would probably not decrease the CPI by much, because most stores would already hit in the cache and hence the stalls due to memory ordering will be limited to 1 cycle. The reason why most stores will hit in the cache is that most instructions read the memory location before they write into it. This is only not true for MOVs into memory.

3.2.6 Execution Stage Tradeoffs

Our design of the execution stage as separate units which generate results and operation type signal eventually selecting correct results lead to very simple control at the expense of more hardware. It also decreased the debugging effeort we would have spent by having a large monolithic unit and selecting its inputs and outputs using muxes.

3.2.7 Writeback Stage Tradeoffs

We decided to dedicate a full pipeline stage for the update of the architectural state. This has advantages in terms of supporting precise exceptions. Our system never experiences out-of-order writes. Therefore, we do not need to recover any state on an exception. Another way we could handle the update of architectural state was using a reorder buffer or a history register. We did not follow these approaches so as not to complicate the design.

One design choice we should have made was to incorporate a store buffer for handling stores to memory in order to reduce D-cache contention and give priority to reads. This would complicate the design, but decrease the CPI perhaps dramatically.

3.2.8 Optimizations

We tried to optimize the logic in every module that we thought would be on the critical path. We especially devoted careful attention to the signals generated by the decoder and optimized the logic. For example, to keep the critical path of the decode-fetch loop under 5 ns, decoder generates two different encodings of the instruction size in parallel and independently of each other. One encoding is a one-hot encoding of the instruction size, which is used to control the shifting of the concatenated IR in the fetch stage. The other encoding is a two's complement encoding used to update the new value of the number of valid bytes in the IR.

To keep the cycle time under 5 ns, we needed to have a very fast adder. The 32-bit 2-input adder we built runs at 2 ns. It is a hybrid Carry-Select adder made up of four 8-bit Carry Lookahead Adders. We also built a 3-input 32-bit adder with 2.6 ns delay using redundant binary arithmetic. Our 4-input adder was not as optimized, but it ran at 3.8 ns, which is quite fast compared to the targeted cycle time.

One stage where we could have optimized more was the D-cache controller and its generation of the hit/miss signal, which was on the critical path of our design.

3.3 Tradeoffs in the Memory Subsystem

Memory subsystem is the most important part of our system that would result in a huge performance benefit, if we had more time to optimize the logic and to implement more aggressive design choices. The bulk of this performance benefit will come from optimizing the D-cache. Therefore, we will only focus on D-cache design tradeoffs here.

We decided to implement a direct-mapped D-cache due to its speed and simplicity. We now feel that we should have at least supported it with an auxiliary structure (like a victim cache or store buffer) or made the D-cache 2-way associative. On the sample programs we wrote, our direct-mapped cache seems to suffer from conflict misses, which leads to very high CPI's. The disadvantage of an auxiliary structure is the more complex control. The disadvantage of 2-way set associativity is the increased delay. However, we believe that both increased control or increased delay are worth the price for reducing the conflict misses.

Another design choice we could have made was increasing the block size of caches. This would likely lead to lower miss rates on both the I-cache and the D-cache. However, such an organization would have increased the contention on the bus and the access delay of the memory.

One other way of organizing our cache was by banking it. This way, the cache would be able to handle concurrent read and write if the read and write were to different banks. This also would have increased the performance of our implementation. However, we

would still need to stall when the read and write conflicted in the same bank. So, the extra complexity due to the RAM cell having only one port would still remain.

3.4 Tradeoffs in the Bus Arbiter, Bus, Memory, and Peripherals

We decided to specialize our bus arbitration logic for the requirements of our system. Only the processor can be the bus master. As a consequence of this design choice, addition of new peripherals (such as DMA) onto the bus may require some effort and changes to the arbitration logic.

Our bus was 128 bits, which is the same as the block sizes for the I-cache and the D-cache. This simplified the complexity of the bus. All the data communicated on the bus is communicated based on a cache-block granularity (except for I/O devices). Therefore, we do not need any other control signals on the bus that denote the granularity of transferred data.

4 Critical Path Analysis

The Critical Path in this pipeline is primarily caused due to the delay in getting a stable stall signal from the DCache. There is an initial 1 ns worst case delay in giving a stable address to the Cache. The cache takes 3.5 ns to do a tag compare and check if the address hits in the cache. The stall signal from the DCache is dependent on this signal which decides the next state for the dcache. As the stall signal logic is dependent on the next state, there is an additional 1.55 delay to give a stable stall signal.

The pipeline latches are dependent on this stall signal and they are generated within 1 gate delay. This gives us a worst case delay of about 6.4 ns. The rest of the clock is for the latch delay (around 0.45 ns). We clocked our system at 7 ns for safety reasons (for example, in the presence of some process disturbance, the delay on the critical path may be a little higher than what it is calculated to be).

The critical path timing diagram (at the end of the Appendix) shows us an example of the critical path delay. The DC.cache.hit is the hit signal in the cache. dcache_stall is the stall generated by the DCache. DE_values_latch_we, AG_valid_we, FE_WE_IR, FE_WE_eip are the latch write enable signals.

This path can be opimized by optimizing the logic that generates the dcache_stall signal based on DC.microseq.miss and DC.cache.hit signals.

Table 5 in Appendix shows the delay of each pipeline stage. The first column shows the actual delay from the inputs to the stage to the latching of the final results. This, of course includes the signals coming into the stage from other stages than the previous one. We

can see that the critical path delays of all stages are very close to each other. The reason for this is that D-cache stall signal is sent to all stages to write-disable the latches in case of a D-cache miss. However, the decode stage latch has the worst delay, as discussed in the previous paragraphs.

The second column in Table 5 shows the delay of each stage (including the latch overhead of 0.45 ns) without including any stall signal coming from other stages. We can see that these delays are well below 7 ns and little above 5 ns, which was our targeted cycle time.

5 Limitations

This section discusses the limitations of our design and suggests what can be improved and how. We believe that limitations in the memory subsystem contribute to the performance degradation far more than the limitations in the processing core. Therefore, we will discuss the limitations of memory subsystem first, although some of those limitations relate closely to the processing core.

5.1 Memory Subsystem Limitations

The main limitations of our design were due to our design choices of the memory subsystem. There are three important problems our memory subsystem suffers from:

1. Conflict misses in the D-cache degrade the hit rate of the cache.

2. Stores that miss in the D-cache consume available bus bandwidth (and stall the processor).

3. Small I-cache line size results in too many I-cache misses.

4. Long critical path of the D-cache controller.

To solve the first problem, i.e. conflict misses, we could have extended our D-cache design to make it set-associative. Another way we could solve the problem was by including a victim cache which would be accessed in parallel with the D-cache. Both of these approaches adversely impact the cache access time, which is on our critical path. However, the adverse effect on cycle time is limited to a 2-input mux delay. If this delay is not tolerable, we could use a victim cache which is accessed on a miss on the D-cache before going to the memory.

The second problem, i.e. stores consuming available bus bandwidth and stalling the processor, can be alleviated by using a store buffer for pending stores. The D-cache would again be accessed in parallel with the store buffer, which extends the cycle time of the processor. However, our pipeline would not stall if a store missed in the D-cache.

The missing store would be inserted into the store-buffer and would be scheduled onto the bus when there is available bus bandwidth.

The third problem, i.e. high number of I-cache misses, can be solved by increasing the size of each I-cache block. We could do this by increasing the width of the data bus and memory along with the line size of the I-cache. In that case, no additional complexity would be added into the system. However, if we keep the data bus and memory width at 128 bits while increasing the I-cache line size, then fetching of one cache line from memory would require two memory accesses, which would take quite long considering that the SRAM part provided does not have support burst mode. An increased I-cache line size also requires enough latency tolerance in the fetch unit to be able to deal with more instructions coming in.

The fourth problem is the critical path of the D-cache controller, which is the limiter of our cycle time. To achieve a higher clock frequency we should optimize the way miss signal is generated in the D-cache. We have identified the ways to optimize this signal, which would reudce the cycle time by around 0.8 ns, but we have not implemented this optimization due to time considerations.

We believe all of these problems need to be tackled first to reduce the CPI of our system, because they form major bottlenecks.

Another problem that may be addressed later is to support simultaneous reads and writes to the cache lines. This is harder to implement without the existence of dual-ported RAM cells. However, one way to alleviate the problem is to bank the cache, which requires significantly more complex design.

5.2 Processing Core Limitations

We discuss three very important limitations regarding the processing core.

One important limitation of the processing core is related to the rate of instruction supply to the core. In the previous section, we suggested that by increasing the I-cache line size, we can increase the instruction supply rate significantly. To make the impact of increasing the instruction cache line size, we can also make the instruction register wider. This way, we could overlap some of the I-cache miss latency with useful work done in the processor.

The way we handle branch-type operations in the processing core adversely impacts the CPI. Although some of the branch operations are resolved early, they still result in a 4-cycle bubble, which is significant. Including branch prediction in our design would have made our processing core faster.

