Microarchitecture Final Report
EE 382N-19
5/13/16

Esha Choukse
Nicholas Kelly
Kishore Punniyamurthy

1 Introduction

Our major design objective was to create a somewhat realistic, in-order, x86 processor. Since it was in-order, maximizing MLP and leveraging locality (to improve latency) was of utmost importance. Thus we put in features such as: multiple MSHRs, a Store and Eviction Buffer, Critical-word first, and Memory Banks with Row buffers. Additionally, in order to ensure instruction throughput, we added a branch predictor, instruction queue, a banked register file, and data forwarding (with scoreboarding).

2 System Overview

In general, our overall system is similar to conventional computer systems (see Figure 1). We have the main processor, which contains the pipeline stages, as well as caches and memory controller. The CPU is connected to memory and other I/O devices through a 32-bit data bus. Data from these devices is accessed through the memory controller within the CPU based upon the TLB memory mapping (i.e. Memory-mapped I/O).

3 Pipeline Overview

Our final pipeline (see Figure 2) consists of eight stages:

- **Fetch (IF):** Instruction fetching/buffering as well as branch prediction
- **Decode 1 (DE1):** Instruction length decoding and normalization
- **Decode 2 (DE2):** Instruction decode, pipeline control, operand formatting
- **Register Fetch (RF):** Registers, data forwarding, preprocessing
- **Address generation (AG):** Implements different addressing modes
- **Load (LD):** TLB, loads/stores, store buffer
- **Execute (EX):** Instruction execution functionality
- **Writeback (WB):** Register/memory writeback

Inputs/outputs are passed synchronously through registers with valid signals for each stage. However, there are direct connections between stages (asynchronous) for writeback, data forwarding and other functionality. While we initially were planning on a cycle time lower than 10ns, we ended up having a cycle time of 14ns.This was primarily due to a critical path (discussed later) between LD and DE1 (or other stages).

4 Stages

4.1 Fetch Stage

The Fetch stage consisted of the instruction fetch mechanism (interface to the I-Cache), branch prediction, and misprediction/jump handling.

4.1.1 Instruction Fetch

IF (Instruction Fetch, see Figure 3) is responsible for fetching the data for each instruction in the program. It accesses this data through the I-Cache in a simplified (as compared to the D-Cache) interface which essentially only supports cacheline-aligned accesses (16-bytes). Additionally, IF is decoupled from DE1 through two FIFOs: the Instruction Queue and EIP Queue. The *Instruction Queue* buffers the data from the I-Cache, allowing instructions essentially prefetched based upon speculation and be more resistant to I-Cache misses. However, the usage of the Instruction Queue depends on the latency from memory, the average instruction length, and the amount of mispredicted branches (flushes) in the program. The queue consists of 4 entries, each of which contains a 16-byte instruction data portion, 1-bit indicating a *Flow Change*, and 1-bit indicating an exception (ECODE).

The *EIP Queue* buffers decisions made from the branch predictor. Whenever there is a change in instruction flow apart from "Not-taken" (i.e. *Flow Change*), the predicted (or mispredicted) EIP is pushed onto the queue. The queue consists of 4 entries, each of which contains a 32-bit EIP address and a 5-bit branch location (within two cachelines). Primarily, the Instruction Queue allows for the IF stage to buffer instruction data regardless of whether stages are stalling the pipeline (which can happen often often due to a cache-miss or store-dependent load). Similarly, the EIP Queue allows for branch speculation to make nested predictions. However, since our pipeline is only 8-stages, this will likely only be utilized for small (distance),long (iterations) loops. All accesses (other than the first unaligned address in a jump) made to the I-Cache are cacheline-aligned. Upon an unaligned jump, the first partial cacheline is fetched. Since the jump EIP is associated with this cacheline, the partial size can be calculated.

4.1.2 Misprediction

Upon a misprediction (as indicated from the EX stage), the entire front-end is invalidated. Additionally, since there may be a I-Cache access already in-flight (which cannot be canceled), it may have to wait till the I-Cache is available for a request of a different address. Thus, IF tracks the status of the I-Cache so the misprediction request can be immediately fulfilled when the I-Cache is available.

4.1.3 Branch Prediction

In order to keep the Instruction Queue full, we utilize a two-level (GShare) branch predictor coupled with a BTB (Branch Target Buffer). This allows us to immediately redirect the EIP with no delay at all (in the ideal case). The branch predictor is updated with information from EX upon each branch. Similarly, for taken branches, the BTB is updated with the correct target EIP and branch location. Since all predictions are made at a cacheline granularity, the address used to update the tables is $BR_{Address} + BR_{Length} - 1$. This guarantees that the entire BR instruction will be fetched in the cacheline before fetching from the branch target. Additionally, complications arise when there are multiple branches within a cacheline. Since jump/call instructions can be very small ($\geq$ 2-bytes), this can likely happen with small loops or switch/case-like paradigms. In order to combat this, we use a partitioned 4-way BTB where each half of the cacheline gets 2-ways. Thus, depending on the address, the closer result (in terms of BR instruction) is used. For now, on a conflict within the grouping of 2-ways, the first way is used as the target. Similarly, the branch predictor should also support a partitioned scheme, but the change was not made since it was late in the testing/design phase. Currently, the branch predictor has a branch history register that tracks the last 8 branches which is XOR'd with the last 8 bits of the cacheline address to index into the tables and each entry in the table contains a two-bit counter.

Ideally, we would have liked to implement a Call/Return stack, a partitioned branch predictor (to select which prediction/branch within a cacheline to select), and a more efficient forwarding path (e.g. forward address from AG for unconditional branches).

4.2 Decode 1 Stage

The Decode 1 (DE1, see Figure 4) stage is responsible for instruction normalization and length decoding. Length decoding is essential for shifting the instruction buffer (so it always points to a valid instruction)

and also for determining the address of the next instruction (used to determine mispredictions).

4.2.1 Instruction Normalization

The *Instruction Buffer* is populated from the IF stage through either a bypass path or the Instruction Queue. The bypass path is activated if DE1 is invalid (during startup or branch) or the buffer is currently empty. Otherwise, bytes from the top of the Instruction Queue are shifted in. These bytes are shifted in parallel with prefix/opcode length decoding, allowing all potential 16-bytes to be available at decode. The bytes from the queue are first shifted in based upon the initial buffer size (before length decoding). After length decoding, the "filled" (instruction with queue bytes) is shifted by the instruction length which becomes the next value of the instruction buffer. If the initial size of the instruction buffer is $\leq$ 16-bytes, then the top entry is popped of the Instruction Queue.

4.2.2 Length Decoding

Length decoding first consists of an initial step to find prefixes (maximum of two) and a second opcode (0x0F). Only the operand-Size prefix is decoded in this stage, since it is necessary for length decoding. After the location of the primary opcode is found, the prefixes/opcode are shifted out of the buffer and the primary opcode and potential MODRM are fed to the *Length Decoder*. The Length Decoder consists of generated *espresso* logic (two-level logic) from a table. The generated logic uses optimized NAND gates (alternating inverting levels) and is completely buffered. This logic produces lengths of each section in the instruction including: MODRM, SIB, IMM, and DISP. The overall length is calculated by a carry-save adder which adds the lengths of each section in parallel. Utilizing the CSA simplifies the Length Decoder, otherwise the logic doubles in complexity.

It is important to note that DE1 was designed with respect to minimizing logic. If logic is unrestrained, a separate decoder could be used starting on each of the first four bytes of the instruction. The correct value would be selected with a MUX at the end, allowing for completely parallel calculation of length (and the possibility of decoding multiple instructions). However, since length decoding was not on the critical path, it was unnecessary.

4.2.3 Branching

As described earlier, a *Flow Change* bit is set for each entry in the Instruction Queue if there is a change in flow (i.e. taken branch). Thus, when the bit at the top of the queue (or local DE1 bit) is set, it indicates there is a BR instruction within the buffer. The branch address field in the EIP Queue specifies the location of the branch within the instruction buffer. Upon reaching the branch location, the EIP from the EIP Queue is popped off as well as clearing the Instruction Buffer and filling it with a popped entry from the Instruction Queue. The *Next EIP* for the instruction is set to the target (popped EIP) which allows for a misprediction to be detected in the EX stage. Otherwise, if there is no branch in the buffer, Next EIP is simply the current EIP plus the instruction length.

4.3 Decode 2 Stage

The Decode 2 (DE2, see Figure 5) stage is responsible for Instruction decoding, operand extraction, and pipeline control (e.g. interrupts/exceptions). There are two flavors of instructions, those which are a single μop and those which are multiple (i.e. use $\mu code$). The only instructions that use $\mu code$ are those which perform multiple instructions, such as CALL, RET, and IRET. Additionally, the Pipeline Controller performs special operations for emitting μops as well as controlling/flushing portions of the pipeline.

4.3.1 μop Decoder

The μop Decoder is for 1 μop instructions coming from DE1. It decodes the control signals based upon the opcode, MODRM, and operand-size. Similar to the Length Decoder, it utilizes *espresso* logic to generate

the control signals for the instruction. Additionally, for *μcode*-based instructions, it emits an address to be used for indexing into the *μcode* ROM.

4.3.2 Pipeline Controller

The *Pipeline Controller* is responsible for handling REPNE, Exceptions/Interrupts, Mispredictions, and HLT. It is implemented as a Mealy-FSM with 4-states. In the case of normal instructions (non-*μcode*) it performs no function; although, for *μcode* operations, it directs the *μop* Sequencer to execute from *μcode*. For Interrupts, it performs the following actions:

1. Flush the entire front-end (IF/DE1)
2. Inject bubbles until the last instruction is at WB
3. Save the Next EIP value of the last instruction (for pushing onto stack) and begin emitting the *μcode* interrupt initialization routine
4. Upon the last *μop*, invalidate the DE2 stage

The last *μop* is a jump to the location specified by the entry in the IDT which effectively activates the IF stage and begins fetching instructions from the interrupt handler. Alternatively, Exceptions are triggered from the WB stage when an *exception bit* bit is set. The response to an exception is similar to an interrupt except the entire pipeline is flushed and the EIP of the instruction in WB is saved (for pushing onto the stack).

