A Micro Architecture Design Report for a given subset of x86 ISA

Aniket Deshmukh

Anyesha Ghosh

Prateek Sahu

Department of Electrical And Computer Engineering

Final report for the course project for

EE 382N.19 under Dr. Yale Patt

May 2018

Abstract

This project is a culmination of 2 months of work toward the design for processor based on a subset of Intel's x86 ISA. It uses some fundamental design elements found in basic processors: pipelining, decoupled memory read and writes, register forwarding and branch prediction. The primary motivation behind this design was obtaining optimal IPC, while maintaining a respectable cycle time.

The designed processor supports all the instructions specified in the problem statement, with full support for General Protection and Page Fault exceptions, and interrupts from 2 devices. Memory is accessed through an I-cache and a D-cache, connected to a banked main memory via a 32-bit bus. The design also features a GSHARE predictor, register forwarding, and decoupled load-store queues. All the verification tests successfully passed when the design was simulated, and performed fairly well on the benchmarking tests.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 General Requirements

1. All opcodes and prefixes specified.
2. All addressing modes and data types specified.
3. Instruction and data caches.
4. The design and specification of the external bus.
5. The handling of cache accesses and memory, including the functionality of an MMU and basic segmentation.
6. The design of one simple and one complicated I/O device.
7. The handling of exceptions (e.g., segment limit) and interrupts (e.g., I/O device).
8. A responsible cycle time.

1.2 Salient Features

1. Pipelined
2. Two level GSHARE branch predictor
3. Non blocking caches
4. Banked Memory
5. Register Dependency Resolution before Write Back

Our design is optimized for IPC, while giving careful consideration to cycle time by adding lots of customized logic.

Chapter 2

Design Discussion

2.1 Fetch

2.1.1 Overview

The fetch stage is responsible for fetching instructions from the I-cache. It contains a base register: PC, which keeps track of the EIP (Extended Instruction Pointer) for the bytes in the current code segment. The fetch VA (Virtual Address) is constructed by adding the contents of this register to the current value of the code segment (CS), which is routed from the segment register file (SRF). For faster translations from the TLB, the most recent translation is cached in the fetch stage within a pre-tlb register; the VA is connected to both the TLB, and the pre-tlb register. A match in either results in a TLB hit, and the obtained page number is concatenated with the page offset to form the PA which is used to access the cache.

The PC register is then updated. The next PC is determined by 5 separate signals: with having a fixed priority (Fig. A.1, Pg. 46). These correspond to invalidation signals from the WB, EXEC, and DEC2 stages, a hit in the branch predictor, and PC+8 (normal increment). The appropriate EIP is chosen and written into the PC register.

The fetch phase also detects exceptions in I-cache access. Limit checking is perform on the bare output of the PC (since the limit is segment relative), violation of which causes a GP (General Protection) exception. The PF (Page Fault) exception is generated if no translation for the VA is found in the TLB. On either, the corresponding

exception opcode (8'hCC, 8'hDD) padded with NOPs is selected and sent to DEC2 instead of the cache line bytes.

The fetched/exception bytes are sent to the instruction buffer after concatenation with the address of each byte (PC), branch target, and other branch related information.

2.1.2 Branch Predictor

The branch predictor is 2 level gshare predictor with a global branch history register (BHSR), a global 32-entry PHT, and a 32-entry directly mapped BTB. The predictor is updated when a branch instruction is resolved in the EXEC stage. A prediction is read every fetch cycle, but is valid only if there was a hit in the BTB: this is because the micro-architecture does not support early branch resolution.

The PHT is indexed by folding bits [7:3] and [12:8] of the prediction PC or the update EIP. This is then XORed with the contents of the BHSR (the upper bits, for faster saturation of the counter for branches in within 32 addresses). Each entry of the PHT is a 2-bit saturating counter that is incremented or decremented based on the direction of the resolved branch. The predicted direction is read off the MSB of the counter.

The BTB is indexed with bits [7:3] of the prediction PC or the update EIP. Each BTB entry contains a valid bit, the branch nEIP (EIP+size of the branch, same as the incremented EIP after the branch) and its corresponding target EIP. A read address hits in the BTB if the valid bit is set and the truncated prediction PC (bits[31:3]) matches with the same bits in address of the branch in the BTB.

2.1.3 Instruction buffer

The inputs to the instruction buffer consist of size of the current instruction in the buffer and the fetched entries. We fetch 8 bytes from the cache, and up to a total 24 bytes can be written to the instruction buffer. This number was derived from the fact that the length of 98% of instructions in a typical x86 program is under 8 bytes in size. The valid entries in the buffer are indicated by an empty pointer register which is initialized to zero. The empty pointer points to the start of the invalid entries, i.e. the

first entry that can be written by newly fetched data. The pointer is incremented by the fetch width when valid entries are available at the fetch and the buffer isn't full. The pointer is decremented when an instruction is read from the instruction buffer, by the size of the instruction that is read. Both operations can happen simultaneously, and the pointer is modified accordingly. The pointer increment and decrement logic is shown as part of DEC2 in Fig. A.2 (Pg. 47).

To shift the entries into the instruction buffer, fetched entries (if any) are appended to the end of the last valid instruction buffer entry. This is achieved by multiplexing the output of each instruction buffer register with the fetched entries. Which fetched entry is used is based on the empty pointer; Fig. n shows the structure of the shift tree.

2.2 Decode

Decode is divided into two stages. The first stages calculates the instruction size and latches the corresponding bytes of the instruction buffer into the DEC2 latch. Along with the bytes, information about each byte of the instruction generated during calculation of the size and key bytes (opcode1, opcode2, modRM and SIB) of the instruction are also latched.

2.2.1 DEC1

Size calculation

Six modules in decode are responsible for generating the signals, enumerated in 1 to 6.

1. op_pre: Compares the first 3 bytes with the provided list of prefix bytes. A prefix in byte 0 is always valid; for byte 1, the prefix is valid if the byte matches a prefix and the previous byte was a prefix, and so on. The module also generates the number of prefixes in the current instruction which allow for further decoding of the instruction.

2. op_op2: The first opcode byte is the one after num_prefix bytes. This is compared with 8'hF0 to check whether the instruction has a second opcode byte.

3. op_imm8: The opcode bytes are passed through a comparator array that checks for whether the opcode has an associated byte immediate.

4. op_imm16_32: The opcode bytes are passed through a comparator array that checks for whether the opcode has an associated multi-byte immediate.

5. op_modrm: The opcode bytes are passed through a comparator array to check for whether that opcode comes with a modrm byte. The modrm byte is then forwarded to the next module.

6. fn_modrm: The modrm byte is examined and displacement size, and whether the instruction has an SIB byte is determined.

The final size calculation uses the signals described in Table 2.1. Sizes for all possible input variants are calculated (max size is 13) and placed along with the 11 input signals. This forms an 11-input truth table for each of the 4 bits of size. The corresponding logic was generated using a logic minimizer (Espresso).

Table 2.1 Components of size calculation

- has_prefix1
- has_prefix2
- has_prefix3
- has_opcode2
- has_imm8
- has_imm16
- has_imm32
- has_modrm
- has_sib
- has_disp8
- has_disp32

These are also forwarded to decode 2 as the help reduce the amount of replicated logic.

The stall and invalidation logic at the input latch (instruction buffer) is based on the contents of the empty pointer register. When full, the instruction buffer stalls. When not enough bytes for decode are available, the output is invalid.

2.2.2 DEC2

The second decode stage generates the actual control bits that are used by the pipeline. Majority of the control signals come from the control store. Access to the control store is made using the opcode and modrm bytes. The two opcode bytes in combination with the opcode extension field (r/m) of the modrm byte are sufficient to uniquely identify an instruction (or instruction variant). These are fed to a comparator array that determines the address of instruction (variant), and are part of the control store. Schematics can be found in Fig. A.3 (Pg. 48).