Another important limitation in the processing core is the limited number of forwarding paths. Only execution and writeback stages can forward data to the register read stage, and they can only forward the destination register. Especially dependent ALU operations

suffer from 3-5 cycle stalls even with partial forwarding. We could extend our design to have the capability of executing dependent back-to-back ALU operations by adding another ALU to the Address Generation stage which is capable of executing register-register operations. We would also need to extend our design to include forwarding paths from Address Generation and D-cache access stages.

5.3 Limitations in Bus Arbiter, Bus, Memory, Peripherals

The main limitation in this part of the system is the bus contention due to multiple simultaneous bus accesses. This bus contention occurs when the I-cache and the D-cache require the bus at the same time and results in long stalls in the processor core. A solution to this problem is by providing multiple channels to memory and supporting multiple transactions on the bus. Multiple channels to the memory can be provided by dual-porting the memory (which is hard given that the SRAM parts have only one port) or banking the memory. Supporting multiple transactions on the bus is possible using a more complex bus and memory controller design. However, we feel that these complexities are worth tackling due to the extremely long stalls caused by bus contention.

6 Conclusions

We have explained our design of a system that implements a subset of the x86 ISA. This project has shown us several directions we should follow in our later designs, many of which we discussed in Section 5. Although these limitations are important to address, one reason we could not address them was the complexity of the ISA, which required tedious design and debugging to get to work correctly. The time spent on designing and optimizing the variable-length instruction decoder (which needs to decode instructions whose opcodes do not seem to have any relationship with their function) could be better spent by implementing out-of-order execution on a fixed-length, uniform-decode ISA.

In any case, we have achieved one of our design goals in this project, while falling short on the other one.

7 Appendix

Table 1: Instruction Set

Opcode	Instruction	Description
04	ADD AL, imm8	Add AL to imm8
05	ADD AX, imm16	Add AX to imm16
05	ADD EAX, imm32	Add EAX to imm32
80	ADD r/m8, imm8	Add r/m8 to imm8
81	ADD r/m16, imm16	Add r/m16 to imm16
81	ADD r/m32, imm32	Add r/m32 to r/m32
83	ADD r/m16, imm8	Add sign-extended imm8 to r/m16
83	ADD r/m32, imm8	Add sign-extended imm8 to r/m32
00	ADD r/m8, r8	Add r8 to r/m8
01	ADD r/m16, r16	Add r16 to r/m16
01	ADD r/m32, r32	Add r32 to r/m32
02	ADD r8, r/m8	Add r/m8 to r8
03	ADD r16, r/m16	Add r/m16 to r16
03	ADD r32, r/m32	Add r/m32 to r32
0F C8	BSWAP r32	Reverses the byte order of a 32-bit register
0F AB	BTS r/m16, r16	Store selected bit in CF flag and set
0F AB	BTS r/m32, r32	Store selected bit in CF flag and set
0F BA	BTS r/m16, imm8	Store selected bit in CF flag and set
0F BA	BTS r/m32, imm8	Store selected bit in CF flag and set
E8	CALL rel16	Call near, relative, displacement relative to next instruction
E8	CALL rel32	Call near, relative, displacement relative to next instruction
FF	CALL r/m16	Call near, absolute indirect, address given in r/m16
FF	CALL r/m32	Call near, absolute indirect, address given in r/m32
9A	CALL ptr16:16	Call far, absolute, address given in operand
9A	CALL ptr16:32	Call far, absolute, address given in operand
FC	CLD	Clear DF flag
F4	HLT	Halt
FE	INC r/m8	Increment r/m byte by 1
FF	INC r/m16	Increment r/m word by 1
FF	INC r/m32	Increment r/m doubleword by 1
40	INC r16	Increment word register by 1
40	INC r32	Increment doubleword register by 1
CF	IRETD	Interrupt return
73	JNB rel8	Jump short if not below

75	JNE rel8	Jump short if not equal
0F 83	JNB rel16/32	Jump near if not below
0F 85	JNE rel16/32	Jump near if not equal
EB	JMP rel8	Jump short, relative, displacement relative to next instruction
E9	JMP rel16	Jump near, relative, displacement relative to next instruction
E9	JMP rel32	Jump near, relative, displacement relative to next instruction
FF	JMP r/m16	Jump near, absolute indirect, address given in r/m16
FF	JMP r/m32	Jump near, absolute indirect, address given in r/m32
EA	JMP ptr16:16	Jump far, absolute address given in operand
EA	JMP ptr16:32	Jump far, absolute address given in operand
88	MOV r/m8, r8	Move r8 to r/m8
89	MOV r/m16, r16	Move r16 to r/m16
89	MOV r/m32, r32	Move r32 to r/m32
8A	MOV r8, r/m8	Move r/m8 to r8
8B	MOV r16, r/m16	Move r/m16 to r16
8B	MOV r32, r/m32	Move r/m32 to r32
8C	MOV r/m16, Sreg	Move segment register to r/m16
8E	MOV Sreg, r/m16	Move r/m16 to segment register
B0	MOV r8, imm8	Move imm8 to r8
B8	MOV r16, imm16	Move imm16 to r16
B8	MOV r32, imm32	Move imm32 to r32
C6	MOV r/m8, imm8	Move imm8 to r/m8
C7	MOV r/m16, imm16	Move imm16 to r/m16
C7	MOV r/m32, imm32	Move imm32 to r/m32
90	NOP	No operation
8F	POP r/m16	Pop top of stack into r/m16; increment stack pointer
8F	POP r/m32	Pop top of stack into r/m32; increment stack pointer
58	POP r16	Pop top of stack into r16; increment stack pointer
58	POP r32	Pop top of stack into r32; increment stack pointer
1F	POP DS	Pop top of stack into DS; increment stack pointer
07	POP ES	Pop top of stack into ES; increment stack pointer
17	POP SS	Pop top of stack into SS; increment stack pointer
0F A1	POP FS	Pop top of stack into FS; increment stack pointer
0F A9	POP GS	Pop top of stack into GS; increment stack pointer
FF	PUSH r/m16	Push r/m16
FF	PUSH r/m32	Push r/m32
50	PUSH r16	Push r16
50	PUSH r32	Push r32

6A	PUSH imm8	Push imm8
68	PUSH imm16	Push imm16
68	PUSH imm32	Push imm32
0E	PUSH CS	Push CS
16	PUSH SS	Push SS
1E	PUSH DS	Push DS
06	PUSH ES	Push ES
0F A0	PUSH FS	Push FS
0F A8	PUSH GS	Push GS
F3 A4	REP MOVS m8, m8	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS m16, m16	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI]
F3 A5	REP MOVS m32, m32	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI]
C3	RET	Near return to calling procedure
CB	RET	Far return to calling procedure
C2	RET imm16	Near return to calling procedure and pop imm16 bytes from stack
CA	RET imm16	Far return to calling procedure and pop imm16 bytes from stack
D0	ROR r/m8, 1	Rotate eight bits r/m8 right once
D2	ROR r/m8, CL	Rotate eight bits r/m8 right CL times
C0	ROR r/m8, imm8	Rotate eight bits r/m16 right imm8 times
D1	ROR r/m16, 1	Rotate 16 bits r/m16 right once
D3	ROR r/m16, CL	Rotate 16 bits r/m16 right CL times
C1	ROR r/m16, imm8	Rotate 16 bits r/m16 right imm8 times
D1	ROR r/m32, 1	Rotate 32 bits r/m16 right once
D3	ROR r/m32, CL	Rotate 32 bits r/m32 right CL times
C1	ROR r/m32, imm8	Rotate 32 bits r/m32 right imm8 times
D0	SAL r/m8, 1	Multiply r/m8 by 2, once
D2	SAL r/m8, CL	Multiply r/m8 by 2, CL times
C0	SAL r/m8, imm8	Multiply r/m8 by 2, imm8 times
D1	SAL r/m16, 1	Multiply r/m16 by 2, once
D3	SAL r/m16, CL	Multiply r/m16 by 2, CL times
C1	SAL r/m16, imm8	Multiply r/m16 by 2, imm8 times
D0	SAL r/m32, 1	Multiply r/m32 by 2, once
D2	SAL r/m32, CL	Multiply r/m32 by 2, CL times
C0	SAL r/m32, imm8	Multiply r/m32 by 2, imm8 times
D1	SAR r/m8, 1	Signed divide r/m8 by 2, once
D3	SAR r/m8, CL	Signed divide r/m8 by 2, CL times
C1	SAR r/m8, imm8	Signed divide r/m8 by 2, imm8 times
D1	SAR r/m16, 1	Signed divide r/m16 by 2, once