Mispredictions are triggered by the EX stage, where the pipeline controller simply flushes the entire pipeline. In addition, if an I-Cache request is in-flight the Pipeline Controller has to buffer the target address until after the response returns. REPNE is controlled by the Pipeline Controller where DE2 first requests a temporary value of ECX which it locally decrements. A SCAS μOP is emitted for each iteration, and the ECX and Z-flag value is checked for the end condition. Upon an end condition or an exception/interrupt a MOV *μop* is emitted to save the ECX value to the RF. This saves the temporary state allowing for the operation to be resumed after the interrupt/exception.

4.3.3 *μop* Sequencer

The *μop Sequencer* controls which *μop* is emitted to the RF stage. It receives direction from the Instruction Decoder and Pipeline Controller on which *μop* to fetch from the *μcode* ROM as well as which operands to override.

4.3.4 Operand Extraction

Operands (MODRM, SIB, Immediate, Displacement) are extracted from the instruction through shifting by their respective lengths. Additionally, the Control signals emitted by the *μop* Sequencer are used to format the operands for the RF stage. This includes normalizing the IMM/DISP bytes, selecting the correct registers and valid signals for each SRx field and performing overrides for *μcode* operators or other special-cases. Some of these special cases include: using SS as the segment when ESP/EBP are specified and using ES for string operations. Similarly, the interrupt/exception initialization *μcode* emits a special *μop* which utilizes the SIB addressing to perform a jump to the value at the $IDTR + ECODE \times 8$ address.

4.4 Register Fetch Stage

In this stage (see Figure 6), all the inputs required to complete the instruction (except memory operands) are accessed. Major components of this stage include:

- Register File Block
- Scoreboard
- Predicate unit

- Data Forwarding
- Shadow Registers

4.4.1 Register File Block

The Register File block (see Figure 11) is a combination of 4 register files - General purpose registers, MMX registers, Segment registers and EFLAG register. 5 bits are used to denote registers, the first 8 encoding (2 most significant bits : 00) are used for general purpose registers, the next 8 are for MMX registers. The next 8 encoding (2 most significant bits : 10) are for Segment registers. The last 8 encoding (2 most significant bits : 11) are for used for denoting RegH (higher 8-bit) variants (e.g. AH, BH, CH) of general purpose registers. The 32-bit, 16-bit, and lower 8-bit variants of registers have the same encoding and are differentiated using the data-size. This makes it easier for the *Scoreboard* to track dependencies across them. The higher 8-bit variants (e.g. AH, BH) have a different encoding since accessing and writing them back involves shifting. However, the Scoreboard treats them same as other variants of registers (i.e EAX, AX, AL, and AH will set the same Scoreboard bit). The 2 most significant bits are used to decide which register file is to be accessed or written.

The general purpose register file has 2 banks to support SIB addressing mode (requires upto 3 registers), while using less area for register file. The banks are selected based on the least significant bit of the register encoding. In our implementation of Repeat SCAS , the decode temporarily holds and modifies the ECX value while generating the repeated SCAS instructions. The ECX register is read by decode even before the repeat SCAS reaches the RF stage. This special request by decode is supported by overriding the register source values in RF stage provided there is a port available.
EFLAG register is accessed and written depending on dedicated control signals.

4.4.2 Scoreboard

Our design has a 24-bit scoreboard (one bit for each register, except for AH, BH, CH, and DH). The scoreboard bit of the destination register is set when the instruction leaves RF stage and is cleared at write back. The Write-After-Write (WAW) condition will cause a problem since the scoreboard bit will be reset by the initial write (even though a second write is in pipeline). For all registers except EFLAG, this condition is resolved by creating a false dependency, ensuring that 2 writes to same register are never in the pipeline simultaneously. This does not affect the performance significantly since in most instructions (except MOV, POP) the destination register is an input (2 address machine) as well.

However, this is not true for the EFLAG register. Creating false dependency between instructions which will modify EFLAG register will severely affect the performance. Our design has a *Counting scoreboard* for the EFLAG register. It has a 3-bit counter which increments every time an instruction which modifies EFLAG crosses RF stage. The counter is decremented on every EFLAG writeback. Any instruction which wants to read EFLAG will stall at RF stage till the counter value is 0.

4.4.3 Predicate unit

This unit resolves the conditional instructions (CMOVC, JNE, and JNBE) in RF stage itself. Depending on the evaluation of predicate, CMOVC is modified into regular MOV or NOP instruction. For conditional jumps, the appropriate offset (next instruction or target) is selected based on the predicate value.

4.4.4 Data Forwarding

We have implemented Data Forwarding in our design. The valid values are forwarded from EX stage after the output is computed. It was possible to compute and forward the data in execute stage since it was not the critical path and hence did not affect the cycle time. The destination register values in all pipeline latches beyond RF are compared with the register sources in RF stage. It is a daisy chain configuration, so if a match occurs, the matched data from that stage is forwarded, otherwise the data from the next stage is

forwarded. The daisy chain ensures only the earliest match in the pipeline will reach RF. For example, if a match happens both in AG and EX stage, no valid forwarding will reach RF and the instruction in RF will continue to stall.

4.4.5 Shadow Registers

Every register file entry has a corresponding shadow register, which can be use to take snapshot of the register file. Our design implements a few instructions as *μcoded* instructions (e.g. CALL, RET). Shadow registers are used to rollback the register file if any exception/interrupt occurs during the execution of a *μcoded* instruction. A snapshot of the register file is taken at the beginning of *μcoded* instruction. The values are restored if any interrupt/exception occurs during the execution of *μcoded* instruction.

4.5 Address Generation Stage

We have a separate stage for address generation and segment limit check, since the multiple levels of adders were a potential critical path. Figure 7 shows the address generation logic. Kogge-stone adders were used to make the computation faster.

4.6 Load Stage

We tried to optimize our system for memory, since in an in-order pipeline, cache-misses and memory accesses account for many stalled cycles. The following optimizations were done in the memory system in order to achieve better performance:

- **4-way cache:** The D-cache is a 4-way, 8 set cache with a Tree-LRU replacement policy and 16B cachelines.
- **Store Buffer:** We have a Store Buffer with 4 entries that we use to allow any loads to go ahead before stores, unless there is a conflict in address, or the store buffer is full. Hence, in the general case, stores only occur in the cycles where the instruction in the load stage does not need a memory load access.
- **Critical Word First:** Since the data bus to the memory is only 32 bits, a 16B cache line takes 4 cycles to be transferred. In order to avoid the needless stall, we have the memory wrap around the data in a cache line starting at the exact load address. The load stage is unstalled as soon as the first transfer of 32 bits completes (or 2 transfers for a 64-bit memory access).
- **Row Buffer:** Each bank in the memory has a 64B row buffer. This is done by always driving the row address and chip enable to all the banks. By keeping the information on which row is open per bank, we can get a row hits, which bring down the memory look-up time for a load from 60ns to 0ns (8 cycles to 4 cycles to read complete cache line data in our system). Hence, row buffer hits directly translate into 50% better access time.
- **Bank-level parallelism:** We have 16 banks in order to be able to process multiple requests in parallel, and have different rows open in different banks.
- **Multiple Memory Requests in transit:** We have 4 MSHRs to support up to 1 I-cache miss, 1 D-cache miss, and 2 eviction requests being processed in parallel. This, in combination with the bank-level parallelism and row hits, translates into a busy bus where no bus cycles are lost if there are requests available with no bank conflict.
- **Eviction Buffer:** We have an eviction buffer of 2 entries, which on a hit leads to no stalls in the load stage.

The MSHRs and memory bank optimization is explained in more detail in later sections. Here, we explain the load stage alone, meaning, the interaction between loads/stores of the instruction with the D-cache and store buffer.

Figure 8 shows the load stage. Note that if an instruction leads to a page fault or protection exception, the load or store associated with it never starts. Also, once an instruction leads to an exception, all following instructions received in the load stage will be ignored until the decode_flush signal is received. This is to avoid unnecessary load operations (keeping the cache clean) and to avoid committing the wrong store operations.

Upon a load: If the instruction in the load stage needs a memory load access, it goes through the following steps:

1. The Unaligned Check Logic block checks if the access crosses a cacheline or page boundary. If a page boundary is crossed, it leads to a load stage stall, since the TLB needs to be accessed twice. For a cacheline boundary crossing, we have a stall as well, since it needs two D-cache accesses (assuming a cached access). However, for an uncached access, only one access to the memory is required.

2. The physical load address is matched across all the valid entries in the store buffer. If there is a match in the range of 1 cache line, that store is scheduled to the D-cache when ready and the load stage is stalled until the store is committed.

3. Once the load is scheduled to the D-cache, upon a hit, we have the data. Upon a miss, the eviction buffer is looked up. If there is a hit in the eviction buffer, the data is returned from there and the eviction entry is written to the cache.

4. If the load led to a D-cache miss and an eviction buffer miss, it is written to the MSHR. Since memory system is implemented *Critical-Word-First*, as soon as the required data is available on the bus, the load stage is unstalled. The cache is filled in once the complete cache line has been transferred.

Upon a store: If the instruction in the load stage needs a memory write access, it goes through the following steps:

1. If the instruction does not have any exceptions related to it (page and protection faults are first checked for), a store buffer entry is allocated to it.

2. The store buffer is a circular queue to make sure that the stores happen in order with each other.

3. If the store buffer is full, it sends a signal to the D-cache scheduler asking to schedule the store before the load. The load stage is stalled until a store buffer entry is successfully reserved for the instruction.

4. The ID of the reserved store buffer entry is then passed along through the latches to the WB stage.

5. Upon receiving the WB value from the WB stage, the store buffer entry is marked ready.
   Instructions with multiple *μops*: If the store buffer entry belongs to a *μcoded* instruction, it can still not be committed. This is because if a different *μop* of the instruction led to an exception, it would lead to an inconsistent state. Hence, the store buffer keep track if the instruction belongs to a *μcoded* instruction. If so, the store buffer entry is not marked ready until the WB value is received, and a *commit signal* is received from the WB stage, with the terminal micro-op of that instruction. In case of an exception, a *flush signal* is received, leading to flushing of all the store buffer entries related to that *μcoded* instruction.