Control Store

Each each instruction is composed of one or more uops, which are defined by their control signals. DEC2 issues multiple uops using uop pointer register that starts of at 0. Bit 30 of the control bits determine whether the instruction has multiple uops, and the uop pointer increments every cycle till a 1 is seen in the fetched control store entry. The final address used to access the control store ROM is the sum of the address obtained via the comparator array and the uop pointer. Successive uops of the same instruction (variant) are thus stored in consecutive locations.

Control bits override

The control store entries are not sufficient for all the control signals in the datapath. Some signals are generated through combinational logic directly from the opcode, modrm and SIB bytes. Other signals are modified control store signals based on the opcode, modrm and SIB bytes. Instruction group specific signals are drawn directly from the comparator array for address calculation. The list of control bits that are overridden or generated directly through the decode bytes and their corresponding generation logic is specified in the Table 2.2 below.

DEC2 special opcodes:

Certain opcodes in decode lead to special behavior. This are listed below:

- SETD, CLD: The DFlags register is located in the decode stage. It is read from each cycle, and written into when the said instructions arrive. On an invalidation, the value of the the DFlags register is changed to that of the the instruction that caused the invalidation.
- HLT: On a halt, DEC2 stop issuing further instruction and stalls. It also all the stages in the pipeline before DEC2, effectively stalling the whole pipeline. The instruction can however be invalidated, which allows prior branches, exception and CS writes to overwrite the halt. In the present design, an interrupt during halt will invalidate the halt, and execute the interrupt handler.
- EXCP: An exception can arrive at DEC2 either from the datapath or the FETCH stage. A datapath exception results in invalidation of the current instruction. At the end of the cycle, the exception opcode (8'hCC for GP, 8'hDD for PF) is latched into the opcode1 field. A FETCH exceptions' opcode is directly latched the opcode1 field since it is treated as a regular instruction. Following this, the uops for the EXCP opcode are fetched, and executed in the datapath. The IDTR is also present in DEC2, and send the address of the IDTR entry corresponding to the exception vector along with the last uop of the exception, which is JUMP instruction. The final jump loads the address of the exception handler into the PC register in the FETCH stage.

Table 2.2 Control Store Override Signals

Bits	Description	Combinational Logic
[61]	Return Immediate	opcode1==(C2 | CA)
[60:56]		Passed
[55]	SR1 read_en	0: [SIB]|[disp]?0:control_bits[55] Override ignored if 1st uop of POP, 2nd uop of PUSH
[54:52]	SR1 src	SR1 comes from R/M field for indirect jumps
[51]	SR2 read_en	1 for indirect jumps
[50]	segR1 read_en	0 if mod is 11 1 if push or pop control_bits[50] otherwise
[49:46]		Passed
[45]	Mem read_en	0 if mod==11 1 if pop reg control_bits[45] otherwise
[44]	DR1 write_en	1 if mod==11 && control_bits[43] Control_bits[44] if
[43]		Discarded
[42]	DR2 write_en	1 if mod==1 and opcode==XCHG Control[42] otherwise
[41]		Passed
[40]	Mem write_en	0 if mod==11 1 if pop reg control_bits[40] otherwise
[39:32]		Passed
[31]	ALU pass	0 if mod==1 and opcode==MOVQ Control[31] otherwise
[30:0]		Passed

2.3 Register Read

The Register Read stage of the pipeline takes care of the read interaction with various register files of the processor. This is the only stage which deals with register reads. Our design implements 3 register files namely

- The General Purpose Register File (GPRF) : 8 x 32 bit entries
- The Segment Register File (SRF) : 6 x 16 bit entries
- The streaming SIMD Register Files (aka XMM): 8 x 64 bit entries

All the three register files are dual read and single write port the standard). The Register Read stage reads information from the appropriate stage as required by the instruction. The Register read gets a maximum of 2 Segment Register reads and 2 GPR or XMM reads for a particular instruction, and 3 GPR reads for SIB instruction. For SIB variant of CMPXCHG instruction, the Register Read stage can get 4 GPR register reads, two pertaining to the Index and Base registers, one for the source register and one for (E)AX register.

This stage (Fig. A.4, Pg. 47) gets each of the Segment Register reads from input latches in the form of SegR1 and SegR2. Each of them is associated with a corresponding ReadEnable bit which is passed to the Register File to get the corresponding data for the given register number. SegR1 always depicts the segment register used for calculation of memory addresses, usually the DS. SegR2 depicts the register number if a particular Segment register is involved as the source, or destination operand in an instruction. SegR1's data, termed "SegRc1" is summed up with the given Displacement value to for the base address denoted by "addr1". With each read of Segment Register File, we get the corresponding Segment Limit for the given Segment as well.

The GPRF or XMMRF are indexed by the same register number provided by SR1 and SR2 in the register-read(RR) latch. These also have a similar ReadEnable signal associated with them. An extra XmmOp control signal determines if the GPRF or the XMMRF is indexed for data for SR1 and SR2. In case of a memory operation, where the address is generated by the register value, we always index using SR1 into the GPRF even if the XmmOp control signal is set. In the case of an SIB instruction, the corresponding base and index registers are provided by the Base and Index latches in RR and have an associated ReadEnable bit of their own. These are always read from the GPRF. In case of memory operations, the "src1" value is used as indirect

addressing value and this value is assigned to the "addr2" value which is further used by Address Generation Stage.

Separation of addr1 and addr2, instead of adding them to form address, in this stage was a design decision since addr1 in case of SIB involves a 32-bit adder and formation of address using this could lead to two 32-bit full adders in a pipe-stage which affected the critical path of the design.

The "src2" value is muxed with immediate values being provided as a source which is then latched at the end of the clock cycle for further processing. The Immediate is sign extended to 32 bits appropriately according to size provided by Decode for the instruction. SR1's value, "src1" is muxed with the displacement value for JMP instructions, because the displacement latch provides the Code Segement value for a far-jump instruction.

2.3.1 Register dependency check

The registers SR1, SR2, Base and Index used to index into GPR or XMM are checked with corresponding register values (termed DR1, DR2) from each subsequent pipeline stage, i.e Address-Generation, TLB-Access, Memory-Access, Execute, and Write Back to determine dependencies. In case of a dependency, the pipeline stalls till the corresponding register value has been correctly written back to the register file. We have implemented register data forwarding between Write-Back and Register Read stage, which implies, a requested register data if is available in Write Back to be written-back to the the Register Files, it can be forwarded to the Register Read stage. The dependency check logic selects between the correct datas in WB to appropriately forward the data to the requested register.

2.3.2 2-cycle SIB operation

If an instruction has SIB addressing and has a register read as well, like in instructions like ADD m32, r32 where the m32 is SIB addressed, we need a total of 3 register reads. In such cases we makes the Register Read span two cycles using a FSM to read the Index and Base registers and then the Source/Destination register subsequently.

The output is valid at the end of the two cycles with the value in the r32 being one of "src1" or "src2" and the Base and Index register values being added appropriately to for "addr2" value.

For fast implementation, the Segment Limit is first decremented by 3 (or 1 in case of 16 bit operand Size) using custom logic which calculates the subtraction in half of the time for a standard 20 bit CLA adder circuit. With this, the SegRc1 is added to get the maximum possible start of the request address for no violation to occur.

2.4 Address Generation

This stage of the pipeline calculates the virtual address of a memory instruction either to be read from or written to for the completion of the instruction. The virtual address is calculated by summing up of the direct and indirect address generated by Register Read stage, names "addr1" and "addr2". In case of direct addressing, address given by "addr1" is taken as the virtual address. The direct addressing is a control bit provided by Decode2. Apart from virtual address calculation, this stage also sums up the segment register value (SegRc1) for address calculation with the corresponding Segment Limit. This gives the valid Segment Region which can be accessed by the instruction. Design schematics are provided in Fig. A.5 (Pg, 50).