D3	SAR r/m16, CL	Signed divide r/m16 by 2, CL times
C1	SAR r/m16, imm8	Signed divide r/m16 by 2, imm8 times
D1	SAR r/m32, 1	Signed divide r/m32 by 2, once
D3	SAR r/m32, CL	Signed divide r/m32 by 2, CL times
C1	SAR r/m32, imm8	Signed divide r/m32 by 2, imm8 times
FD	STD	Set DF flag
90	XCHG AX, r16	Exchange r16 with AX
90	XCHG r16, AX	Exchange AX with r16
90	XCHG EAX, r32	Exchange r32 with EAX
90	XCHG r32, EAX	Exchange EAX with r32
86	XCHG r/m8, r8	Exchange r8 (byte register) with byte from r/m8
86	XCHG r8, r/m8	Exchange byte from r/m8 with r8 (byte register)
87	XCHG r/m16, r16	Exchange r16 with word from r/m16
87	XCHG r16, r/m16	Exchange word from r/m16 with r16
87	XCHG r/m32, r32	Exchange r32 with doubleword from r/m32
87	XCHG r32, r/m32	Exchange doubleword from r/m32 with r32
34	XOR AL, imm8	AL XOR imm8
35	XOR AX, imm16	AX XOR imm16
35	XOR EAX, imm32	EAX XOR imm32
80	XOR r/m8, imm8	r/m8 XOR imm8
81	XOR r/m16, imm16	r/m16 XOR imm16
81	XOR r/m32, imm32	r/m32 XOR r/m32
83	XOR r/m16, imm8	sign-extended imm8 XOR r/m16
83	XOR r/m32, imm8	sign-extended imm8 XOR r/m32
30	XOR r/m8, r8	r8 XOR r/m8
31	XOR r/m16, r16	r16 XOR r/m16
31	XOR r/m32, r32	r32 XOR r/m32
32	XOR r8, r/m8	r/m8 XOR r8
33	XOR r16, r/m16	r/m16 XOR r16
33	XOR r32, r/m32	r/m32 XOR r32

Micro-op	Description
FAR CALL1	Push NextEIP
FAR CALL2	Push CS, far jump
POP EIP	Pop EIP
POP CS	Pop CS
POP EFLAGS	Pop EFLAGS
FAR RETIMM	Far Return Immediate
PUSH EFLAGS	Push EFLAGS
PUSH CS	Push CS
PUSH EIP	Push EIP
LIDT1	Load first doubleword from IDT
LIDT2	Load second doubleword from IDT
REP FIRST	First REP MOVS iteration
REP ITER	Intermediate REP MOVS iterations
REP DONE	ECX == 0, clear pipeline

Table 2: Micro-operations

Control signal	Size (bits)	Description
operation	40	one-hot encoding of the operation
src_id	3	register file src port id
src_needed	1	instruction needs to read regfile src port
wr_src	1	instruction needs to write regfile src port
dest_id	3	register file dest port id
dest_needed	1	instruction needs to read regfile dest port
wr_dest	1	instruction needs to write regfile dest port
base_id	3	register file base port id
base_needed	1	instruction needs to read regfile base port
wr_base	1	instruction needs to write regfile base port
index_id	3	register file index port id
index_needed	1	instruction needs to read regfile index port
srcseg_id	3	segment register file src port id
srcseg_needed	1	instruction needs to read segfile src port
addrseg_id	3	segment register file addr port id
addrseg_needed	1	instruction needs to read segfile addr port
destseg_id	3	segment register file dest port id
destseg_needed	1	instruction needs to read segfile dest port
wr_destseg	1	instruction needs to write segfile dest port
uses_imm8	1	instruction uses 8-bit immediate
uses_imm16	1	instruction uses 16-bit immediate
uses_imm32	1	instruction uses 32-bit immediate
uses_imm	1	instruction uses immediate
rd_mem	1	instruction needs to read memory
wr_mem	1	instruction needs to write memory
wr_af	1	instruction needs to write AF flag
wr_cf	1	instruction needs to write CF flag
wr_df	1	instruction needs to write DF flag
wr_of	1	instruction needs to write OF flag
wr_pf	1	instruction needs to write PF flag
wr_sf	1	instruction needs to write SF flag
wr_zf	1	instruction needs to write ZF flag
operand_mode	2	access size/operand size (8, 16, or 32)
shift_left	1	shift is a left shift
cl_needed	1	shift/ror uses CL
chk_exc	1	instruction needs to check exceptions
num_ops_con	2	number of micro-operations after this one
modRMmem_true	1	instruction uses modR/M address
dis_mux_ctrl	2	one-hot control for address generation adder
base_mux_ctrl	2	one-hot control for address generation adder
index_mux_ctrl	5	one-hot control for address generation adder

Table 3: Control signals as output by the decoder

Data signal	Size (bits)	Description
nextEIP	32	EIP after this instruction
nextEIP_CS	32	EIP+CS after this instruction
imm8	8	value of the 8-bit immediate
imm16	16	value of the 16-bit immediate
imm32	32	value of the 32-bit immediate
farptr	48	value of 48-bit far pointer
disp32	32	value of the displacement
tag	8	tag of the instruction
opcode	8	opcode (or second byte of the opcode)

Table 4: Data signals as output by the decoder

Stage	Delay (ns)	Delay w/o signals coming in from other stages
Fetch Stage	6.75	5.15
Decode Stage	6.85	5.10
Register Read Stage	6.70	4.10
Address Generation Stage	6.75	4.80
D-cache Access Stage	6.60	5.10
Execution Stage	6.70	4.55
Writeback Stage	6.70	5.05