6. The store buffer always sends the next ready request to the D-cache scheduler, but is only scheduled if there is a match between the load address and store buffer, or the store buffer is full, or there is no load access waiting in a particular cycle.

4.7 Execute Stage

In this stage (see Figure 9), instructions are executed and their results are computed. This stage consists of an ALU, Misprediction logic, BTB update logic, Exception logic, and Destination logic.

4.7.1 ALU

The ALU consists of Kogge-Stone adders, a shifter, and other dedicated units to execute instructions like BTC, DAA, PINSRW, SCAS, and CMPXCHNG. The ALU uses the 4-bit ALU SEL control signal to perform the appropriate functionality. Depending on the instruction, some of the control signals are modified by ALU. For example, in CMPXCHNG, the destination can be memory or register depending on result of the execution.

4.7.2 Misprediction Logic

This unit detects if a branch misprediction occurred, which is performed by comparing the target EIP computed by the ALU and NEXT EIP. The NEXT EIP address contains the starting address of the next instruction fetched and decoded by front end. In case of a mismatch, a misprediction signal is sent to Decode along with the correct target address.

4.7.3 BTB update Logic

This logic provides information to update the BTB information. If a branch is taken, the target EIP is sent to IF stage.

4.7.4 Exception check logic

This unit checks if the EIP of the current instruction and target EIP for a jump instruction are within the segment limit. If the EIP is beyond the limit, an *exception bit* is set and ECODE is set to protection fault vector.

4.7.5 DST2 Logic

This logic selects between store buffer ID (for memory write back) and second destination register depending on the instruction. Instructions like "XCHG", "POP Reg" write back to 2 registers. In such cases, this logic selects the second destination register.

4.8 Write back Stage

This stage (see Figure 10) mainly consists of delay logic for multiple register writebacks, exception handling logic and write back enable logic.

4.8.1 WB Delay Logic

: This logic generates delay signal if the instruction requires 2 register write back. Since the register file has 1 write port, instructions which require 2 register write back (POP Reg, XCHG) are stalled for 1 cycle so that 1 register can be written back each cycle.

4.8.2 Exception handling logic

This logic generates signal to save and restore register file in the shadow register file. When a micro-coded instruction is being executed, this logic signals the Register file to save the register file when the first μop is in write back stage. If an exception occurs in between an micro-coded instruction, then this logic signals the register file to restore the values from the shadow register.

4.8.3 W_EN Logic and MEM_WB_EN Logic

This logic generates register and memory write back enables appropriately if they are valid and do not raise exception (exception bit is 0).

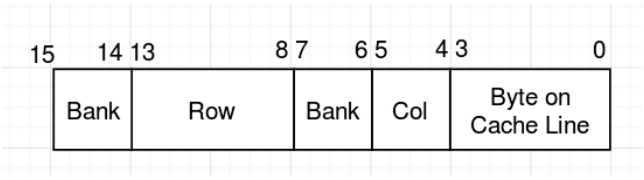


Figure 12: Physical address break-up

5 Memory

Let us now talk about the memory controller, MSHRs, BUS and main memory organization.

5.1 Banks

The memory has been divided into 16 banks in order to provide opportunity for greater parallelism. Figure 12 shows the physical address composition in terms of rows, columns and banks. Effectively, we have 64 columns (1B each) and 32 rows (64B each) in a bank. We divided the address as shown in order to achieve bank-level parallelism, both across pages and within a page.

5.2 Row Buffer

The chip-enables of all the memory chips are always enabled, and a valid row address is always provided. This is to take advantage of the row buffer.

Note that the address setup time for a read is 60ns and for a write is 25ns. By always supplying the last accessed row address to the SRAM cell, we make sure that if the same row is accessed again, there is no address-setup time needed. Effectively, this brings down the time for a cache line read from 8 cycles to 4 upon a row buffer hit (since it takes 4 cycles to send the cache line over the bus). For critical word first, it actually takes just one cycle to get the data (32 bits).

5.3 Memory controller and commands

We have an on-chip memory controller, which talks to the memory over the bus. We tried to come up with a set of commands that allows for the maximum level of parallelism. For this, we uncoupled the opening of a row (address setup) with an actual read or write operation. Also, the memory automatically resets its write enables after a preset time. So no "stop write enable" command is needed from the controller side.

Following commands can be sent by the memory controller to the memory over the bus:

- Open_Row: This command needs the controller to set the BUS_openrow, BUS_bank and send the row number as a part of the BUS_Data. After this command is sent, the corresponding bank should not be accessed for a read till 60ns and for a write till 25ns.

- Mem_Busgrant: This command is sent along with a bank number, column number and data-size. The command asks the memory to start sending the data starting at the given column number in the given bank, in the row that is currently open. Since the memory is critical word first, the address is not necessarily cacheline aligned. For a cacheable access, the data-size would be 16B, otherwise it would be 8, 32, or 64 bits based on the instruction. Once this operation starts, the bus cannot be granted to a different request until the data transfer completes, since the memory will be sending data over in up-to 4 consecutive cycles.

- Start_Write: This command is sent along with a bank number, column number, data-size and data. The bus is again locked up until the write request has sent over all its data (up-to 4 cycles). Note that the memory is smart to reset the write-enable signals internally. Hence, no stop_write command is needed. The data sent over is latched and supplied continuously to the bank until the write is complete.

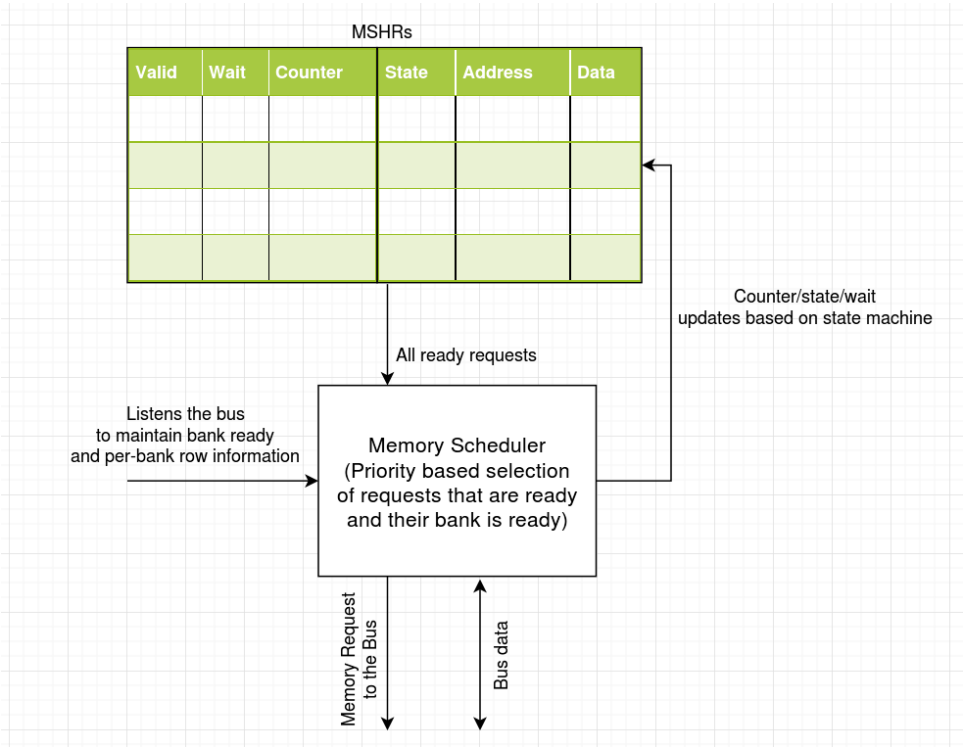


Figure 13: MSHRs and their update mechanism - the state only gets updated for the scheduled request

5.4 MSHRs

We have four MSHR entries which are tied to the type of request they can hold. There is 1 I-cache MSHR, 1 D-cache MSHR and 2 eviction WB MSHRs. Hence that is the type and number of memory requests that can happen in parallel. A request is called ready to schedule for the bus, if its valid bit is set and its Wait bit is low.

Upon having multiple requests ready, the scheduler chooses the request to schedule based on priority. Priority is set such that I-cache gets more importance than D-cache, which gets more importance than eviction requests. This is unless there is a back-pressure from the processor to get the eviction requests done.

A unit of the memory controller listens to the bus constantly to keep track of the row open in each bank and whether a bank is busy or not. This helps us determine if a request can be started and if it is a row buffer hit. Once a request starts, the bank related to it gets locked to it, till the request is complete. Note that this unit listens to the bus traffic, hence, it also tracks the memory requests from the I/O devices that potentially change the states of the banks.

5.5 State machine for memory requests

Following are the states a memory request can be in:

- 000: Not started
- 001: Address setup for read
- 010: Address setup for write
- 011: Start Reading in next cycle (Mem_busgrant)
- 100: Reading Data
- 101: Start writing this cycle
- 110: Writing data

- 111: Request complete

Each state has a counter related to it, which defines how many cycles the request will be in that state.There are also wait and DND bits for requests. The wait bit is set when a request is put into a state after a certain delay. If the wait bit is set, the request can not be scheduled to use the bus in that cycle. If the DND bit is set, there is guarantee that the same request will be scheduled to use the bus in the next cycle.

For example, when a read request does not lead to a row buffet hit, it starts in the state 1 and sends out an Open_Row command on the BUS. The state machine transitions it into state 3, but, with the wait bit set and the counter for the request set to a value so that the request is stalled for 60ns (Read address setup time). After the counter counts down to zero, the wait bit is set to zero and the request is now ready to be scheduled in state 3. In state 3, it sends out a Mem_busgrant command and goes into DND mode, where it listens the data bud for the next 4 cycles if a cached access was made. Figure 14 shows the complete state machine. Note that the state transitions are only done for a request that was scheduled to use the bus that cycle.