2.4.1 Early dependency resolution for stack operations

Such custom increment and decrement adders were also applied on "src1" value. The rationale behind this was, for Stack operations and instruction using CMPS, the ESP register value, and ESI & EDI registers respectively, get incremented or decremented according to the D-Flag and operand Size. The Size of increment or Decrement are 1, 2 and 4, which have been implemented using a modified CLA circuit with input B = 0 and Cin = 1 for increment and B = 11..1 for decrement calculation which reduced any n-bit CLA increment/decrement time by half of a traditional n-bit CLA adder.

The rationale behind calculation of new ESP value (or ESI and EDI values) was to provide an opportunity for early dependency resolution in case of continuous Stack operation or cases like REPNE CMPS where the updated ESI and EDI values are required in high frequency. Consecuitive memory Push and Pops for instructions like CALLS, IRetd, RET and Interrupts etc get the benefit of this in the current

implementation but this can be easily extended to all the above mentioned cases. Due to shortage of time, this enhancement was partially implemented.

2.5 TLB-pipe

This stage accesses the TLB with the given virtual address from Address Generation to get the translation entry for the physical page. The TLB is accessed only on a valid memory read or write instruction. The TLB returns the translation entry which contains both the VPN and the PFN. The corresponding PFN is concatenated with the last 12 bits from the Virtual Address to form the Physical Address. Please refer Fig. A.6 (Pg. 51) for the schematics.

Along with calculation of physical address, the TLB also provides information like if the page is Writable or just Readable. Incase of permission check violation due to write to a Read only page, the TLB generates a general protection exception and forwards this information along with the current instruction EIP to the decode to take appropriate actions. Similarly, in case no translation entry is found in TLB for the corresponding Virtual page number, we get a page fault from TLB which is also forwarded to decode. Along with these, the virtual address is also compared with the segment base address and the limit address to verify if the requested addresses were valid to the particular segment. In case of a limit violation, a general protection exception was raised again.

With the access to TLB, this stage also readies the d-cache access and the valid signals to be used by the cache at the clock edge. In case of any kind of fault or invalidation signals propagated from later parts of the pipeline, the valid signal is reset. SInce the D-cache request is queued in the MSHR, which is only 4 entries deep, we send request to the D-cache only when the instruction is about to leave this stage of the pipeline. Hence, in case of a stall, we do not send the memory request, rather wait for the stall to complete to send the memory request.

Hindsight : It makes more sense to send the request when the address is ready, such that any miss latency is hidden by the stall latencies. But for that, there needed to be some extra communication logic between the processor and D-cache such that D-cache does not return the data while the instruction is still stalling. This seemed a bigger

complication to introduce at a later stage of design, hence we went ahead with the above approach.

Incase of Interrupts or Exception, the TLB access is invalidated and the IDT Vector received bypasses the entire translation logic to be requested from memory. Since we require 8 bytes from memory to process the information, the whole stage again requests memory over 2 cycles, once with the address and next time with address+4.

2.5.1 Memory dependency check

TLB stage needs to send only a single memory request per access so as not to overflow the MSHR. This mandated us to have the Memory Dependency Checker in this stage because any stall due to memory dependency needs to be re-requested to get the correct data and only TLB stage prepares a D-cache request. This was also necessary because we did not implement memory dependence data forwarding.

The memory dependence check calculates the maximum and minimum physical address it touches based on read request size. Similarly it calculates the maximum and minimum physical addresses any of the later stages writes to. With these addresses, the logic just checks if any of the request addresses overlaps with any write addresses. In case of a match, the logic issues a dependence and the stage stalls till the corresponding location has been written to. For this we need to compare against all the memory writes pending the write butter as well. This dependence logic involves a 15-bit adder (to calculate the maximum address) and a 15-bit magnitude comparator, followed by a OR Tree. To avoid this from being in the critical path to calculate dependence, and hence stall logic, we implemented custom size adders, which involved 2-bit RCA for the lower bits and a parallel +4 increment logic on physical address and finally muxing out the upper bits based on the carry out of the 2-bit RCA.

2.5.2 Size calculation for page spill

Page Spills needed two different translations and two subsequent memory acesses for requested addresses. Complications for page spills arose because of unknown request sizes. Any 8-bit memory request could never page spill, while a 2-byte memory request would not spill pages for certain situations which a 4-byte memory request would. To get around these, there has been a good amount of custom logic to figure out spills based on the final 12-bits of VA and the operand Size of the instruction. In case of a

page spill, the TLB pipe stage completes over two cycles with the Virtual addresses corresponding to VPN and VPN+1. Each of the request is passed on to memory stage while the AG and earlier pipeline stages stall.

2.6 Mem-pipe

This stage waits for memory to return valid data. Upon receiving a data valid signal, the stage checks if the returned data is for the address that it requested for. This check is done by comparing the address returned by the D-cache and the request address in the pipeline latch. In case of valid data, the stage checks if there was a page spill in the previous stage. If there was, the stage latches the received data into a register and accepts the new request from TLB-Pipe stage. Once this data has been received, the stage concatenates the two received data, based on the request sizes to form a valid word to be latched into Exec latch for further processing.

Due to nature of implementation, the memory data received can be muxed with either of the two source operands based on type of instruction or micro-instruction. Schematics of this stage are provided in Fig. A.7 (Pg. 52).

For interrupt and exception memory response, the “spill” signal again waits for both data and then processes the 8 bytes of data to extract the CS and EIP values which are latched onto “op1” and “op2” respectively.

2.7 Execute

This stage does most of the heavy lifting of the compute part of the computer.The stage consists of an ALU logic which does one of the following 8 operations -

- NOT
- AND
- OR
- Shift Arithmetic Right
- Shift Arithmetic Left
- 32-bit Addition

- 16-bit Addition with knob to saturate the counter
- Pass A/B based on signals

Apart from these, there exist individual blocks of logic to compute DAA and SIMD Shuffle word instructions. Execute stage is shown in Fig. A.8 (Pg. 53).

2.7.1 ALU implementation

The ALU has been traditionally implemented with proper considerations given to timings. Since the time critical portion for the ALU block is a 32 bit Adder, we decided to implement a CLA. Any parallel prefix adder would have done the job much faster, but considering the significant latencies in most other stages, it seemed a wasted effort to implement a significantly harder adder without any added benefits. With that in mind, most muxing of other ALU operations like for PASS, AND, OR, etc are done separately and earlier so that we have minimal mux latencies for critical length 32-bit adder result to be produced by the ALU. The Shuffle Word logic have been shown separately instead of being clubbed within the ALU since the input operands varied significantly for Shuffle. Keeping DAA separate made sense in terms of portraying the fast path for DAA which runs parallely and faster than the rest of the design. Please refer Fig. A.17 (Pg. 62) for internal schematics of ALU Block.

Flags calculation

Flags are computed for many of these operations like Add, And, Or, SAL and SAR. The implementation calculates flags from the result for almost every arithmetic operation. Finally, “Flags Modify” control signal, which are appropriately filled for each instruction variant, decide if the computed flags are selected or the existing value of the flag remains. The implementation details have been shown in Fig. A.20 (Pg. 64).

Similarly there are two other flag related signals, namely “flags used and flags compared”. These are also part of the control store. Flags Used bits are set for flag bits which are used for execution of a particular conditional instruction (eg conditional JMP). The “flags compared” bits determine what is the particular bit polarity is supposed to be for the condition to be satisfied. If all the relevant flag bits match, then the conditional instruction proceeds as normal, else the effects for the instruction are nullified.