Table 5: Delay of each pipeline stage

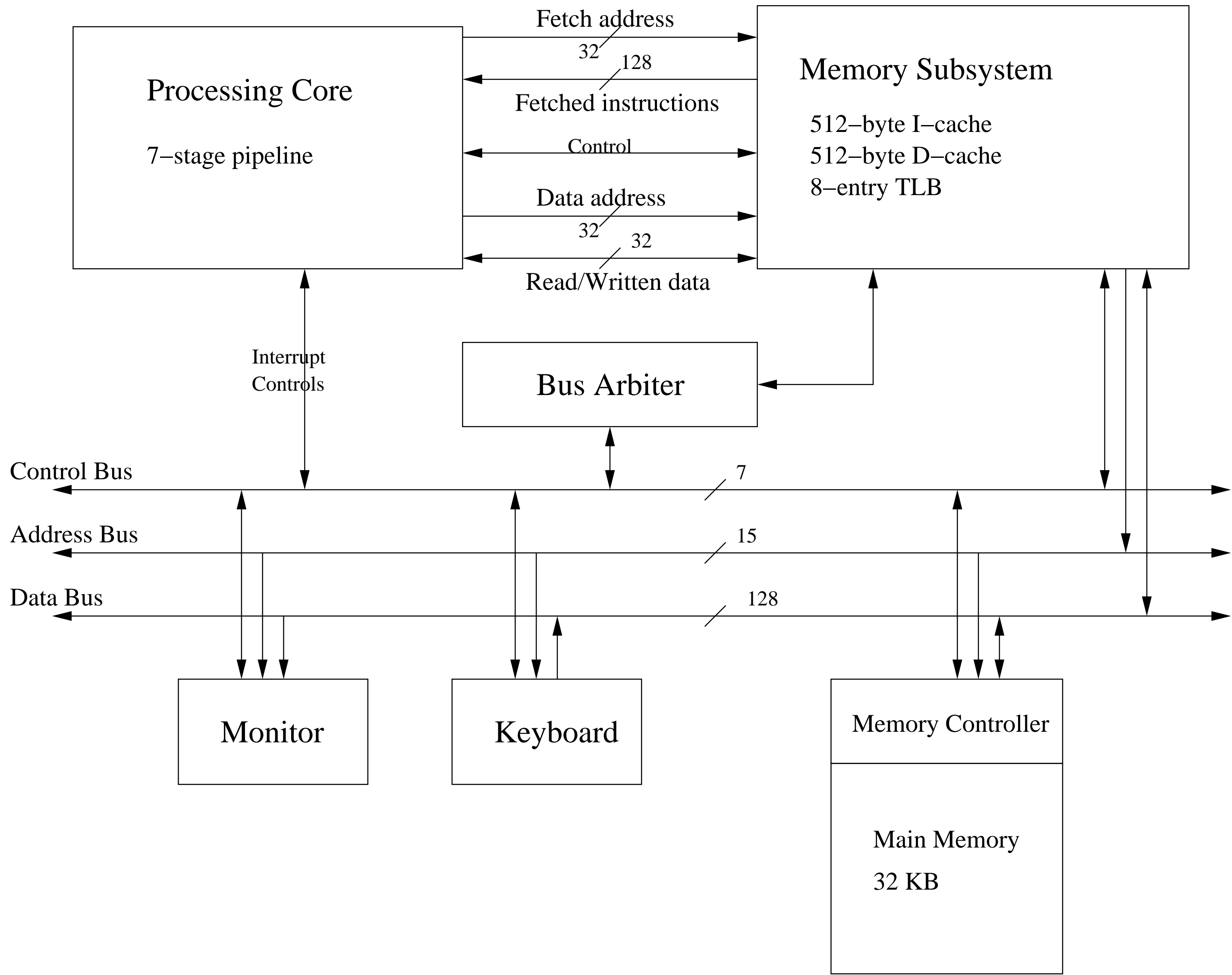

Figure 1: System Overview

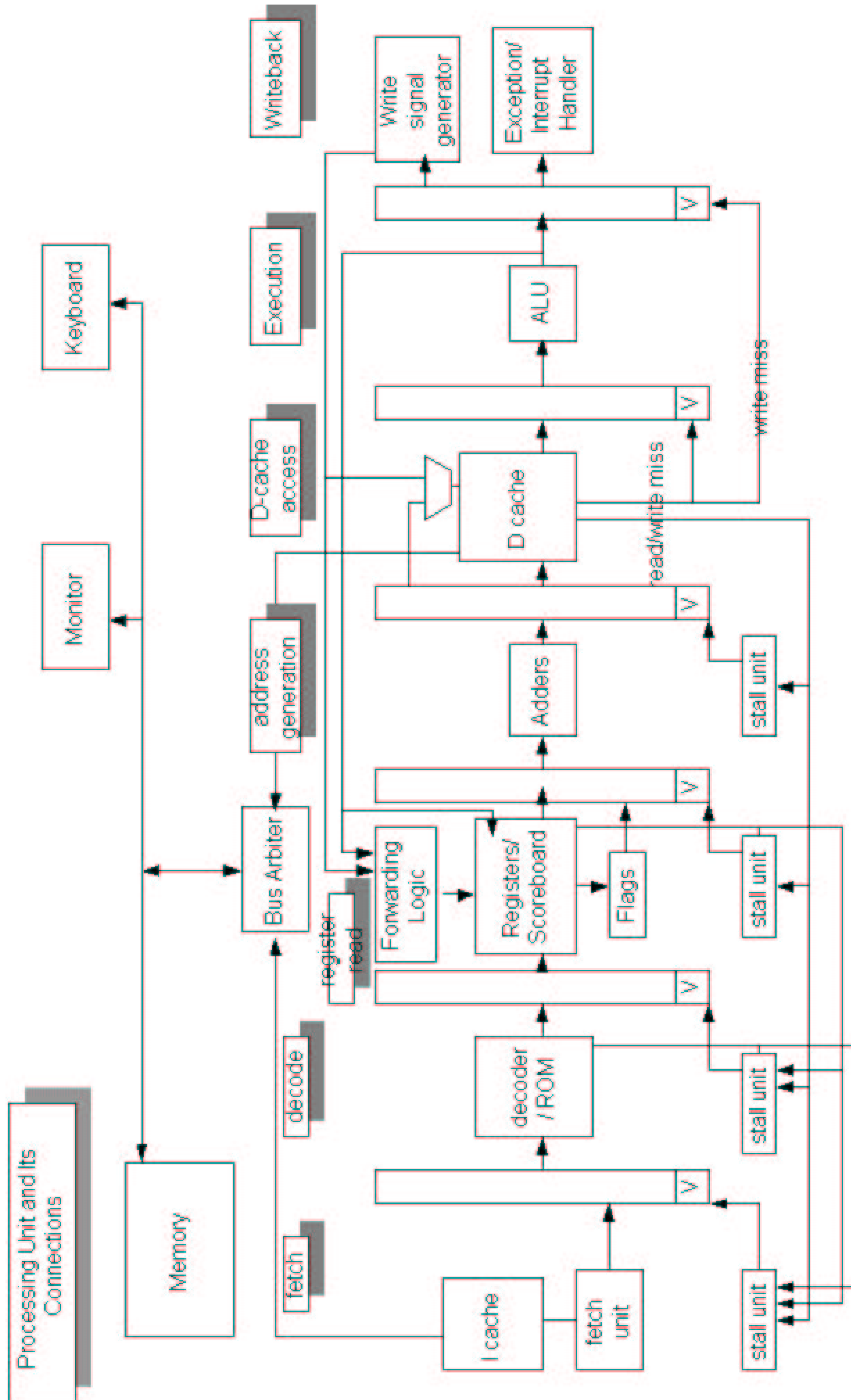

Figure 2: Processing Unit Overview

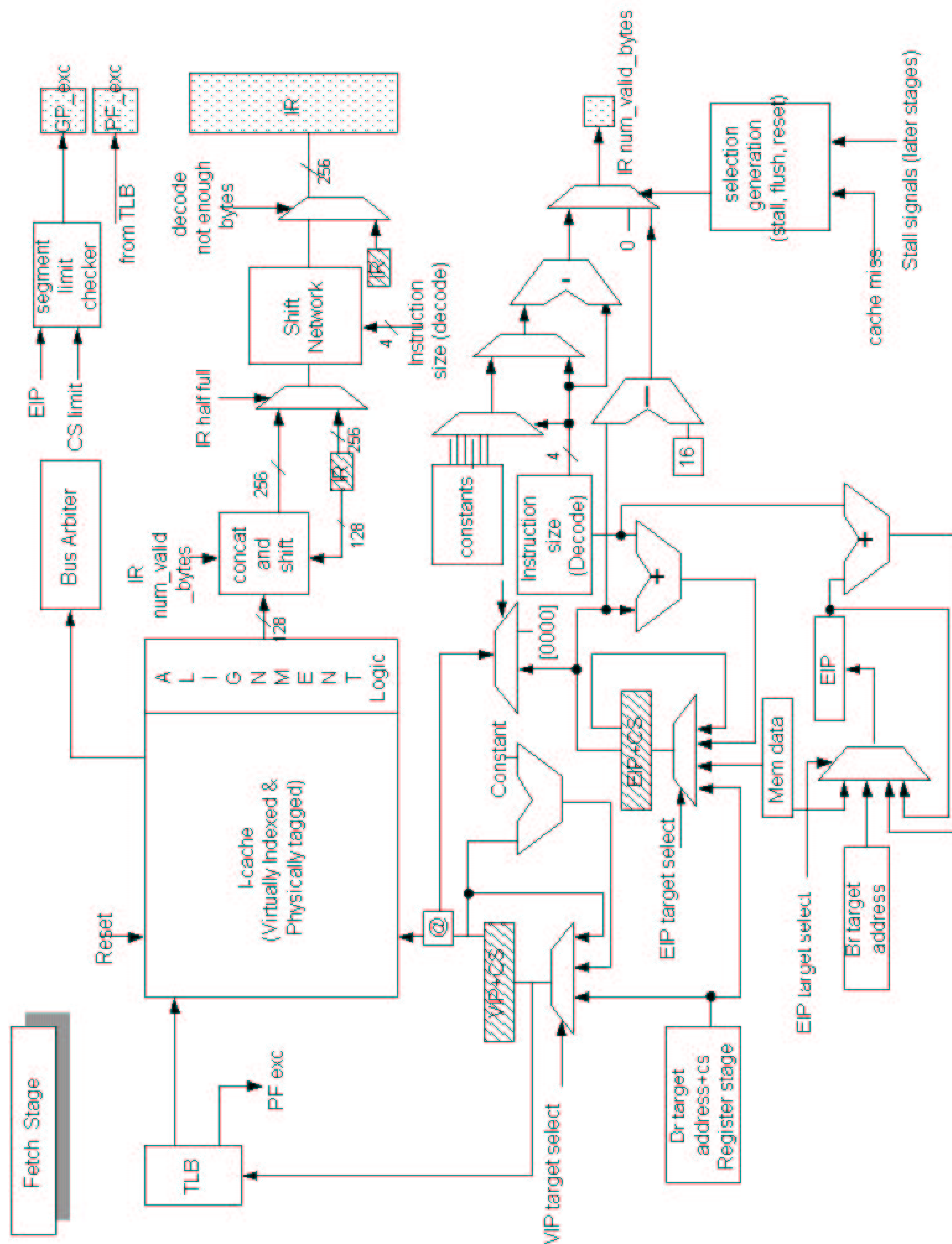

Figure 3: Fetch Stage Block Diagram