6 Bus and arbitration

The synchronous BUS consists of the following lines:

- BUS_clk
- BUS_reset_b
- BUS_DS: Data size of the request (2 bits)
- BUS_memgrant: Grants the memory permission to put the data on bus starting next cycle
- BUS_OpenRow: Commands the memory to open the associated row in the specified bank
- BUS_Bank: Bank number related to the current command
- BUS_Col: Column number related to the request
- BUS_Data: 32 bit data line
- BUS_Valid_Memctrl: The memory controller sets this bit whenever it is using the memory
- BUS_Valid_DMA: The DMA sets this bit when it is talking to the memory
- BUS_DMA_DND: The DMA sets this bit when it is in the middle of writing the data onto the bus (state 110). The memory controller cannot take over the bus when this bit is set.
- BUS_DMA_Command: The memory controller sets this bit when the program accesses a Memory mapped IO register for DMA.
- BUS_KB_Command: The memory controller sets this bit when the program accesses a Memory mapped IO register for keyboard.

Arbitration

To keep the arbitration simple and fast, we make memory controller the default bus driver. This means that the DMA can only drive the bus in the cycles when the BUS_Valid_Memctrl, BUS_DMA_Command and Bus_KB_Command are all zero. The memory controller can take over the bus whenever it wishes to, except, when the BUS_DMA_DND bit is set to 1. This is because we make sure that memory reads and writes for a single request happen in consecutive cycles. Hence, if the DMA is writing a cache line to the memory, the memory controller cannot use the bus for four cycles.

7 I/O

We have DMA implemented as a complex I/O and keyboard as simple I/O example in our system. The TLB has a DMAIO and KBIO bit per entry which tells us if an access is an MMIO access. Whenever one of these bits is set for a request, the memory controller sets the corresponding BUS_command bit high.

7.1 DMA

The DMA, much like the memory controller listens to the bus to keep track of the banks' busy status and open-row status, in fact we re-used the same module for DMA. The state machine per cache-line request of the DMA is similar to that of memory. As mentioned earlier, the default bus arbitration only allows the DMA to control the bus if the memory is not controlling it. In our system, since we support writes to memory at a granularity of 8 bits, unaligned DMA transfers are supported too.

So, once the DMA has all the data transferred into the buffer from the disk (750ns later), it starts trying to get the bus control to either open the required row in a bank (on a row buffer miss), or to starts writing the data onto the bus (in case of a row buffer hit). Once it has written all its data, it sets the DMA_Complete signal, which is a sideband signal from the DMA to the processor. This signal is connected to the decode stage of the pipeline, which enters the interrupt handler after flushing the instructions in the pipeline.

7.2 Keyboard

The simple I/O or, keyboard is implemented as a single register that is connected to the bus. When the processor sends a BUS_KB_Command, the KDR, or, the keyboard data register sends its 32 bit data onto the bus. This is a single cycle transaction and does not require any arbitration.

8 Testing Methodology

We performed testing at both the unit level and system level. For unit testing, specific Verilog testbenches were created to simulate the expected inputs into the stage/unit in order to verify the correctness of the stage individually. In order to verify the correctness of the system, we created random assembly sequences to exercise different execution patterns and addressing modes. These sequences could be generated using many different options such as value bounds, address stride, operand types, and more. We used this to fill in instructions in more directed tests as well as creating long (>700) instruction sequences to verify no hangs occurred. Additionally, we created directed tests such as:

- Address strides, page boundary
- I/O interrupt functionality (Keyboard, DMA)
- Exception functionality (front/back-end page-fault, protection fault, segment-limit fault)
- Control flow (CALL/JMP/Jcc, recursion)

These tested specific parts of the architecture or specific situations that could not be tested with the random instruction sequences.

9 Discussion

Overall, during the project, we found that coming up with a suitable testing environment was one of the biggest challenges.

One of our notable bugs that was caught during checkout was that we used a *Virtually addressed cache and store buffer*. This was because we happened to incorrectly assume that no two virtual pages would be mapped to the same physical page. This led to errors, since we were not able to identify the conflict between load address and store buffer entries based on virtual address. Hence, we changed it to a physically-addressed system, (both cache and store-buffer).

10 Conclusion

At the end of the project, we were able to reach our design goal, producing a fairly aggressive, in-order, x86 processor. We were able to include many memory features, such as MSHRs, a Store Buffer, and Banked Memory. Additionally, we were able to include an Instruction Queue, branch predictor, banked register file, and data forwarding.

However, we did not have time to completely balance the design. Our critical path was dependent on the Load stage and its feedback to the decode stage, forcing a higher than necessary cycle time for all other stages. This could be remedied by pipelining the Load stage or having faster (or pessimistic) stall logic. We also later realized that we could have moved the TLB access to the AG stage in order to make the Load stage do lesser work and balance the stages.

Even so, our processor performs very well in terms of cycle count, primarily due to the amount of MLP we utilize and less amount of time spent stalling.