2.7.2 DAA implementation

DAA has been implemented as per the given pseudo code in the ISA manual. Due to lack of constraint on resource utilization, we implemented three add operations, namely +x6, +x60, and +x66, to cover every variant of the flags. Then based on the existing flag values, the appropriate sum is chosen as the output of DAA logic. DAA implementation are enumerated in Fig. A.19 (Pg. 64).

2.7.3 Branch target resolution

Apart from the traditional working, the Exec stage, also calculates what the branch prediction target was for JMP instructions and then sends out a misprediction signal to fetch and decode along with the correct branch path and the target. This misprediction invalidates all of the previous latches since the fetched instructions were incorrectly executed. The logic implementation are shown in Fig. A.21 (Pg. 65).

2.7.4 Synchronization for multiple u_op instructions

A quirk in our Exec design is the handling of SIMD instructions. Since our processor to D-cache line is limited to 4 bytes, a memory SIMD instruction takes two cycles to fetch the data. This made us take the design decision to handle SIMD and consequently all multi-micro instructions over two cycles in exec stage. SIMD instructions process the first word of information and save the result in a latch till they get the next micro instruction which completes the processing and then the 8 bytes of data is passed to Write Back stage for retirement.

Due to this, we also designed other instructions such as CMPXCHG and CMPS over two cycles as well. The first cycle fetches one of the source operands for comparison, and the second micro instruction fetched the next operand, which is then added to the invert of the previously saved data (with carry in of 1) to handle the subtraction. This made processing of instructions quite simple with a lot less resource utilization.

The operation result of the ALU is stored in the latch named “data1” while any secondary write-back information like incremented or decremented stack pointer, or updated ESI and EDI register values, which are parallely calculated with the instruction are latched into “data2”. “Data2” contains the high bytes of SIMD operation result. Since we have two valid data in WB, the Write enable signals for registers and memory operations are appropriately transformed into 2-bit signals, with lower bit set indicating

"data1" is to be written into the corresponding register, and higher bit set indicating the same for "data2".This is done using a combinational logic for the variety of provided instructions.

For Return instructions, the exec stage, syncs up all the micro-instructions and then proceeds to WB stage. Return instructions pop out EIP and appropriate information and then jump to the popped EIP location. The Exec stage latches in the eip, EFlags and CS values over the various micro-instructions and then writes them all at once in the WB stage. This is done to avoid updating the Code Segment value mid-instruction and then having additional logic to handle such a scenario. The early resolution of ESP values in AG stage helps accomplish this, since it could have resulted in a deadlock situation otherwise.

The same is not true for CALL. Exec does not wait for last-uop signal to spew out valid write back instructions since most of the instructions are Push instruction with the final instruction being a relative jump, whose exception is done by fetch in the next cycle.

2.8 Writeback

This stage is responsible for retirement of instructions. Any valid input latch is considered ready to be retired and the appropriate data is written back into registers or memory. This stage has a 4-state state machine to take care of multi-page SIMD write which can span writes to 3 different address locations. Considering a D-cache bus width of 4 bytes, the first 4 or the second 4 bytes can spill across page boundaries, but not both. The WB also takes two cycles to write to two GPRF registers for instructions like XCHG. The write can occur to two GPRs, one segment register, one XMM register or two 4-byte memory writes. Each of the write enable signals are appropriately set in Exec to indicate which of the two datas are to be written to the corresponding destination address. All register writes are single cycle. This stage design has been provided for reference in Fig. A.9 (Pg. 54).

Any write to the Code Segment register results in an invalidation of the entire pipe since all subsequent instructions have been fetched from a wrong instruction base. This signal is also propagated to Decode.

2.8.1 Write buffer

The WB has a 4-entry write buffer. Each entry of the buffer has a valid, write Size, write address and the 4-byte data to be written back. Since we have a write allocate cache, we are guaranteed to hit in the D-cache during a write except for cases when the particular address in non-cacheable. In case of a valid memory write, the appropriate data and address are put into the buffer and this buffer interacts with the D-cache to send out write requests. Although the data is always a D-cache hit, sometimes the write does not complete in one cycle and a second request immediately the next cycle leads to a wrong execution of the information. Due to this, the D-cache sends a write acknowledgement to the buffer, which acts as a trigger to send the next valid write from the buffer to the D-cache.

The Write Back buffer has been kept 4 entries deep due to need of the current design. In case the buffer is full, it sends out a stall if the WB tries to write memory data to a full buffer.

2.9 Memory Subsystem

2.9.1 I Cache

The I-Cache is an 8 line, directly mapped cache. It is physically indexed, physically tagged (PIPT). The line size is 32B, giving a total cache size of 256B. The large line size effectively acts as a simple prefetcher for instructions, and effectively makes use of the locality of instruction accesses. The I-cache is shown in Fig. A.12 (Pg. 55).

I-Cache Operations

To read a cacheline, the sequence of operations that take place is:

- The cache starts off in the IDLE state.
- On a read request, the provided PA is passed through a cacheline spill calculation unit, which determines whether the requested data spills across a cacheline boundary.
- The I-xache spill register is filled (see below).
- A read request is sent to memory (address = rdaddr) if the access misses in the cache.

- The next state is calculated:
  - if(hit & !spill) next <- IDLE; return data to the processor.
  - if(spill) next <- REQ;
  - if(!hit & !spill) next <- WAIT
- In the subsequent cycle(s), two cases can occur.
  - if(state == REQ)
    - Access cache with the next cacheline address (rdaddr[14:5]+1,5'b0).
    - If(hit), fill the spill register.
    - If(miss), send a read request to the memory.
    - Calculate the next state:
      - if(hit & spill_reg.v0) next <- IDLE.
      - Else next <- WAIT.
  - if(state == WAIT)
    - if(mem_data_vld) Fill the spill register.
    - if(spill_reg.v0 && spill_reg.v1) next <- IDLE, return data to processor.
    - else next <- WAIT;

Filling the Spill Register

The spill register has the following structure: v0,v1,spill_location,address,data. The same structure & fill algorithm is used for the ICache, WB spill register in the DCache & the entries in the DCache MSHR. The pseudocode for this mechanism is:

```
shift&insert(data_in, spill_reg){
  //This function inserts data_in into spill_reg.
  if(data_in is 1st cacheline of a spilled request,
  or cacheline of an unspilled request){
    data_in_temp <- data_in >> spill_reg.addr[4:0]; // ICache
    data_in_temp <- data_in >> spill_reg.addr[2:0]; // DCache
    data_in_temp <- (data_in_temp with upper bytes zeroed out
                     according to spill_reg.spill_location);
    Spill_reg.data <- spill_reg.data | data_in_temp[31:0];
```

```
  }
  if(data_in is 2nd cacheline of a spilled request){
    data_in_temp <- data_in << (4-spill_location);
    Spill_reg.data <- spill_reg.data | data_in_temp[31:0];
  }
}
On allocate{     //i.e. On a read/write request
  //v0: Whether the 1st part of the request has valid data.
    v0 <- hit //(for ICache)
    v0 <- 0   //(for DCache)
  v1 <- !spill;  //v1: If the 2nd part of request has valid data.
          // For unspilled requests, the 2nd part doesn't exist,
          // & v1 is set to 1.
  Addr <- request address.
  Spill_location <- Bytes taken from 2nd request, 0 <- no spill,
                          LSBs of the address <- cache spill
  Data <- 32-bit(0);     //Reset the register data.
  Data <- shift&insert(read_data, spill\_reg); //For ICache hit.
}
On (mem_vld | cache_hit){
        spill_reg.data <- shift&insert(input data,spill_reg);
}
On (data return to cache){
        V0 <- 0; V1 <- 0; //Resets the control bits.
}
```

2.9.2 D Cache

The DCache is a 3 way set associative cache. It has 32 cachelines, with the size of each cacheline being 8B long. Just like the ICache, it is a physically indexed, physically tagged cache. It uses a LRU replacement scheme, and is a write-allocate cache. This is shown in Fig. A.13 (Pg. 56)

The DCache is a non-blocking cache for reads, supporting upto 4 outstanding read requests. This means that the request from the pipeline is first inserted into the MSHR before actually making the request to the cache storage, thus making it a

multi-cycle access even in the case of a cache hit. To cover this latency, the request to the DCache is made in the TLB-pipe stage, and the returned data is read in the next stage (mem-pipe). This effectively covers the cache latency and thus a cache hit is a one cycle access from the pipeline's perspective.