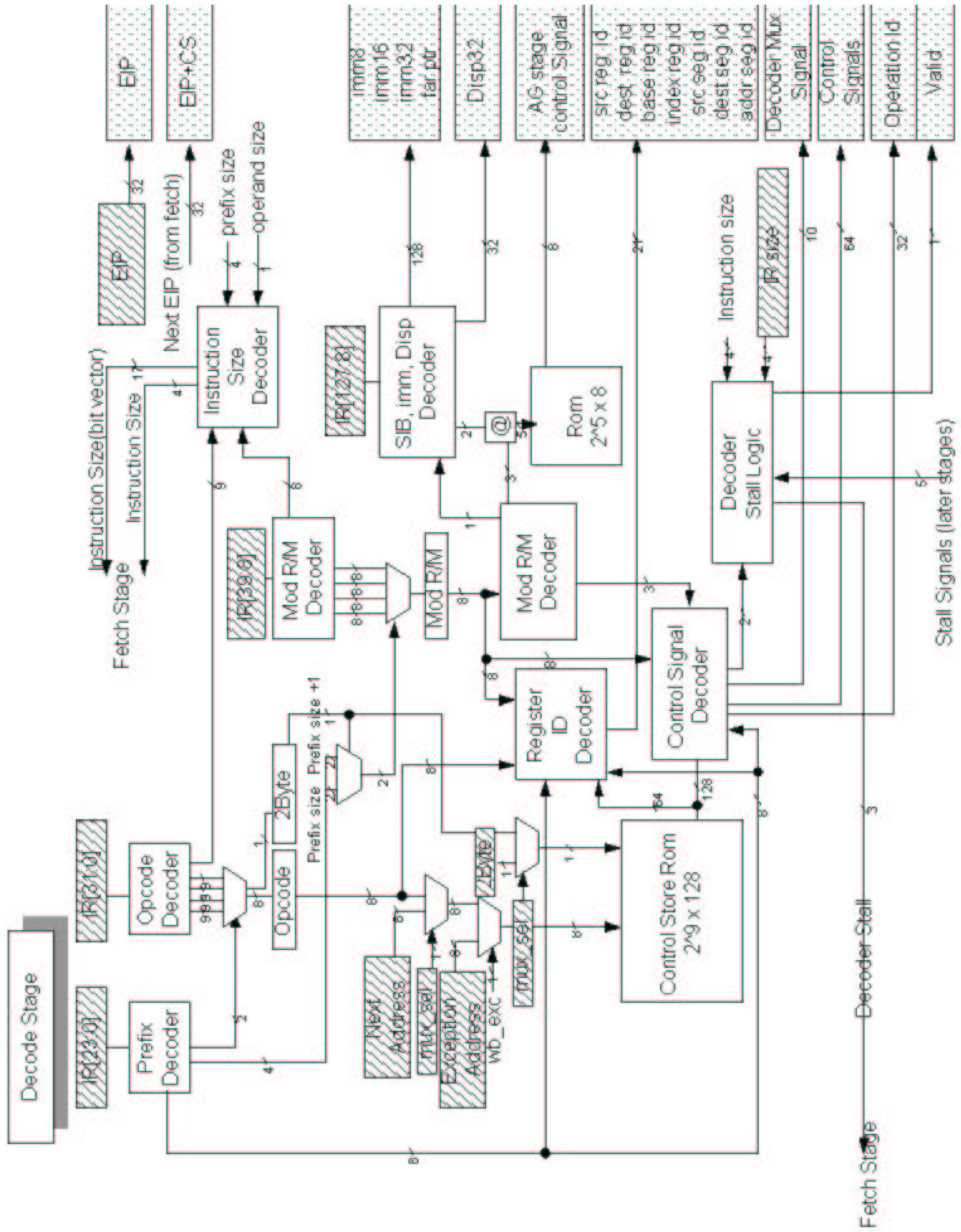

Figure 4: Decode Stage Block Diagram

Figure 5: Register Read Stage Block Diagram

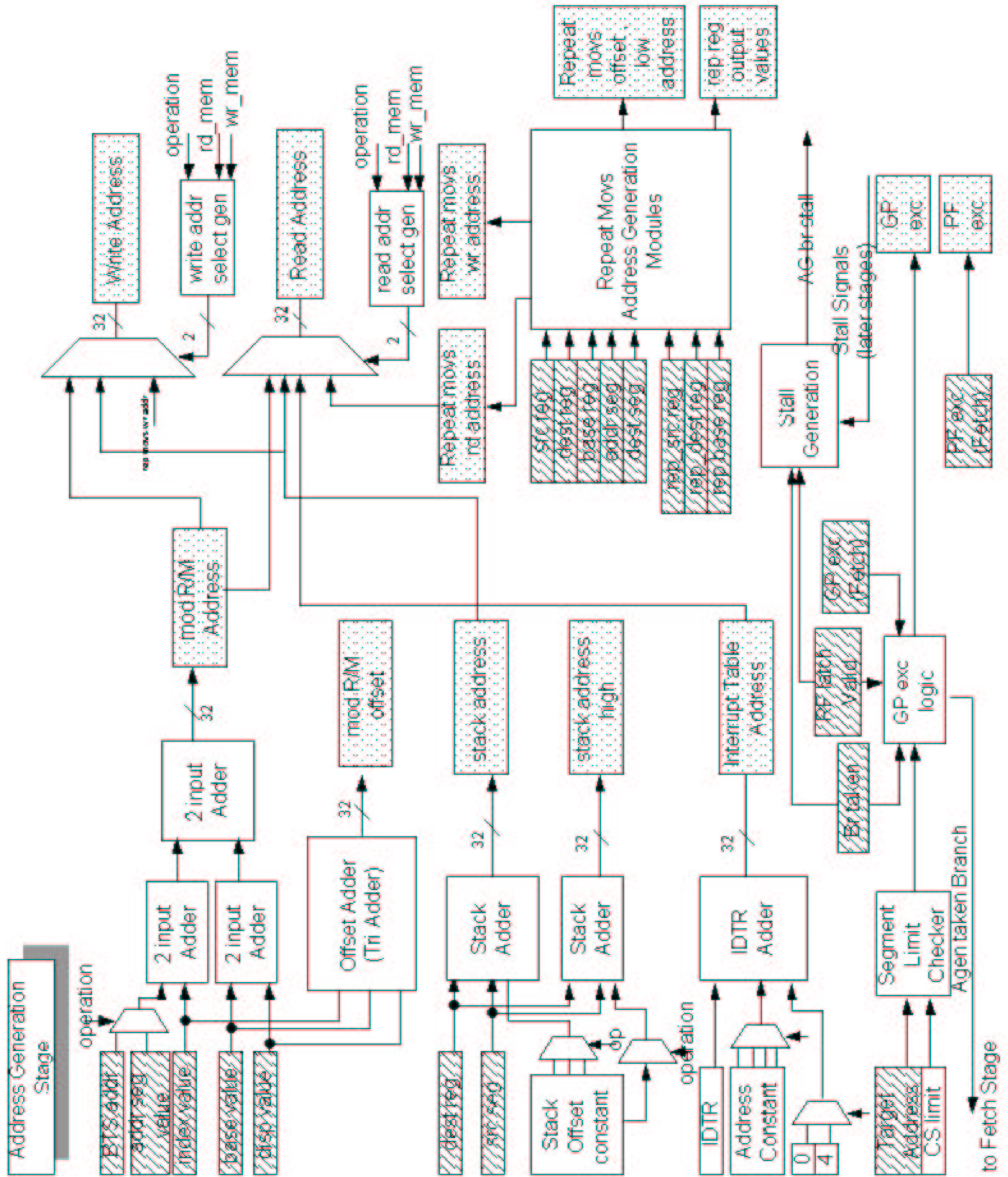

Figure 6: Address Generation Stage Block Diagram

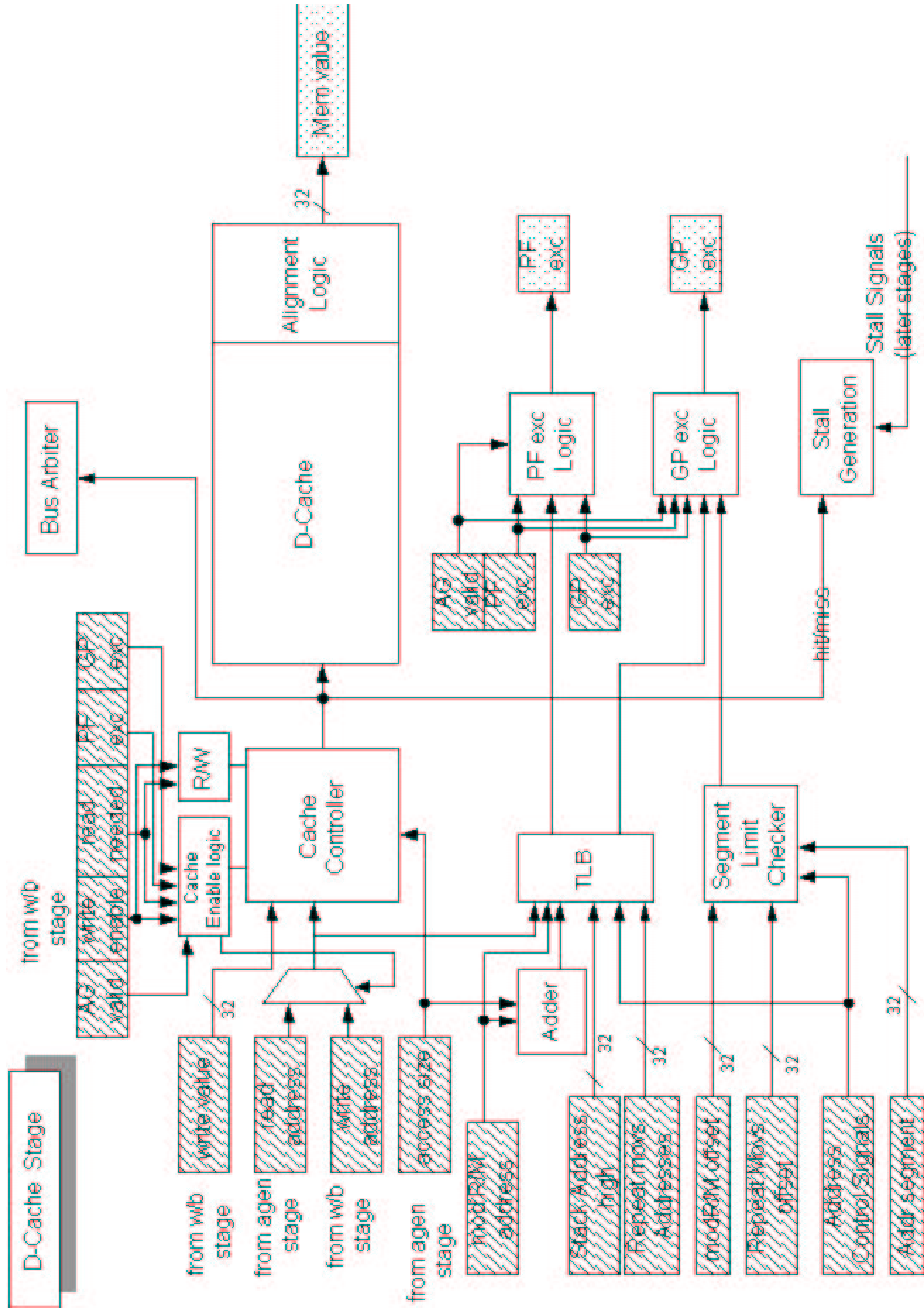