FIGURE 1: SYSTEM DIAGRAM

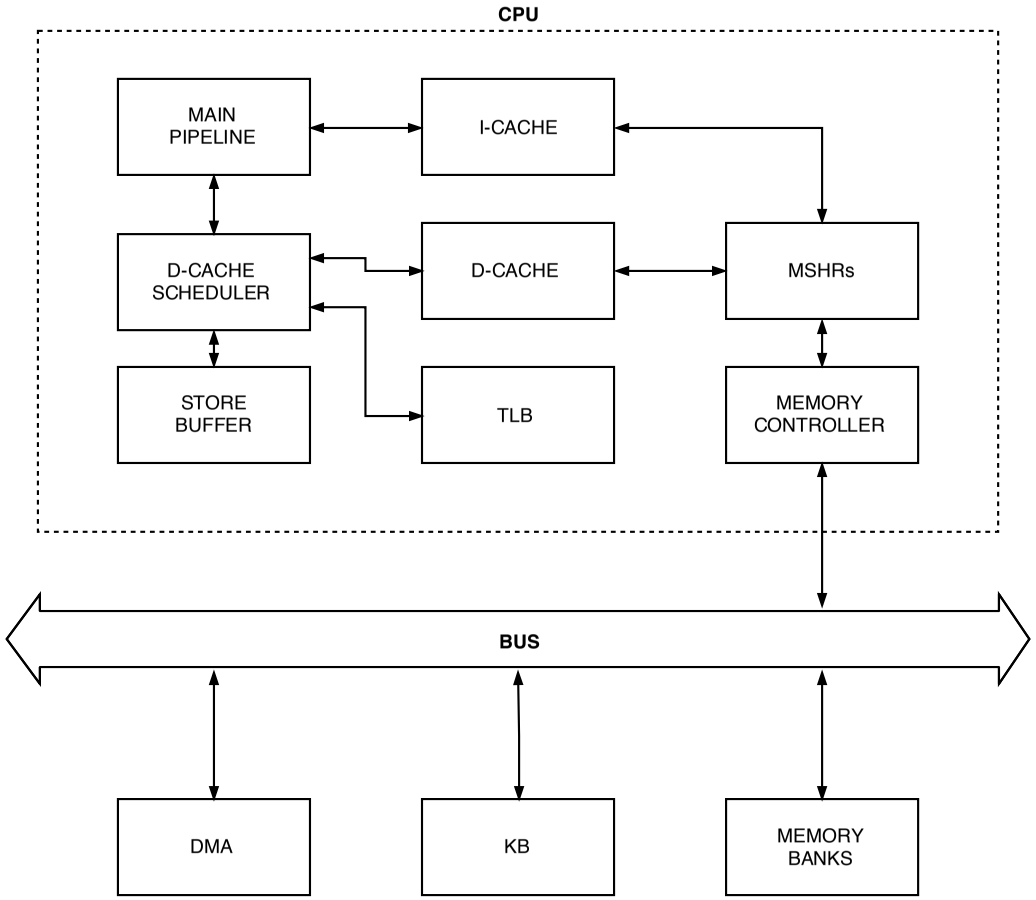


FIGURE 1: SYSTEM DIAGRAM

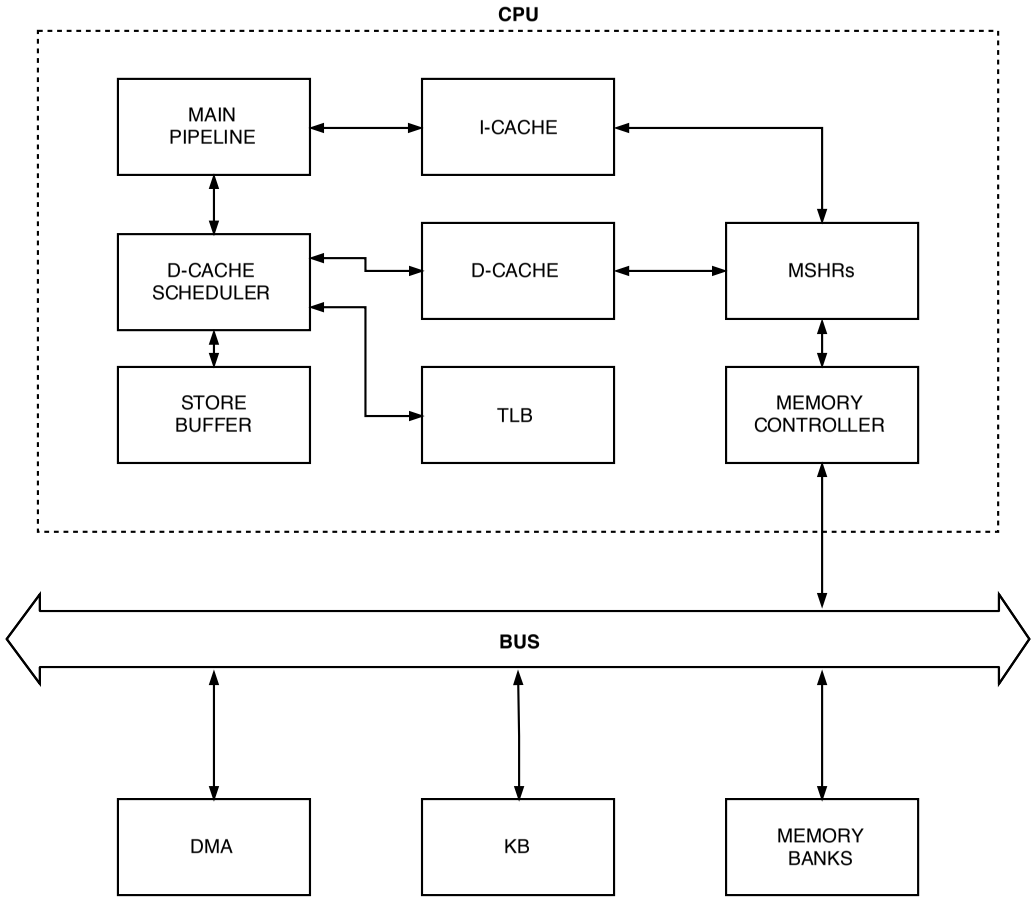


FIGURE 2: PIPELINE OVERVIEW

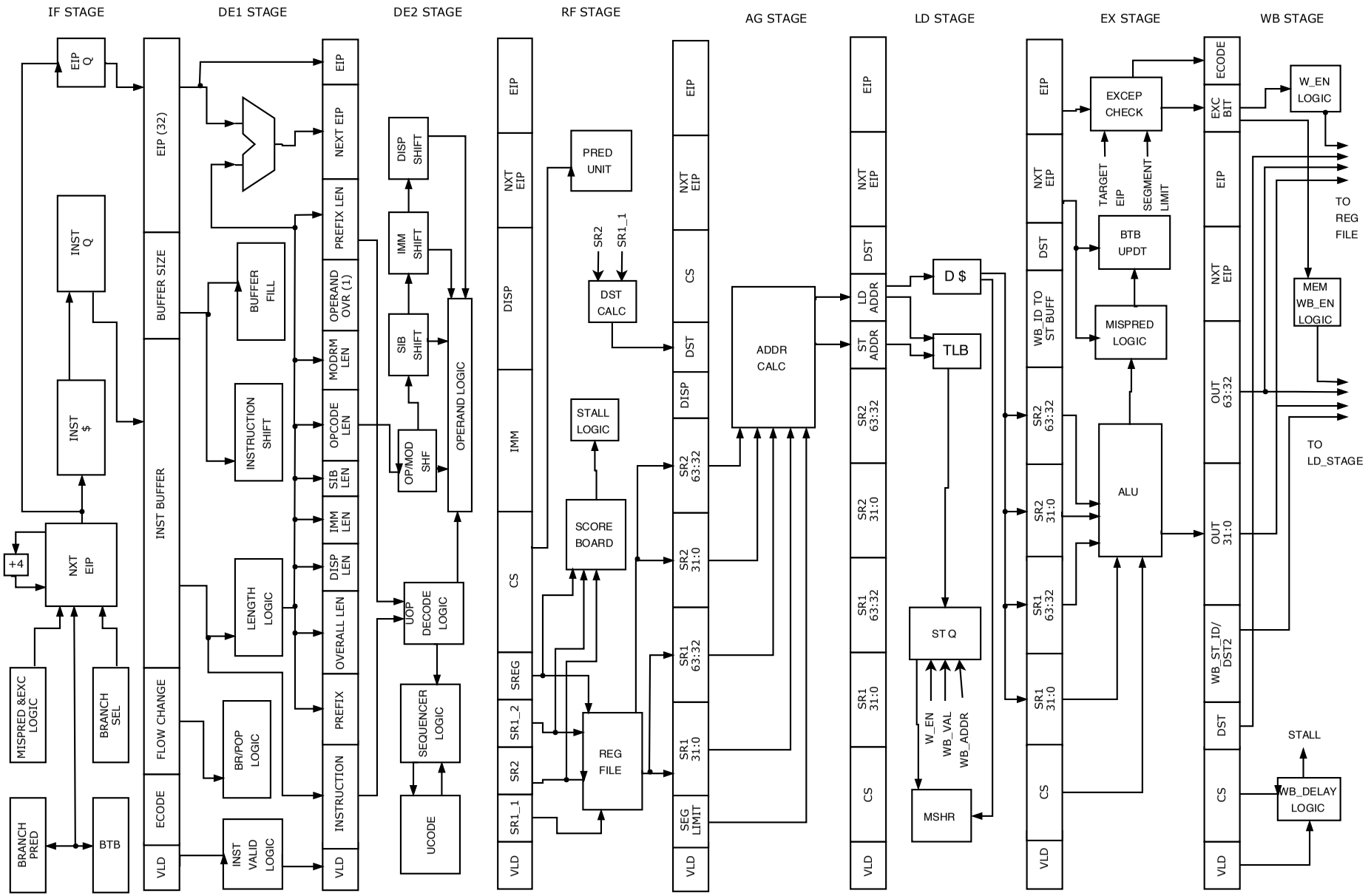