The DCache is a blocking cache for writes. However, this does not present a problem due to two reasons. First, every piece of memory that is written to is also read in the mem-pipe stage, even when it is not required. This prefetches the cacheline for a write, ensuring that all writes hit in the cache. Second, there is a write buffer present in the writeback stage that buffers all memory writes, thus covering up any latency incurred while writing the data back into the cache. The combination of these two factors ensures that the DCache can block on writes without hurting performance. The main pieces of logic in the DCache are explained below.

MSHR

The MSHR consists of 4 entries. The entries are structured like the storage in the ICache (i.e. v0,v1,address,spill_loc,data). Each entry can thus result in upto 2 reads, depending on whether the request spills across cachelines. So, each entry is considered as having two "parts" (referred to as entry_part), with each part accounting for 1 read request. It is indexed using 4 pointers:

- Wrptr[2:0]: Denotes the entry into which the next read request from TLB-pipe is stored. An entry is stored if dcache_rdvld & !mshr_full. Wrptr is incremented by 2 every time a new entry is stored into the mshr.
- Rdptr[2:0]: Denotes the entry which is to be read off from the MSHR, and its data returned to the pipeline. The read data is valid when (v0 & v1 == 1) for the entry indexed by readptr. It is incremented by 2 every time a valid data is read off by the pipeline.
- Reqptr[2:0]: Denotes the entry, entry_part that is to be requested to the cache storage. Its increment logic is shown below.
- Memptr[2:0]: Denotes the entry,entry_part into which the returned memory data is to be written. Its increment logic is shown below.

The read/returned data is shifted and inserted into the MSHR using the same logic as was used in the ICache. The pseudocode detailing the update policies for the MSHR is shown below.

```
//DCache is accessed with i_rdvld,i_address
//DCache returns data with o_datavld,o_address,o_data.
//The array vld = {{v1[3],v0[3]},{v1[2],v0[2]},
//                 {v1[1],v0[1]},{v1[0],v0[0]}}
if(i_rdvld & (~mshr_full)){
  //MSHR contents: {v0,v1,spill_loc,address,data}.
  mshr[wrptr[2:1]] <- {1'b0,~spill,spill_loc,i_address,32'b0};
  wrptr += 2;
}
if(v0[rdptr[2:1]] && v1[rdptr[2:1]]){
  o_datavld <- 1;
  o_address <- mshr[rdptr[2:1]].address;
  o_data <- mshr[rdptr[2:1]].data;
  rdptr += 2;
}
if(vld[reqptr] == 0){
  ca_vld <- 1;      //Signals for accessing Dcache store.
  ca_addr <- mshr[reqptr[2:1]].addr
  reqptr <- (~vld[reqptr+1]) ? reqptr + 1: reqptr + 2;
  shift&insert(ca_data,mshr[reqptr[2:1]].data);
  if(hit){
    mshr[reqptr[2:1]].data <-
              shift&insert(ca_data,mshr[reqptr[2:1]].data);
  }
}
if(mem_vld){
  mshr[memptr[2:1]].data <- s
            shift&insert(mem_data,mshr[memptr[2:1]].data);
  //Shifts memptr to the next invalid (aka miss) location
  memptr <- find_next_zero(vld);
}
if(memptr == reqptr && hit){
  //This ensures that memptr never points to a location
  //that alrady has valid data.
  Memptr <- updated reqptr
}
```

Storage for write request

If the write spills across a cacheline boundary, the signals for the 2nd part of the write are stored in this register. The register contents are valid,spilled_size,address,data. They are calculated as for the ICache. So, the write request to the cache storage is either taken directly from the input data from WB, or from this register.

Bank read & write logic

The DCache data & tag stores are made of ram8b8w$ cells. Since our cache has 32 entries, this naturally splits it up into multiple banks. In each cycle, upto 3 things - a cacheline read, a cacheline writeback, and a memory write can happen in the cache. The bank read & write logic arbitrates between the requests and sends them to the appropriate banks. The pseudocode of this piece of logic is shown below.

```
b_rd <- 0; b_memwr <- 0; b_wbwr <- 0;
rd_bank <- rdaddr[7:6];
mem_bank <- memaddr[7:6];
wb_bank <- wbaddr[7:6];
for each bank{
  if(more than one operation tries to take place in bank)
  Arbitrate according to priority (highest to lowest) :
        -writeback
        -memory write
        -read
  Set bits in b_rd, b_memwr,b_wbwr according to the
                          operation taking place in the bank.
  Send the required addresses & data to the banks.
}
if (evict) { //Signal generated by control bits update logic
  Select the correct data according to the bank & way evicted.
  Send out a store request to the LSQ.
}
if(read operation takes place on a bank){
  if(hit){
    Select the correct data according to hit_way.
    Send to the MSHR.
  }
```

```
  else{
    Send a load request to the LSQ with the read address.
  }
}
```

Control bits update logic

This piece of logic reads off the valid, dirty, and LRU bits, and updates them according to the operations taking place in the bank in the current cycle. It also decides which (if any) cacheline needs to be evicted. The pseudocode for this piece of logic is shown below.

```
//Receives b_rd,b_memwr,b_wbwr from the bank read&write logic.
for each bank{
  Generate the lru_way based on the address given to the bank.
  if(b_wbwr[bank]){
    dirty[wbaddr[7:3]][hit_way] <- 1;
  }
  if(b_memwr[bank]){
    valid[memaddr[7:3]][lru_way] <- 1;
    dirty[memaddr[7:3]][lru_way] <- 0;
  }
  accessed_way = (b_memwr[bank] ? lru_way : hit_way;
  if (b_rd[bank] | b_memwr[bank] | b_wbwr[bank]){
  //some operation takes place in bank.
  Update lru bits using accessed_way.
  }
}
```

2.9.3 Load Store Buffers

This unit buffers the load and store requests from the caches, and coalesces the return data across multiple packets to construct the returned cacheline.

LSQ

This is a pure queue which contains the outgoing memory transactions. It stores the address (15b), the data source (1b), read/write (1b), and the data (64b). Entries are

inserted into the queue whenever a valid mem load/store request is asserted from either of the caches. If the head of the queue has a valid entry present, it is converted into bus transactions and sent to the bus on receiving a grant. This is shown in Fig. A.23 (Pg. 64).

Due to lack of time, the LSQ does not currently support data forwarding from stores to loads.

ICache- & DCache- collect

The collect modules coalesce the returned data across multiple packets of a transaction to form a complete cacheline that is then written into the cacheline. The ICache collect module coalesces data across 8 cycles, thus storing 1 complete ICache cacheline (32B). The DCache collect module can store upto 4 DCache cachelines' worth of data, thus also storing 32B of returned data from memory.

Both of these modules can be extended to support the data rearrangement required for implementing Critical Word First. Again, due to a lack of time, this feature was not implemented in the current design.

2.9.4 Bus

The bus consists of 64 lines, split into 32 data & 32 control lines. The fields of the bus packet are listed below:

- Bus[31:0] : Data (Contains the read/write data, or the interrupt vector for transactions that interrupt the processor).
- Bus[46:32]: Address (Request PA for memory reads/writes).
- Bus[49:47]: Size (Write size for memory writes. Encoded as write size (in bytes) - 1).
- Bus[50:50]: Unused
- Bus[51:51]: _1st (Denotes the first packet of the current transaction).
- Bus[52:52]: Cacheable (Denotes whether the data belongs to a cacheable page. Used by the DCache for bypassing its update machanism.