Figure 7: D-cache Access Stage Block Diagram

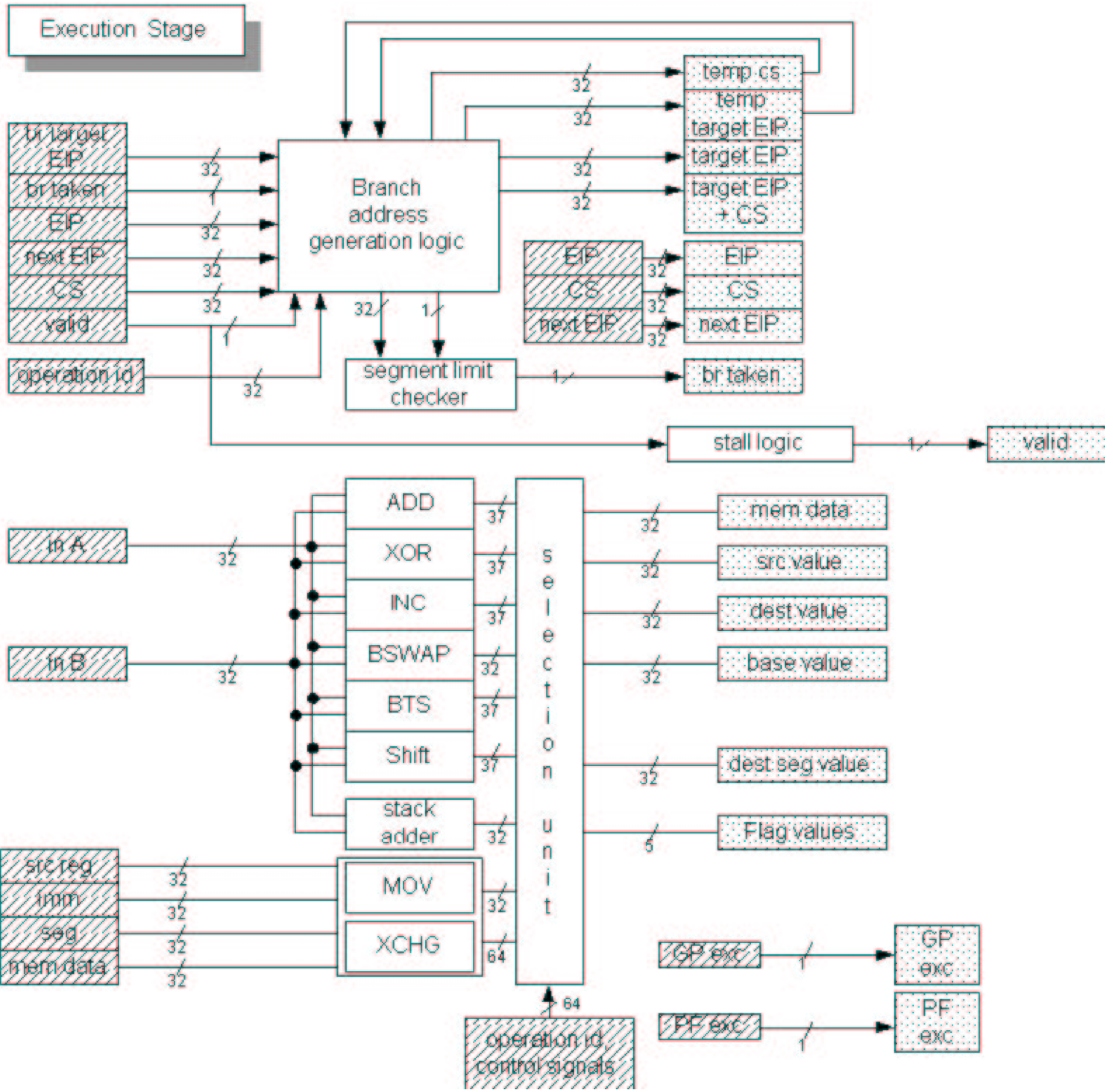

Figure 8: Execution Stage Block Diagram

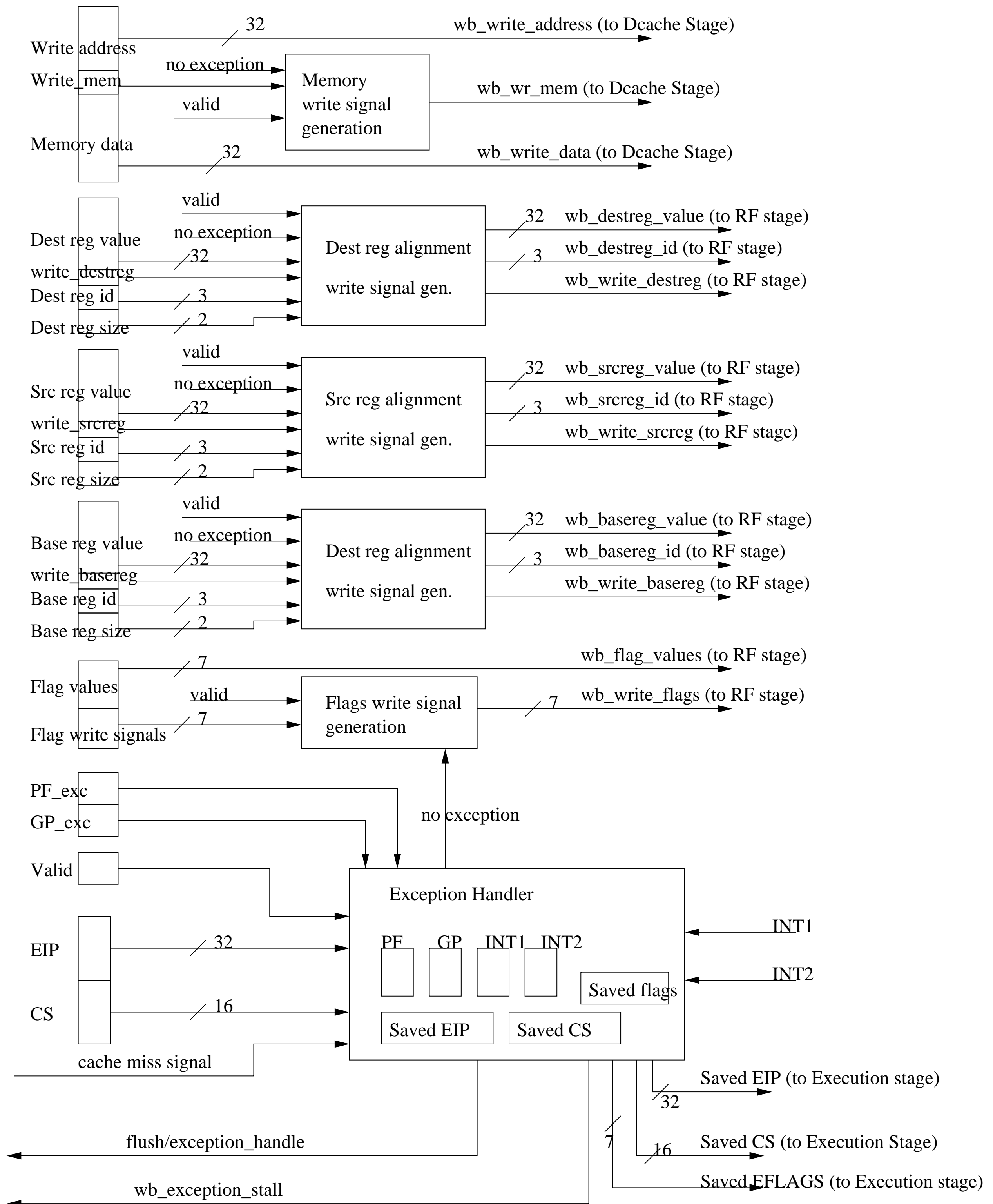

Figure 9: Writeback Stage Block Diagram

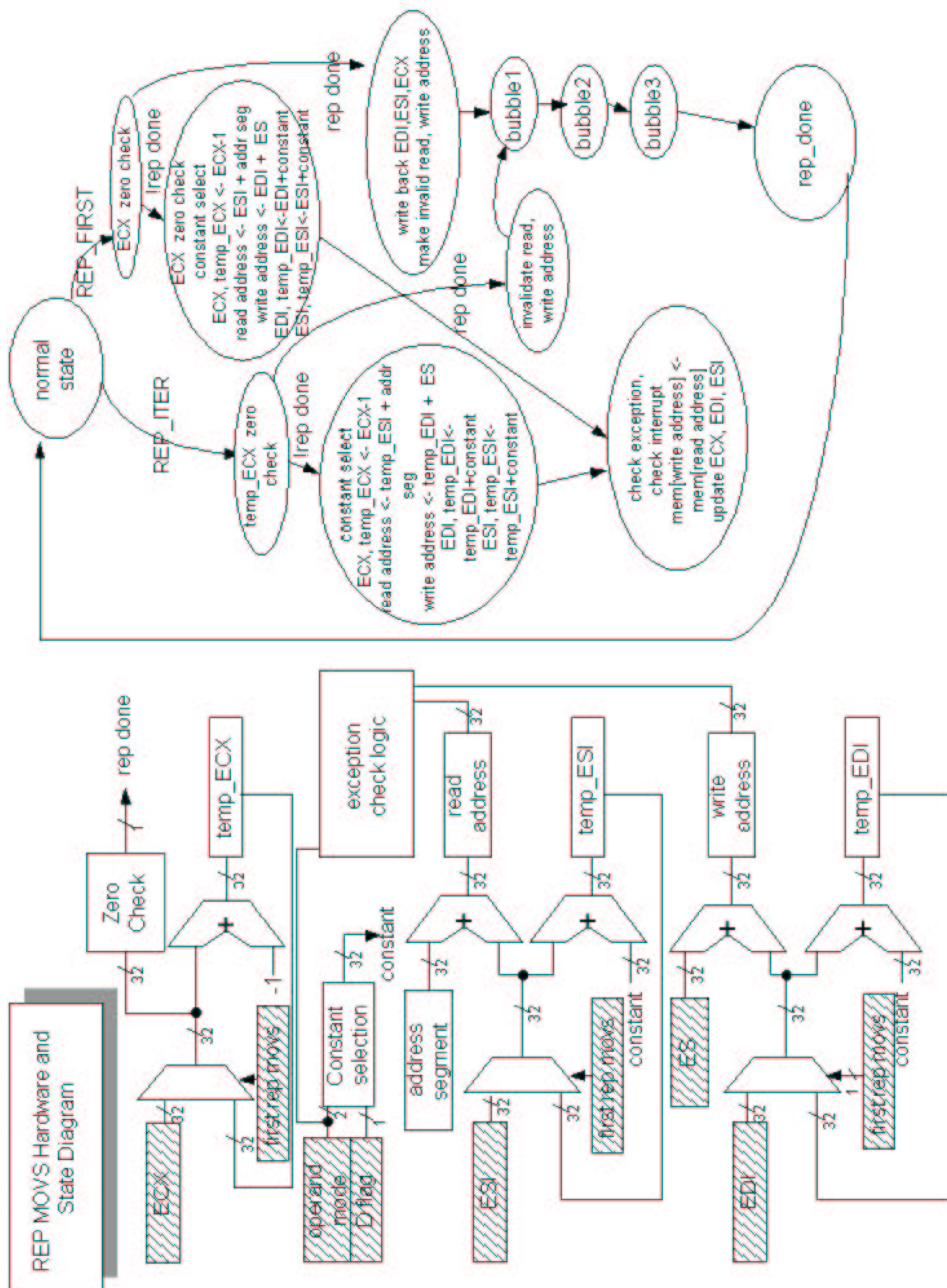

Figure 10: Special Hardware and State Diagram for REP MOVS