FIGURE 3: FETCH STAGE

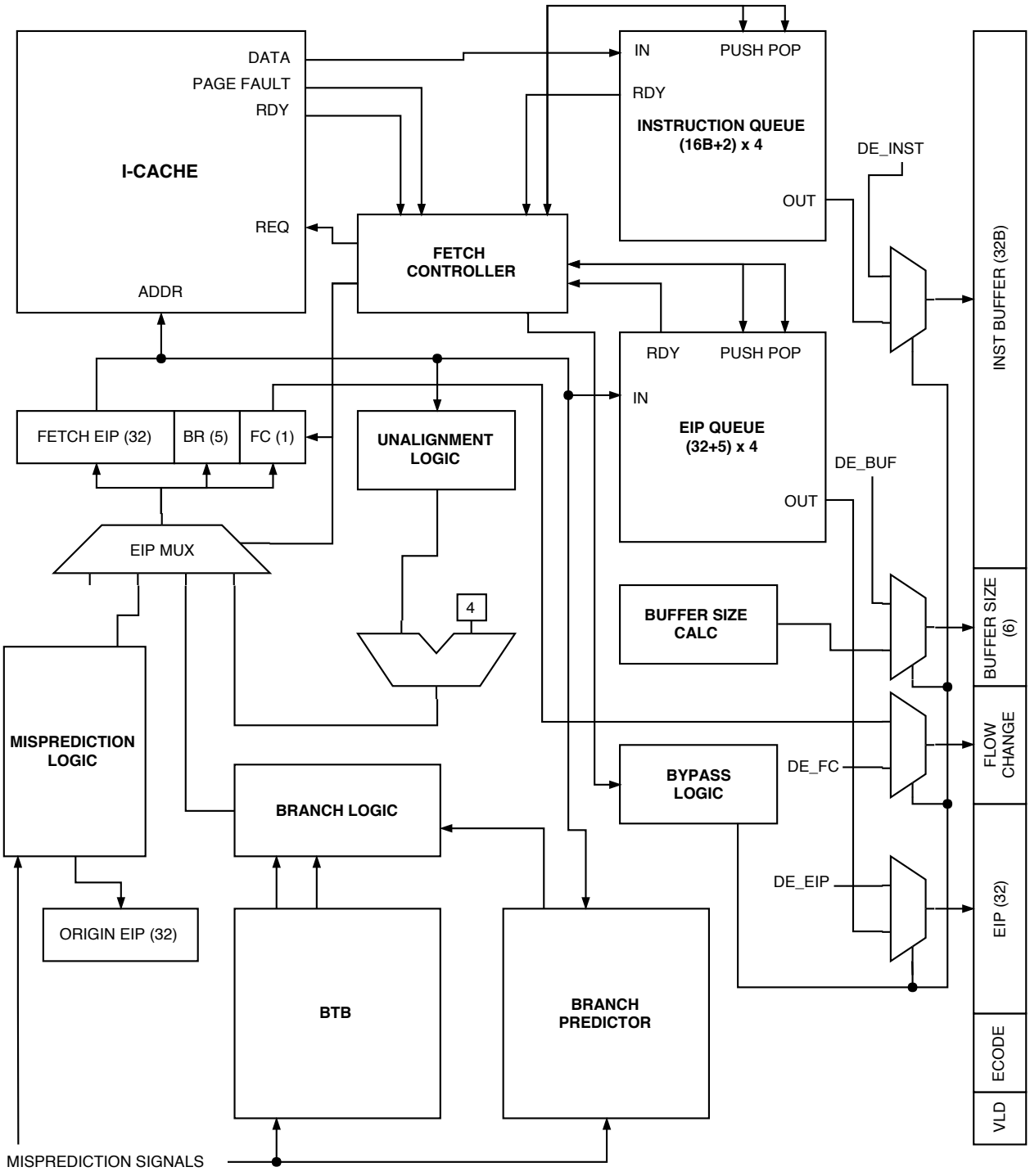


FIGURE 4: DECODE 1 STAGE

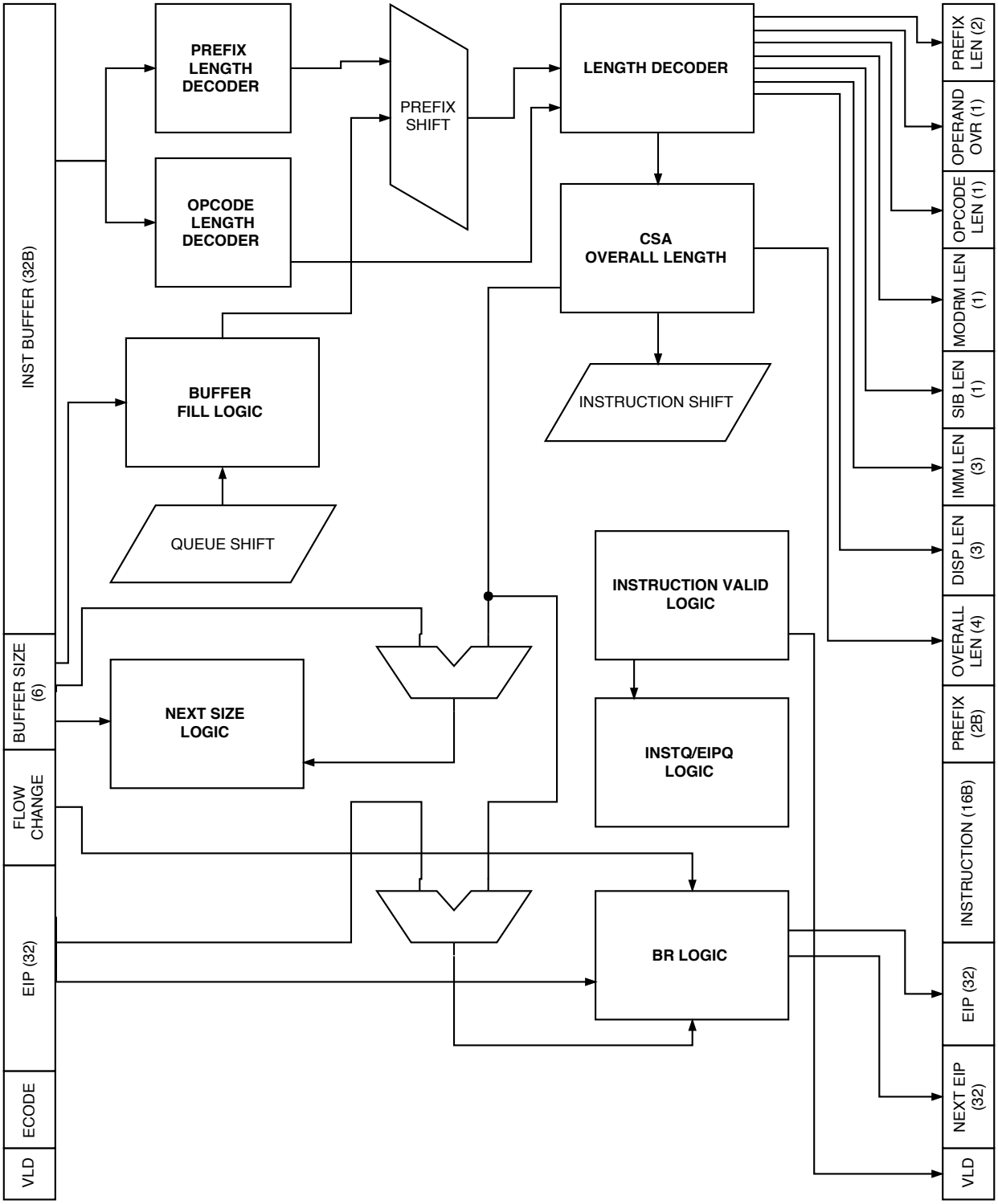


FIGURE 5: DECODE 2 STAGE

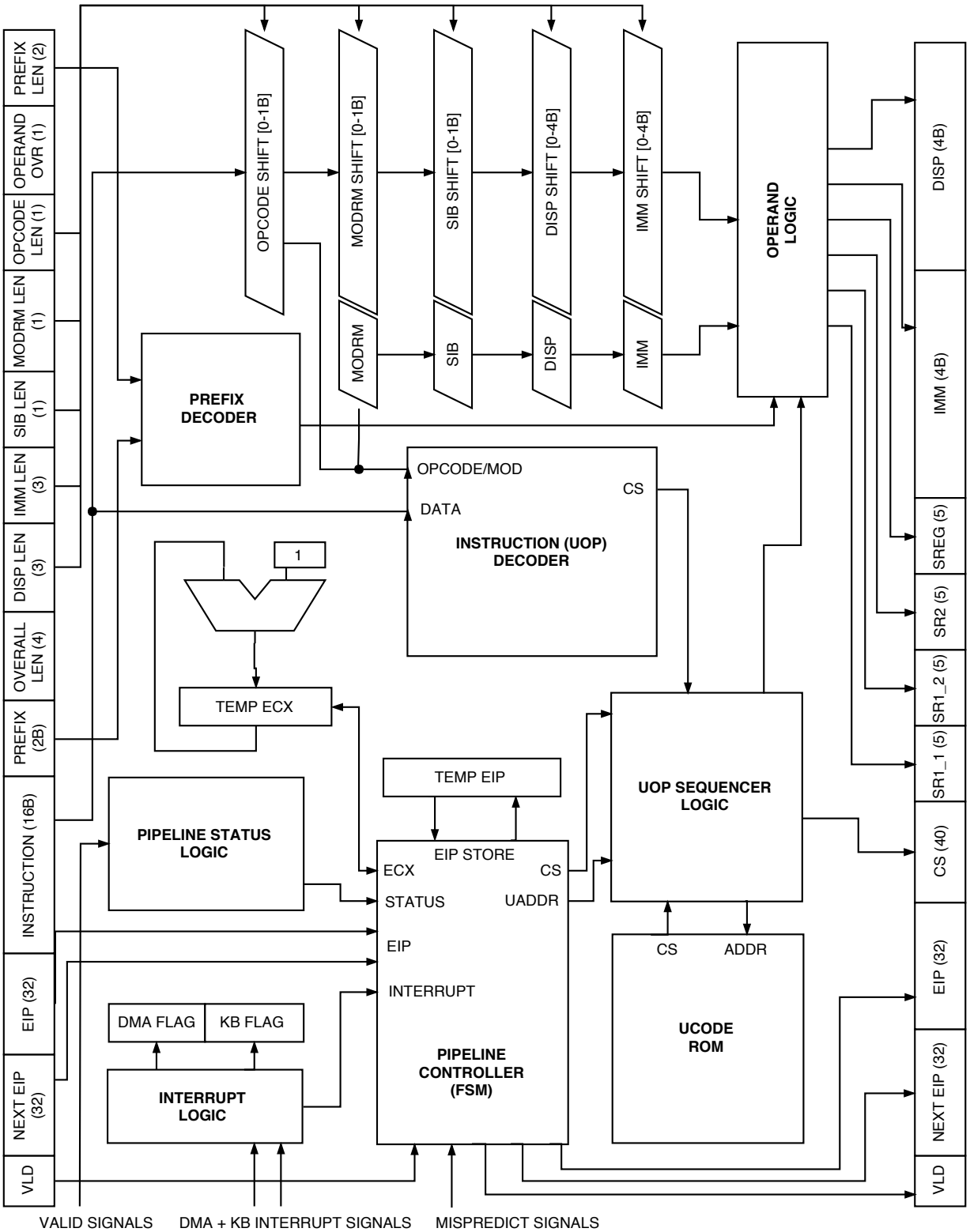


FIGURE 6:
REGISTER FETCH STAGE