- Bus[54:53]: rdwr (Needed to support two different read widths in memory. Encoding: 00 - DCache read, 01-ICache read, 1x-Write).
- Bus[56:54]: dst (destination of the transaction. Encoding: 00 - I/O1, 01-I/O2, 10- Processor, 11- Memory).
- Bus[58:57]: src (source of the transaction. Same encoding as above).
- Bus[59:59]: valid (Denotes a valid transaction).
- Bus[62:60]: reqCycles (number of cycles for which the bus is requested, equal to the number of packets in the transaction).
- Bus[63:63]: interrupt (the current transaction interrupts the processor).

The bus is a pseudo split transaction bus. It is split in the sense that the data request and response are decoupled from each other, but not a full split transaction bus as a multi-packet transaction is given consecutive cycles and hence cannot be split. Bus design details are provided in Fig. A.10 (Pg. 53).

2.10 Main Memory

2.10.1 Memory request buffer

Memory read & write requests from the units are inserted into a request buffer. The write requests are coalesced over multiple (at most 2, in our case) bus cycles and inserted into the buffer. Entries are read from the buffer and sent to the banks for read/write when the required bank is idle. Entries are removed from the buffer when the read data is sent over the bus (for reads), or when the write is complete (for writes). This is shown in Fig. A.14 (Pg. 59)

2.10.2 Banked structure

The main memory is constructed out of sram8b128w$ cells. This allows us to perform writes at a 1B granularity without having to deal with the complications of partial cell writes. The bank width is set to 8B, thus ensuring that a DCache read/write cannot span across banks. These two decisions lead to a simple banked structure with 4 banks. The mapping of the address bits is shown below:

- Addr[2:0]: chip/cell no. within bank

- Addr[4:3]: bank number
- Address[11:5]: Row number within cell
- Address[14:12]: Page number

2.10.3 Memory controller

The memory controller sends read & write requests to the banks, detects "done" conditions for each bank, reads off the data, and sends it to the memory request buffer. To do that, it first associates a counter with each bank. The bank counter starts off when a memory read or write request is sent to the corresponding bank. It then increments every cycle (processor cycle) till it reaches a max. Value, and then drops down to 0. Thus a count of zero denotes an idle bank.

The complete sequence of operations taking place on a memory read/write are:

- The bank for the request at memory_buffer[reqptr] is polled every cycle till its counter becomes 0. If the request is an ICache read request, we wait till the counters for all banks become 0 (as ICache reads span across all the banks).
- The max value for the bank counter(s) is set to 6 (for mem read) or 10 (for mem write).
- The counters count up every cycle till the max value is reached. Once that happens, they assert a done signal. If the request is a write request, no further action needs to be taken, and the counter drops to 0, denoting that the corresponding bank is idle. Reads need to wait at cnt_max till the read data is sent out.
- In parallel, requests already finished are removed from the memory_buffer, till the rdptr reaches our desired entry.
- The data is read off the desired bank (or all the banks, in order, for an ICache read), converted into packets & sent on the bus.
- The read counter(s) drops to zero once the data is transmitted, denoting that the bank is again able to accept a new request.

NOTE: We said above that no further action (like sending an acknowledgement) needs to be taken on a memory write. This is because our processor is in-order, and

there is no loss of transactions in our system. Hence, all writes can be safely assumed to take place in the exact order sent by the processor, and do not need an acknowledgement for reordering or resending data.

Chapter 3

Design Trade-offs

As described in the Introduction, the design was envisioned to be optimized for IPC while giving careful considerations to cycle time. The general Goals and Priorities for our design have been described here.

Goals:

1. Correct execution on the provided subset of instructions.
2. Increasing IPC while remaining in-order.
3. Balancing memory latency with pipeline depth to maximize performance.

Priorities:

1. Improving In-Order IPC numbers.
2. Reduction of Cycle Time.
3. Minimizing Area & Power - by constraining on minimal use of resources.

Based on the above mentioned list of considerations in mind, there were a lot of design trade-offs made before the implementation which are described below.

Trade Offs:

1. The pipeline is In-order instead of out-of-order. This is because of the reduced design & verification complexity of the in-order pipeline, which made it feasible to implement it in the limited time frame for this project.

2. Our pipeline has 9 stages, because it provides a good balance between cycle time & complexity. Decode is split into two stages (one stage (D1) acting like a pre-decode stage, and the other doing the actual decode). This is done as size calculation has quite a high latency.

3. We forward all register data from writeback. This saves 1 cycle, at the cost of extra forwarding logic.

4. We stall on memory dependencies, and do not forward memory data from any stage. This was done because of the relatively high complexity of forwarding specific bytes from one stage to another.

5. All memory write locations are first read, thus prefetching the cache with the equivalent of a 'touch' operation. This permits early detection of exceptions, and provides an easy way of implementing a write allocate cache. On the other hand, it may stall the pipeline in mem-pipe, 2 stages before the stall would be otherwise required.

6. Both loads & stores are on the critical path. This was done so as to permit the use of a unified load-store queue, and reduce some control complexity in the Dcache controller.

7. Microinstruction size: Microinstructions were created for instructions based off structural constraints of the design: primarily the number of read and write ports to the registers and the data cache.

8. Branch predictor: Adding a branch predictor increases the critical path of fetch, and consumes design & verification time. However, we decided that adding a branch predictor was likely to give a good performance boost, and hence decided to include it in our design. As for the structure, we anticipated that the test cases would not feature too many nested branches. Thus the BHSR implemented was quite small (3 bits of history) to ensure that the predictor warms up faster.

9. Early dependency resolution for stack operations: Stack operations frequently tend to happen close to each other, for example, on context switch & restore. We decided to resolve the stack address dependency early to avoid a (at least) 4 cycle stall between stack operations.

10. SIMD Operations : All SIMD Operations are done over two cycles. This decision was made on the basis that D-cache bus length is 4 bytes wide. With a 4-byte bus

from processor to D-cache, any SIMD instruction using memory would need to fetch the information over two cycles. Hence we prioritized on resource utilization, deciding against using extra adders for 64-bit information processing in favor of making the Exec stage process 32-bits of information over two cycles.

11. Instruction buffer size: The fetch stage gets 8 bytes from the instruction cache. Given this design decision, the instruction buffer was configured to hold 3 such sets of fetch bytes, which allows for local caching of around 3-4 instructions (based on the average x86 instruction length) since the D-cache is given priority for accesses to memory over the I-cache.

12. Instruction Fetch Size : The Fetch stage brings in 8 bytes from I-cache on each access. This decision was taken after considering the average size of instructions in the given subset of the ISA. 8 was choses since we wanted a balanced design, i.e. fetching one instruction since we can retire at most one instruction each cycle.

13. Control Store: The choice of which control bits come from the control store, and which are generated by combinational logic was made based on inspection of the signals needed by the datapath. Most instruction specific signals were added to the control store as each instruction has its own control store entry. Instruction variants that were fairly distinct were split into their own control store entries (addressed by both the opcode and the modrm byte). Certain bits were "overridden" by combinational logic if the logic was universal to all instructions. Final, control bits that are instruction opcode independent (SIB for example) were generated via combinational logic.

3.1 Special Cases

3.1.1 Pipeline Invalidation

Interrupt

Decode When an interrupt is detected, the interrupt signal and vector are latched registers at DEC2. DEC2 then stalls until the pipe is flushed (indicated by the pipe clean signal). When this is completed, the interrupt is acknowledged and the interrupt registers cleared. The interrupt is then processed in exactly the same way as an exception, with a different opcode. This allows a different vector to be used in the

calculation of the address of the IDTR entry. The control store entry for interrupt handling differs from the exception handling entry only in the bits for the decode stage.