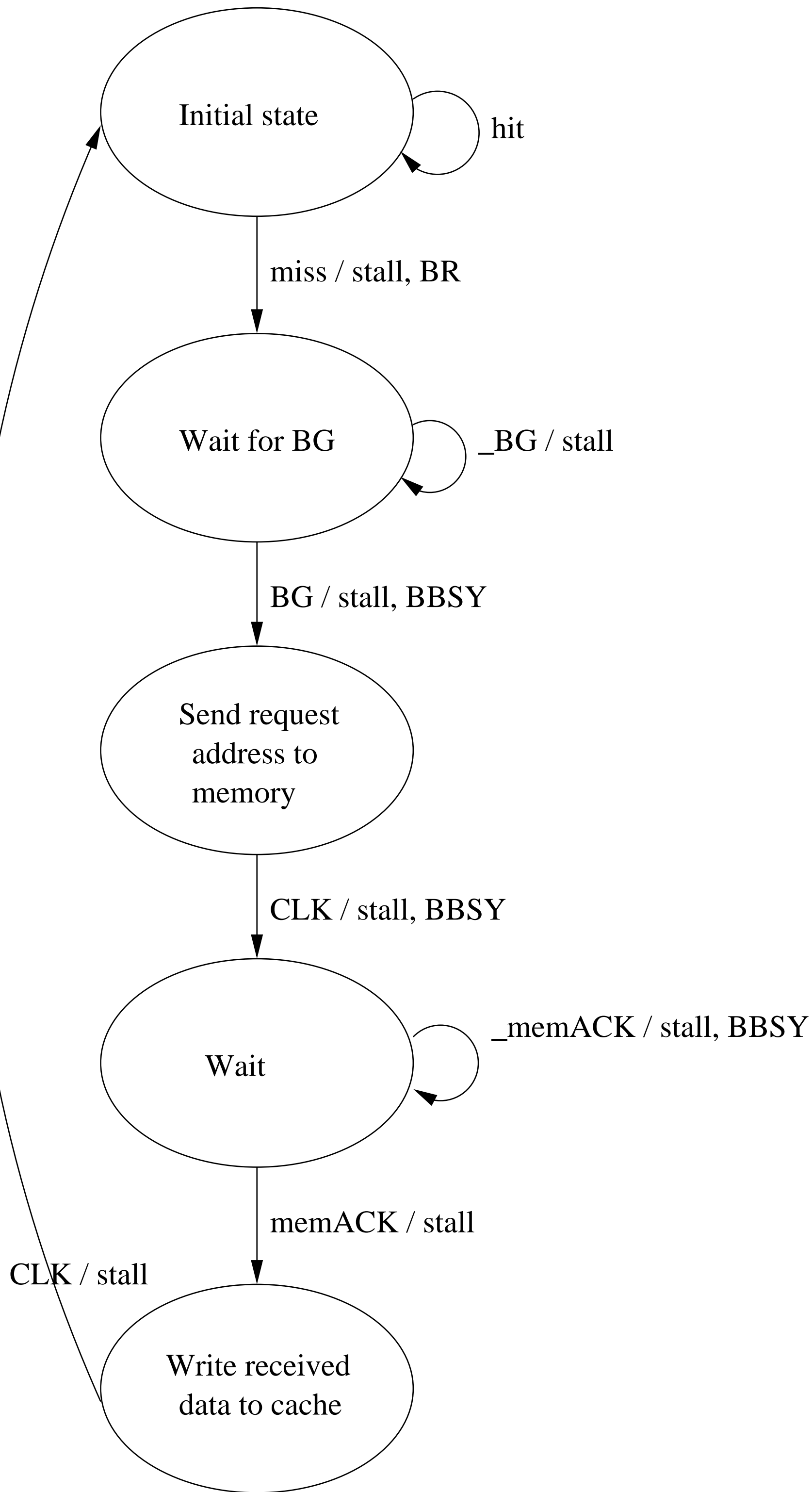

Figure 11: I-cache Controller State Diagram

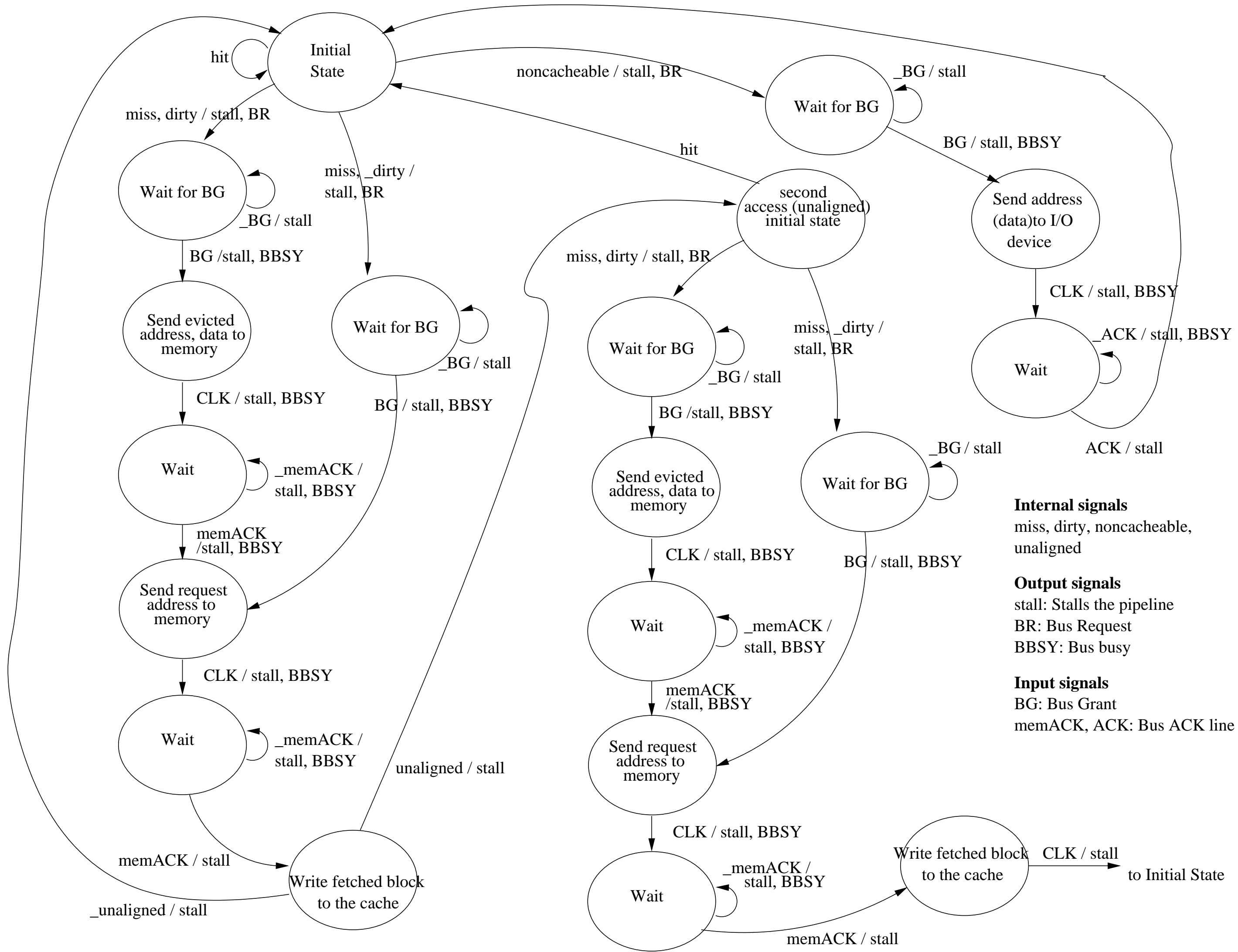

Figure 12: D-cache Controller State Diagram

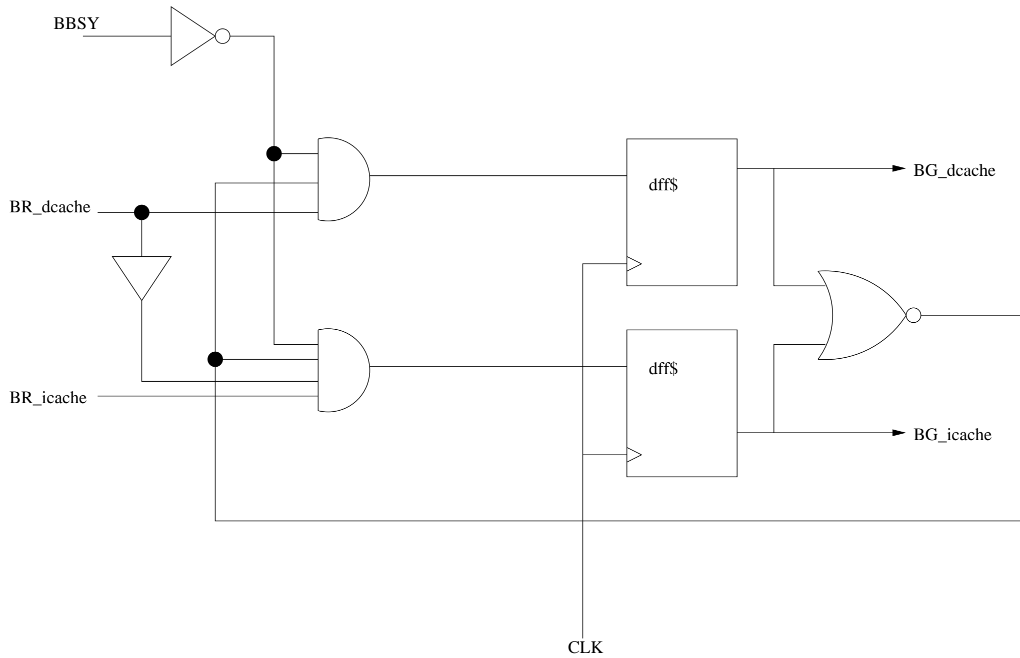

Figure 13: Bus Arbitration Logic