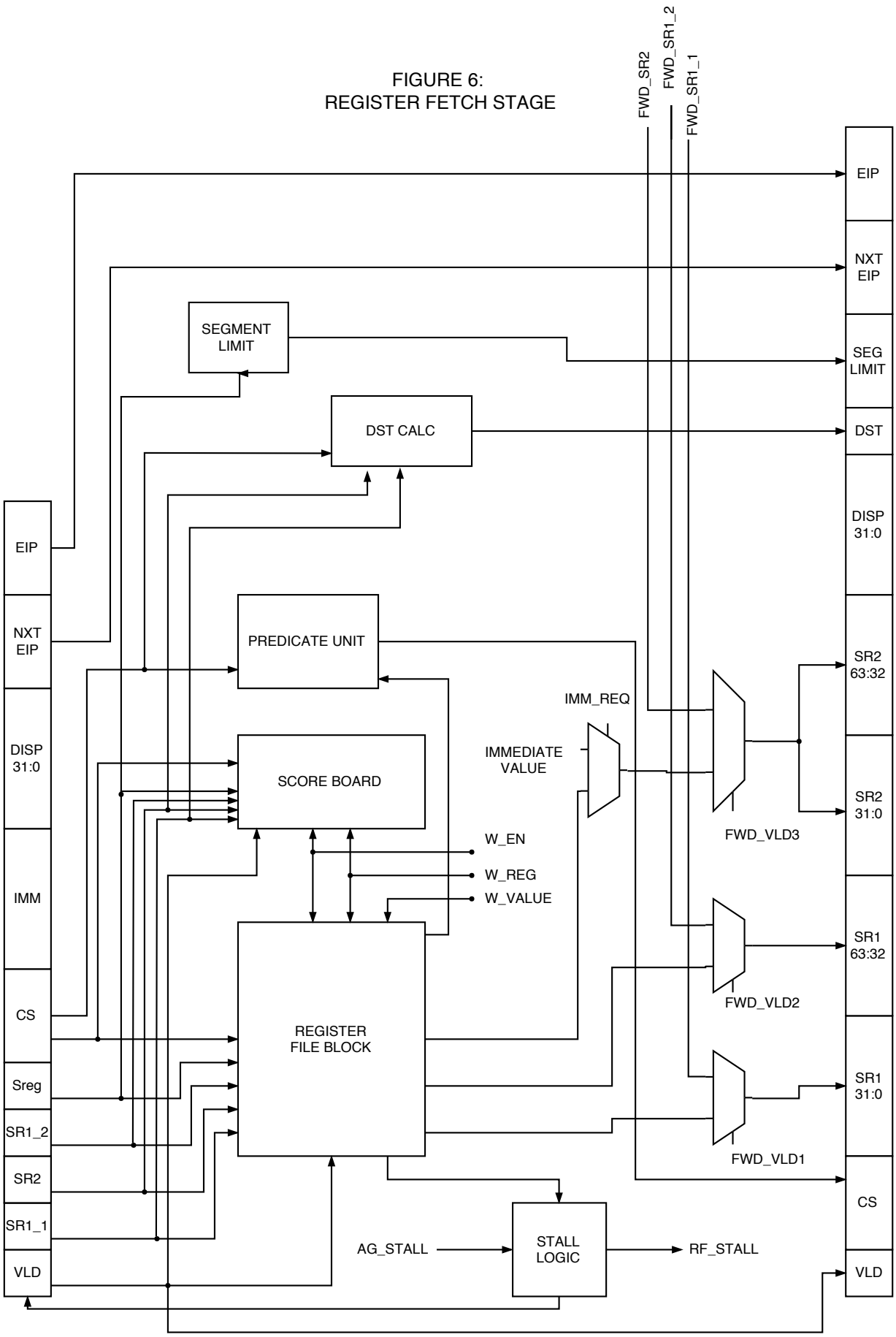


FIGURE 7:
ADDRESS GENERATION STAGE

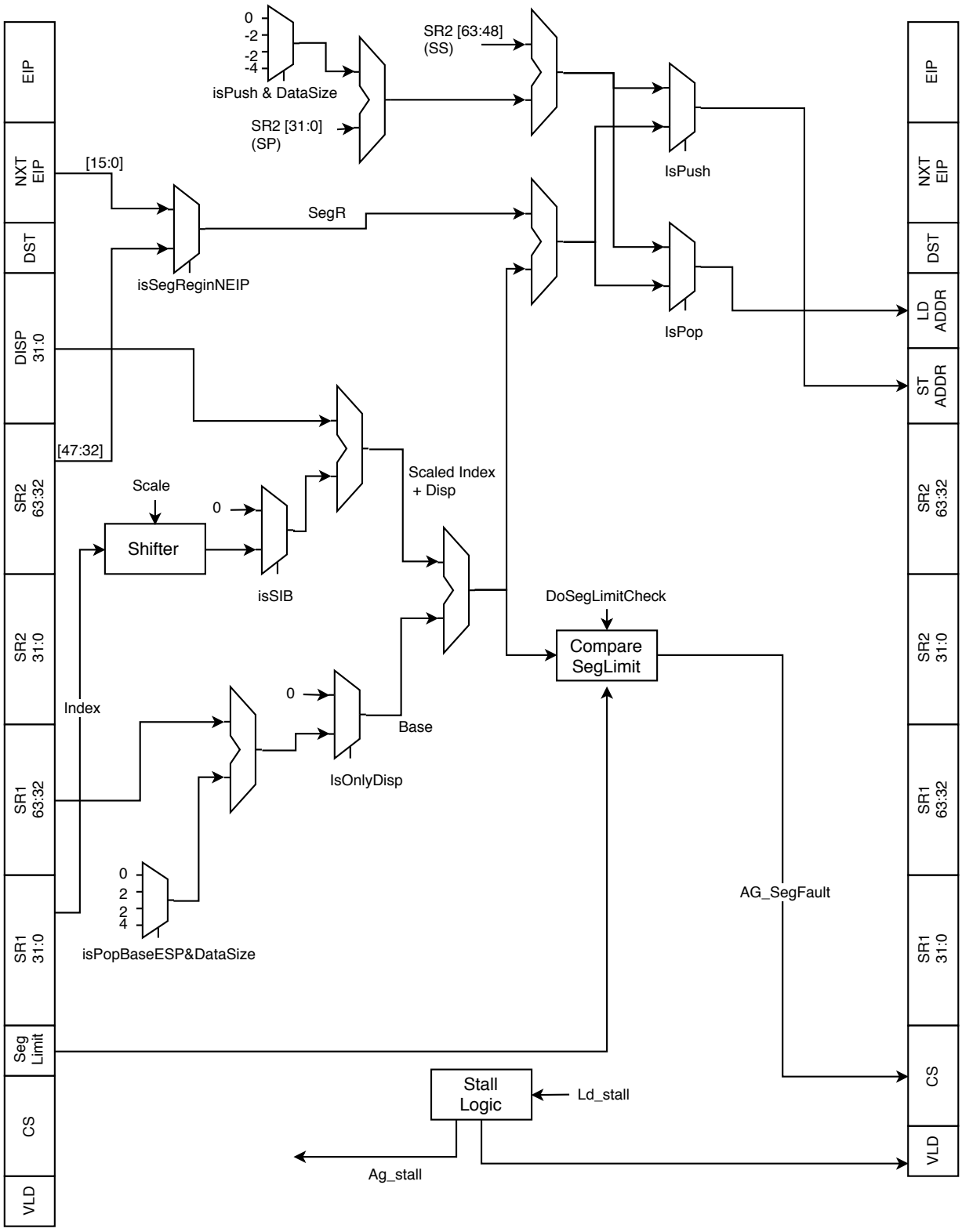


FIGURE 8: LOAD STAGE

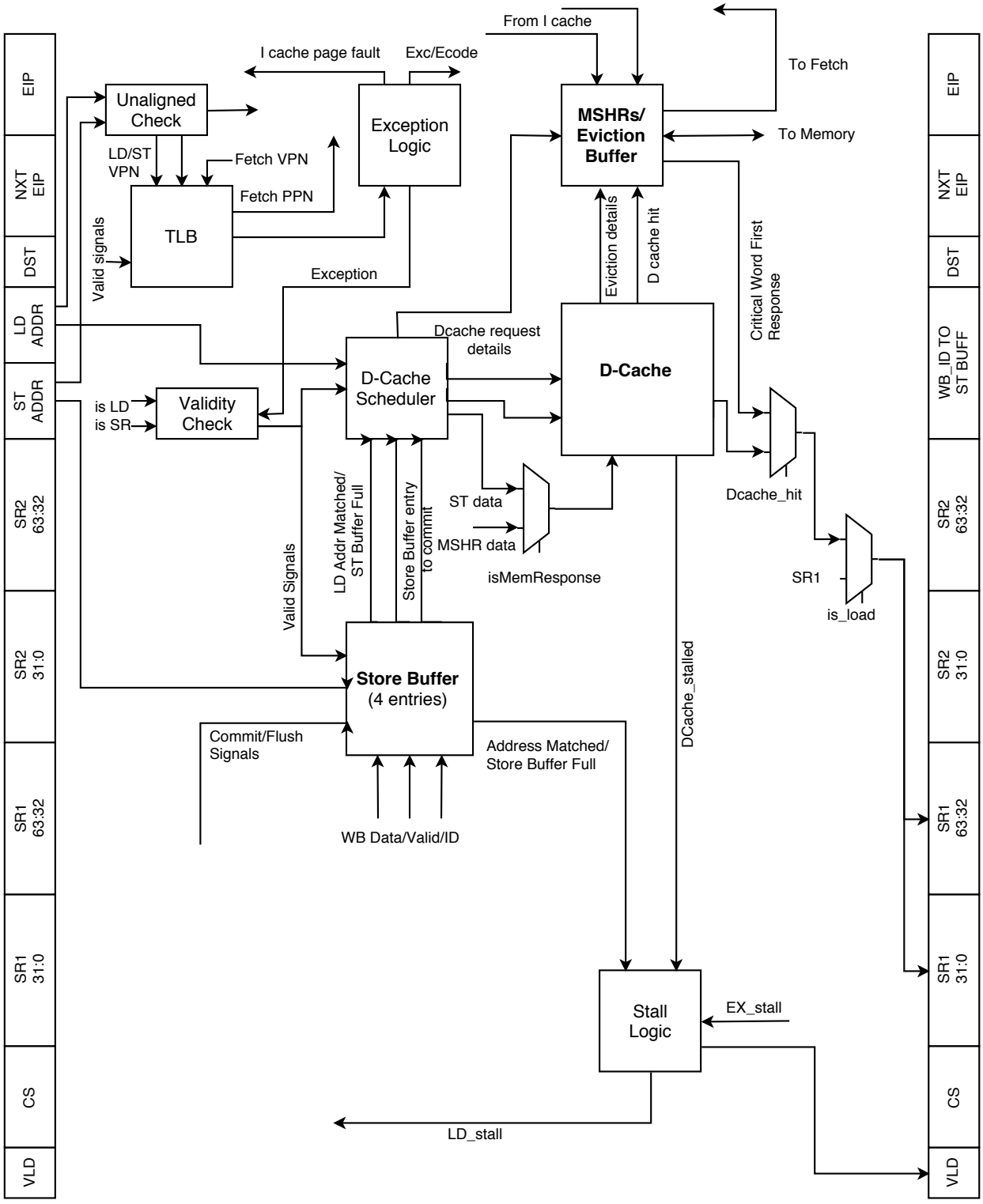


FIGURE 9:
EXECUTION STAGE