Data Path When an interrupt is seen, the Decode stage stops processing data till each of the datapath latches are drained. Once the pipeline is clean of any pending instruction, the interrupt processing starts.

3.1.2 Exception

In the current design, exceptions can occur at two stages, Fetch and TLB-Pipe stage. An exception in fetch stage ignores any fetched bytes from the I Cache and puts the appropriate Exception byte in the instruction buffer. Once Decode stage fetches this byte it starts processing exception micro-instructions. No datapath invalidation takes place in such a scenario.

Incase of an exception in TLB-Pipe stage, the corresponding GP (General Protection) or PF (Page Fault) exception are sent to Decode2 Stage for further processing of exception. An invalidation signal is also sent to Register Read and Address Generation Stages to invalidate any valid processing of information going on in those stages. TLB-Pipe also invalidates its own current execution and marks the d-cache access invalid too.

For scenarios where a particular micro-instruction caused an exception while a previous micro-instruction has successfully passed the stage, we send an invalidation signal along with the current EIP of the instruction to Memory-Pipe and Execution Stages as well. In Mem-Pipe and Exec Stage, if the stage sees an invalidation request from TLB-Pipe, the EIP from TLB-Pipe is compared against the current instruction EIP being processed. If the EIPs match then the corresponding stage is also invalidated owing to the exception caused.

Note: Given the set of Instructions and the design criteria provided, we were sure that there could never arise a situation where a particular micro-instruction generating an exception in TLB-pipe could have a previous micro-instruction which had already reached WB stage. This gave us freedom to not have complex logic to invalidate instruction data from WB stage.

Within Decode2, upon seeing an exception signal, the current instruction being latched into the RR latch is invalidated. The bytes being latched into the Decode2 latch are multiplexed with Interrupt/Exception bytes, and the EIP is multiplexed with the Exception EIP. These are picked, and the valid_in is forced to 1, which results in an exception byte with the EIP of the excepting instruction being loaded into the instruction buffer. Then is then processed as any other instruction and the exception uops are read from the control store.

3.1.3 Branch Mis-prediction

Exec Stage Branch Mis-prediction is detected in Exec Stage as explained in Section 4.7.3. In case of a mis-prediction, an invalidation signal is sent to all previous stages. Every Stage from Register Read till Mem-Pipe is invalidated and the actual branch direction with target is also sent to the Branch Predictor. If it is a RET Imm instruction where stack needs to be popped, then a signal is sent along with the invalidation signal. This signal is checked by every previous stage which might contain the final micro-instruction for these kind of instruction to not invalidate itself. In case of Far jumps, far calls, far returns, the invalidation is not done here but is delayed till Write Back stage. More on this is explained in the section 3.1.4.

Fetch Stage The invalidation signal is coupled with other global invalidates from the datapath and follows that mechanism.

In addition to the global invalidate, a taken prediction requires invalidation of the trailing bytes in the same fetch width as that of the branch since the instructions corresponding to these bytes should not be executed. To achieve this, each valid fetch cycle is associated with a 4-bit identifier. Upon encountering a branch instruction at DEC2 and a taken prediction, DEC2 sends a cleanup signal to the instruction buffer, along with the fetch id of the branch. Instead of using the decoded size, the instruction buffer instead uses the number of bytes in the same fetch id that match in the instruction buffer as the size, and shifts these bytes out of the instruction buffer (the DEC1 valid is set to 0).

Inexact matching of the EIP for prediction lead to another problem: a prediction may arrive corresponding to a fetch width that that does not contain a branch (the branch would have arrived in the prior fetch ids). The prediction then cannot be evaluated at the EXEC stage. Note that this problem occurs only when the prediction

is taken. To prevent this, every time a taken prediction is seen, the corresponding fetch ID and EIP are latched. These remained latched as long as the fetch id of the following instructions is the same and no branch instruction is seen. If the fetch id of the incoming instruction changes and no branch has been seen yet, a DEC2 invalidate signal is issued couple with the saved EIP, and fetch replays the instruction before which the taken prediction was seen.

3.1.4 Change in CS Value

Any change in Code Segment value is detected in Write back stage when the data is being written to Code Segment register. In such a case, all pipeline stages are invalidated since every stage has instructions from a wrong base segment. The correct Code Segment and EIP are sent to Fetch and decode to start fetching from the correct pointer starting from the next cycle.

The D-Flag is set and cleared at Decode 2, i.e. the retirement of STD and CLD flags are completed out of order. To maintain coherence the D-Flag value at Decode time is passed on with each instruction in the pipeline latches. Every invalidation also forwards the corresponding D-flag for that instruction to be overwritten in Decode 2 while processing the next valid instruction.

Interrupts have special register in decode2 stage. When an interrupt occurs, the interrupt register is filled in with a 1, and the interrupt vector register is filled in with the interrupt vector of the interrupting device. The latched interrupt goes through an AND gate, the other input which is the pipe clean signal (this is simply the NOR of all valid latches in the datapath). The interrupt signal is thus acknowledged when the pipe is flushed. After this, ,processing of the interrupt proceeds in the same way as that of an exception.

In the last uop of an interrupt/exception byte, the address of the corresponding vector in the IDTR is calculated. This is appended to the address field of the final instruction, which is an indirect JMP on a memory address.

Chapter 4

Performance Analysis

4.1 Test Statistics

The programs provided for verifying the correctness of the processor (tests 0 to 4) and its performance (5b, 5c, 5d) were used to benchmark our design. Given a cycle time of 10.4ns, Table 4.1 shows the execution time and IPC for these tests on the processor.

Discussion on performance of test cases

- Test 2,3,4: These tests featured a number of exceptions/calls and nested versions of these. Since our design did not feature a return stack, performance for these would be is expected to be bad (especially if a branch is predicted and fall into one of the special invalidation cases). Each JMP at the end of a CALL or RET results in invalidation of all the instructions in the pipeline since the Code Segment is changed.

	Instructions	Execution Time (ns)	Cycles	IPC
Testcase 0	23	2071.28	199	0.1155778894
Testcase 1	143	9562.88	920	0.1554347826
Testcase 2	49	6695.28	644	0.07608695652
Testcase 3	36	3124.68	300	0.12
Testcase 4	48	4070.88	391	0.1227621483
Testcase 5b	2628	87767.28	8439	0.3114113047
Testcase 5c	392	7968.68	766	0.5117493473
Testcase 5d	263	8352.88	803	0.3275217933

Table 4.1 Execution Times for Provided Testcases

- Test 0,1,2: Performance was limited primarily by the datapath. We gained on performance in cases where the register forwarding logic was put into use. However, since we were missing both aggressive register forwarding and forwarding of data read/written to the memory, the IPC drops significantly. Also, some operations (such as SIMD ones) were broken down into uops, where they could have been completed in a single cycle.
- Memory was not a bottleneck since it was banked and could serve requests quite well, as well as the D-cache. The only place where the design lost out was probably the arbitration policy for memory accesses by either cache. Also, increased bus widths between the processor and the cache would have ensure that datapath and decode do not starve in any situation (rather, this design catered to the average case).
- The predictor design was itself quite sufficient for the given tests: however a key element that was missing was early branch resolution. This would have made a fairly large impact on the performance numbers, especially for the cases where a large number of EIP-relative branches are seen.

Evaluations of Specific Components Tables 4.4 and 4.3 show the results for the evaluation of the branch predictor.