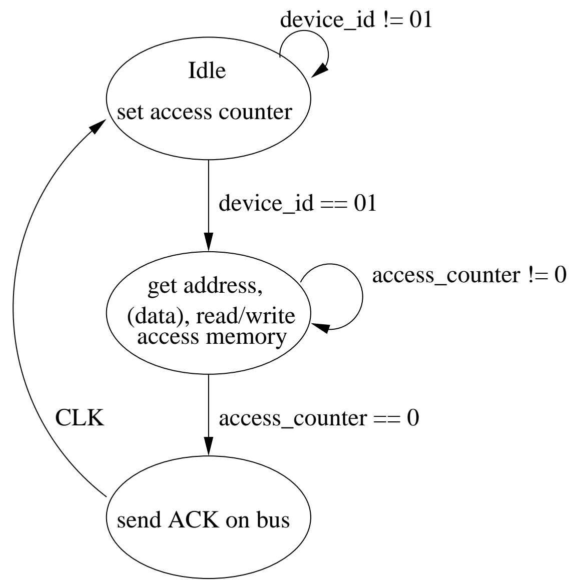

Figure 14: Memory Controller State Diagram

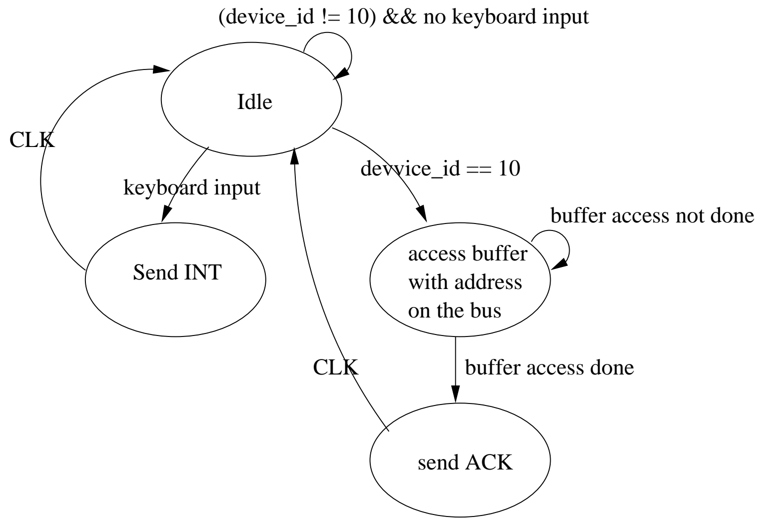

Figure 15: Keyboard Controller State Diagram

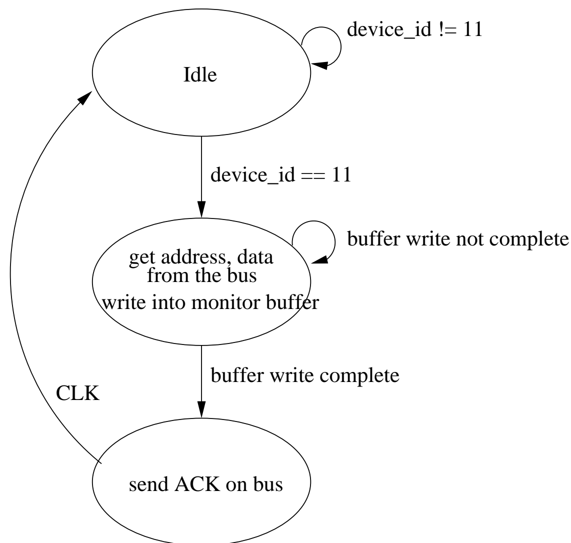

Figure 16: Monitor Controller State Diagram