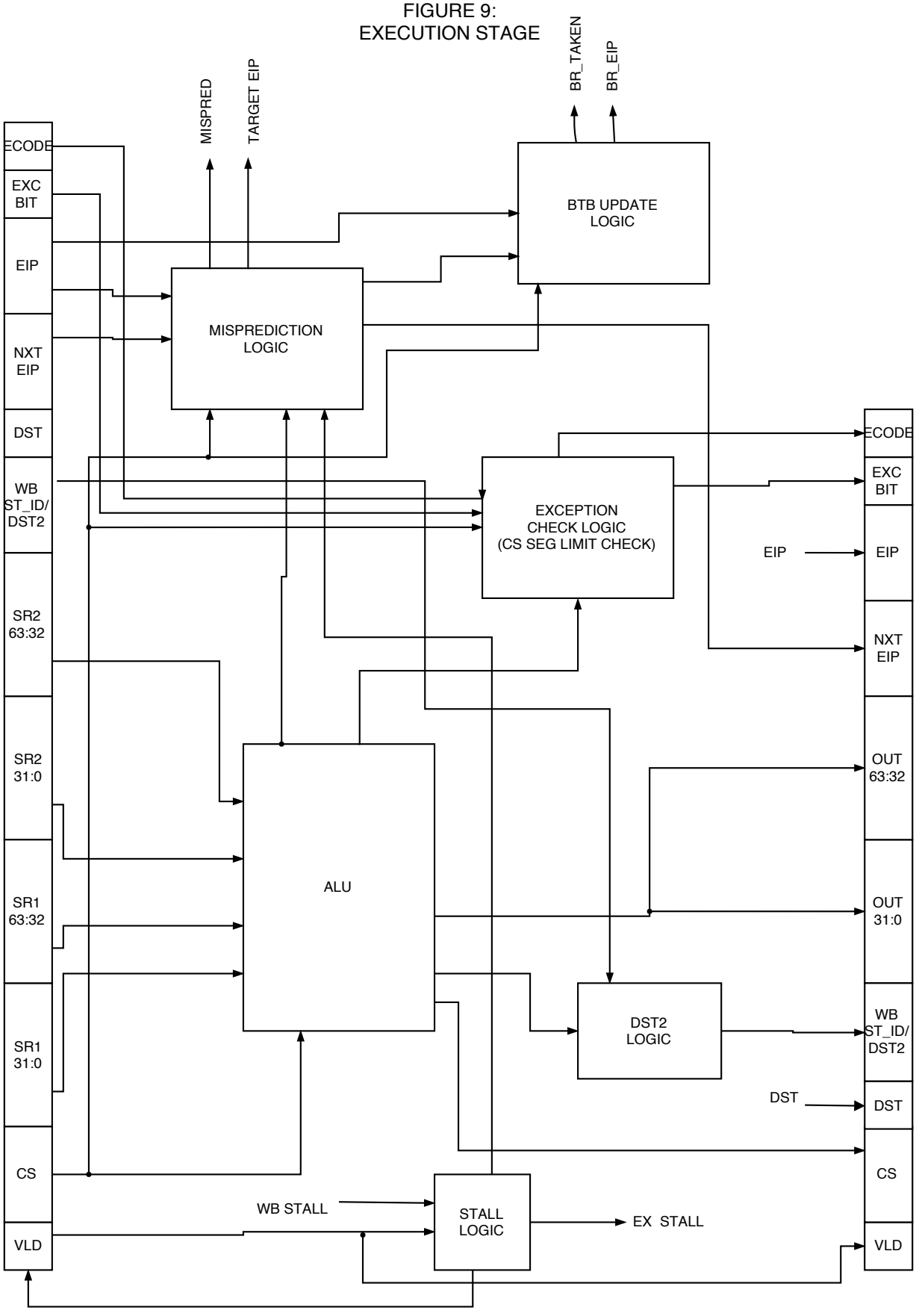
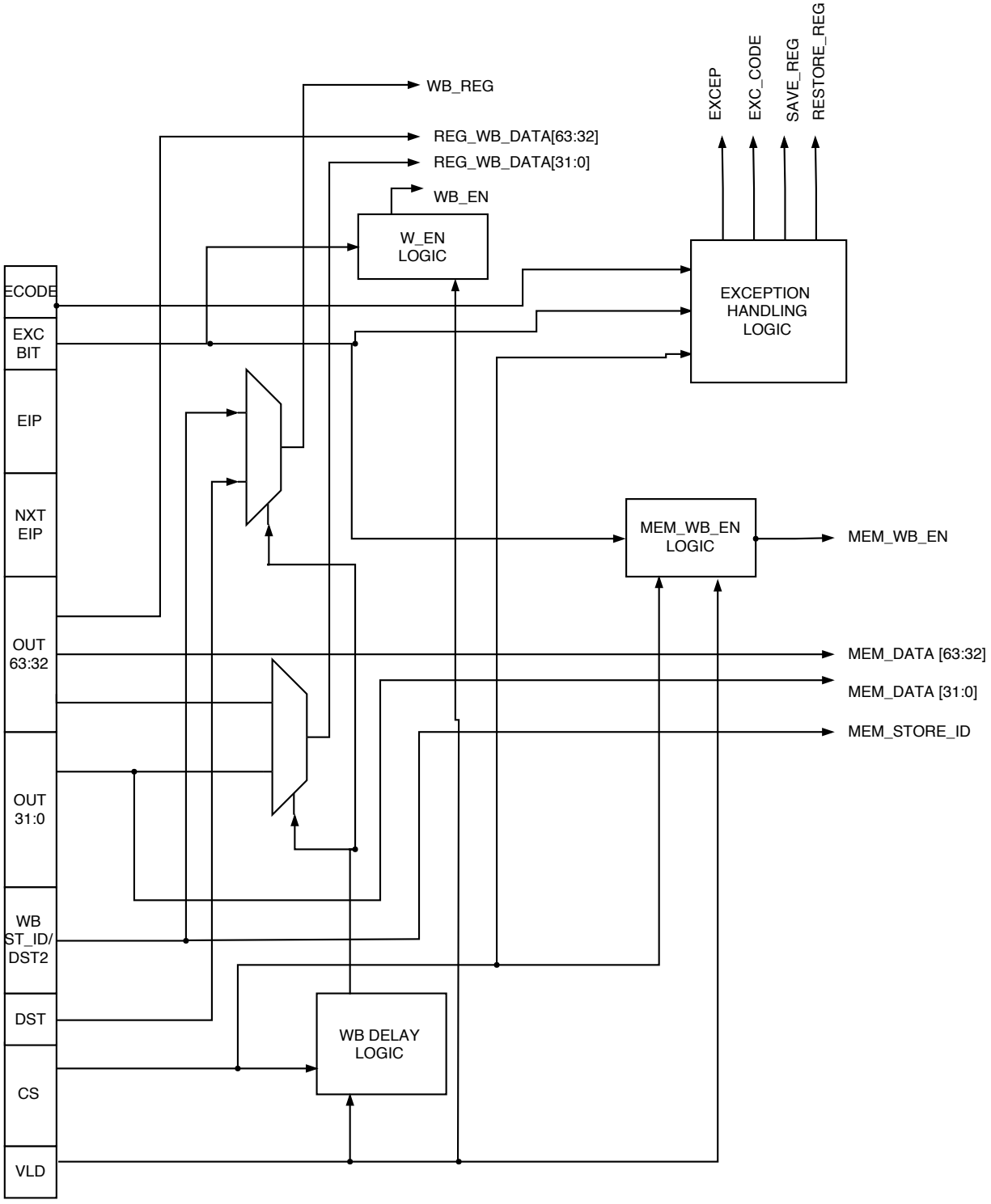


FIGURE 10:
WRITEBACK STAGE



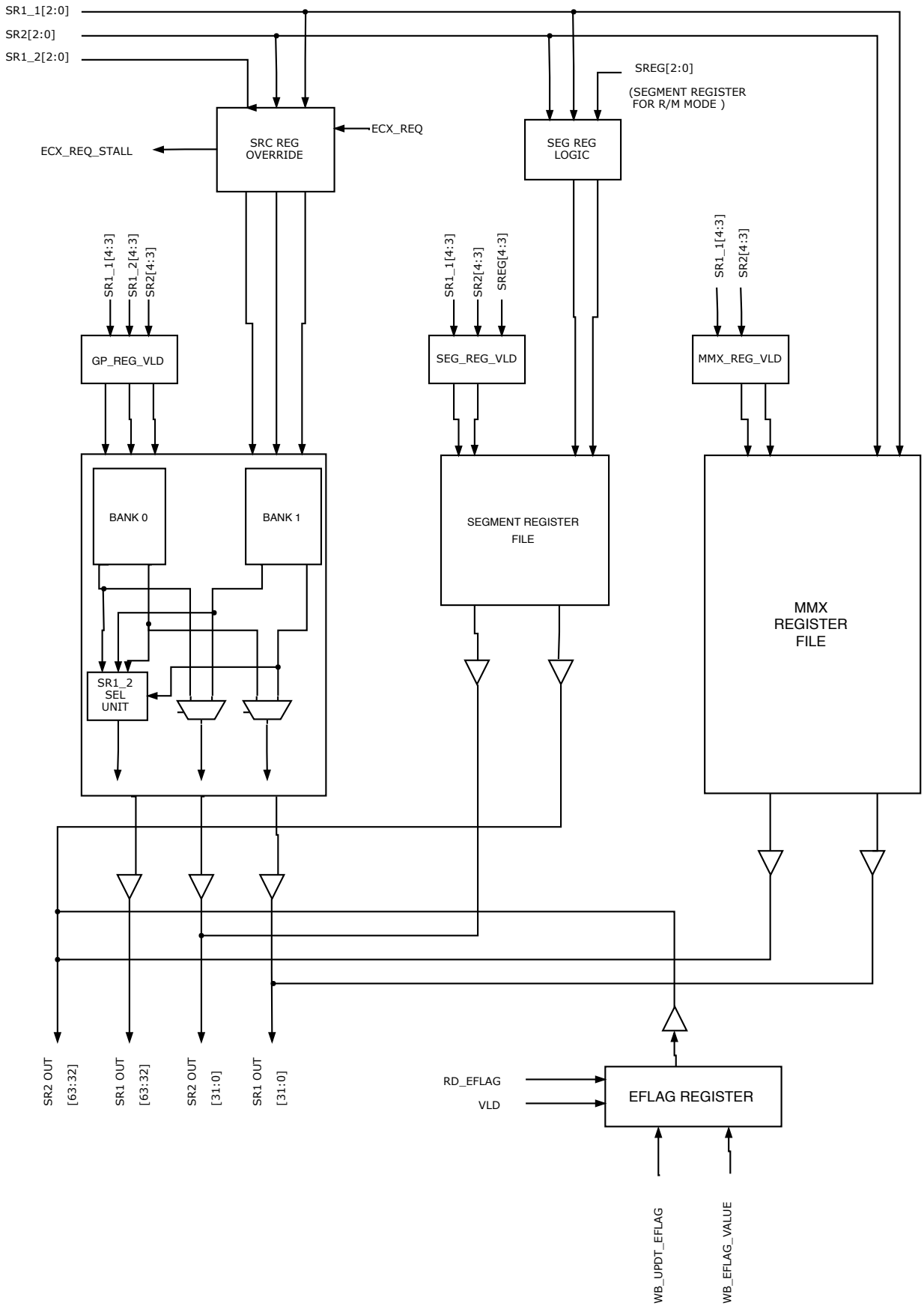


FIGURE 11: REGISTER FILE

FIGURE 14:
MEMORY CONTROLLER
STATE MACHINE