	Execution Time w/ BP (ns)	Execution Time w/o BP (ns)
Testcase 5b	87767.28	112009.68
Testcase 5c	7968.68	11504.28
Testcase 5d	8352.88	11306.48

Table 4.2 Execution Times with and without Branch Predictor

	Branch Instructions	Misses	Hits	Prediction Accuracy (%)
Testcase 5b	1000	383	617	61.7
Testcase 5c	63	6	57	86.36
Testcase 5d	64	6	58	90.63

Table 4.3 Branch Predictor Accuracy for Provided Testcases

The branch predictor provides a fair boost to performance for code that included a large number of branches. Among these, the accuracy of the predictor was quite high for single loops, and acceptable for nested loops.

4.2 Critical Path Analysis

The following table describes the stages the critical path spans, and the corresponding delays. The output of DEC1 feeds into register-array which has a setup time of 0.2ns, giving a total critical path timing of 10.4ns.

	D-Cache	MP	TP	AG	RR	DEC2	DEC1
Path		nor2 nor2 and2 inv1 and2	nor2 nor2 or2 and2	and2	or2 and2	nor2 mux4_sel mux2_in	or4
Delay	5.4ns	6.65ns	7.75ns	8.1ns	8.8ns	9.7ns	10.2ns

Table 4.4 Critical Path Analysis

Chapter 5

Conclusion

5.1 Discussion of Constraints

- Design & verification time: This was the primary constraint in our case, and most design decisions along with the previously mentioned considerations also factored time as a major trade-off which respect to design complexity vs performance.
- Bus width: This would essentially enable more data to be brought into the caches, and thus provide increased benefits if the width of the bus from the cache to the processor were to be increased.
- Cost: Not the most area effective design, since area was not a provided constraint. A fair number of combinational elements (such as the comparator arrays in decode) were replicated several times to improve the critical path.

5.2 Future Design Improvements

1. While we had all the parts to send multiple cache accesses in parallel, we could not integrate them together due to lack of verification time. Including this functionality to make full use of the memory banking & the non-blocking caches would really help performance on memory heavy programs.
2. Having separate queues for loads & stores. This takes the D-cache stores out of the critical path. Allowing forwarding from the store queue to the load queue is another optimization.
3. More aggressive forwarding. Currently, memory data is not forwarded (due to complexity), and register data is only forwarded from Write-Back, even if it

has the potential to be computed & forwarded much earlier. More bypasses & forwarding paths can be added.

4. Early branch resolution: Most unconditional branches can be resolved in the register read stage. Early resolution has a significant effect on branch predictor accuracy, and therefore on overall CPI. This would involve changes in DEC1 to include a precode element that solely detects branches and calculates the effective address if possible.

5. Branch predictor trade-offs: Further study should be conducted on the branch predictor tradeoffs (for eg. optimal length of the history register, PHT size, BTB size, history updated speculatively or after resolution etc.)

6. Better resource utilization: The current design has not been optimized for power or area. Careful attention to these two metrics can help us make better trade-offs on resource usage.

7. Out of order: Making the processor out-of-order instead of in-order gives a dramatic performance benefit.

References

Appendix A

Schematics of Design

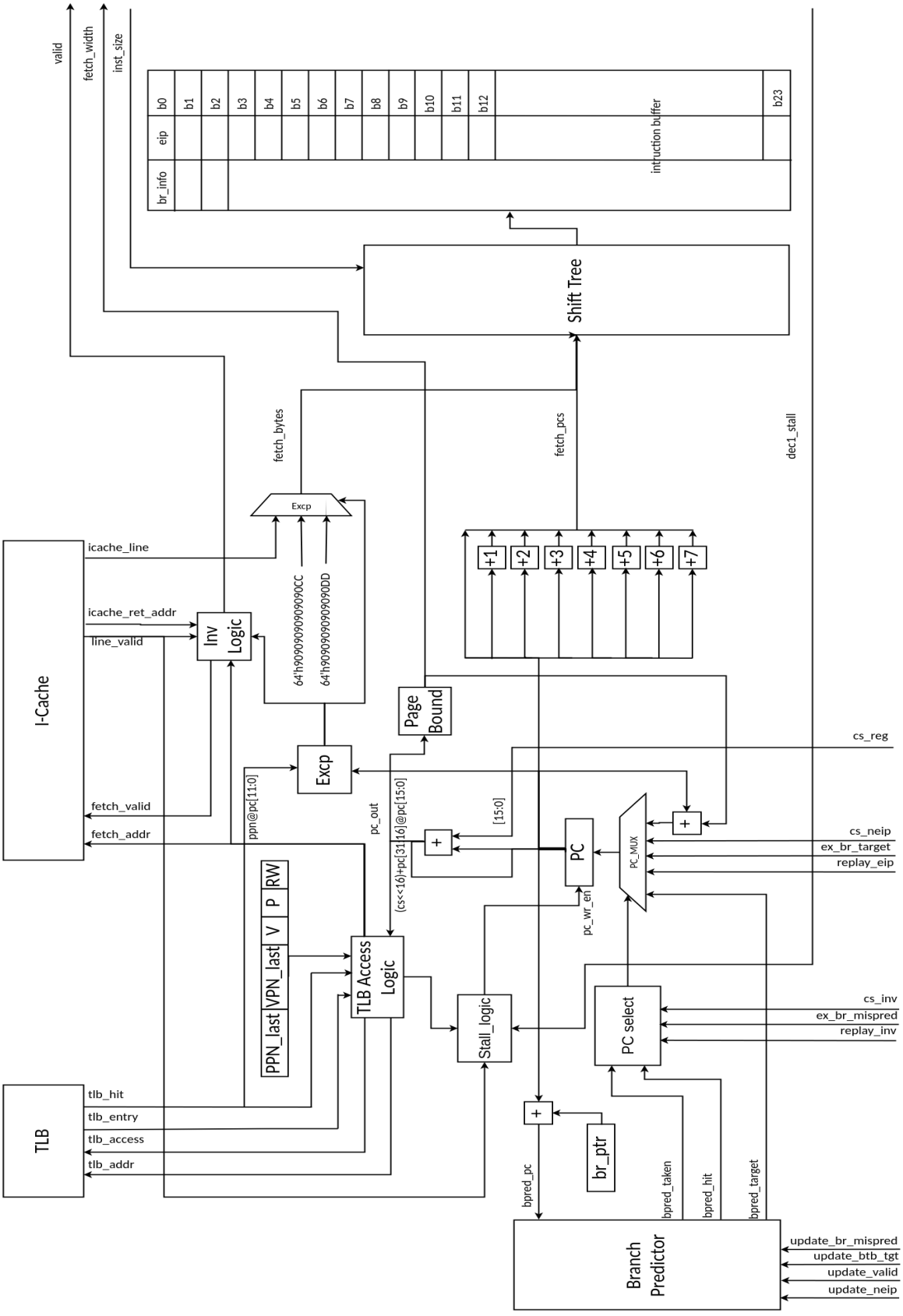

valid
fetch_width
inst_size
b0
b1
b2
b3
b4
b5
b6
b7
b8
b9
b10
b11
b12
b23
eip
br_info
intruction buffer
Shift Tree
fetch_bytes
fetch_pcs
dec1_stall
Excp
64'h9090909090900CC
64'h9090909090900DD
+1
+2
+3
+4
+5
+6
+7
icache_line
icache_ret_addr
line_valid
I-Cache
Inv
Logic
Page
Bound
Excp
cs_reg
fetch_valid
fetch_addr
ppn@pc[11:0]
pc_out
(cs<<16)+pc[31:16]@pc[15:0]
[15:0]
PC
PC_MUX
cs_neip
ex_br_target
replay_eip
pc_wr_en
PPN_last
VPN_last
V
P
RW
TLB Access
Logic
Stall_logic
PC select
cs_inv
ex_br_mispred
replay_inv
TLB
tlb_hit
tlb_entry
tlb_access
tlb_addr
br_ptr
bpred_pc
bpred_taken
bpred_hit
bpred_target
Branch
Predictor
update_br_mispred
update_btb_tgt
update_valid
update_neip

Fig. A.1 Fetch

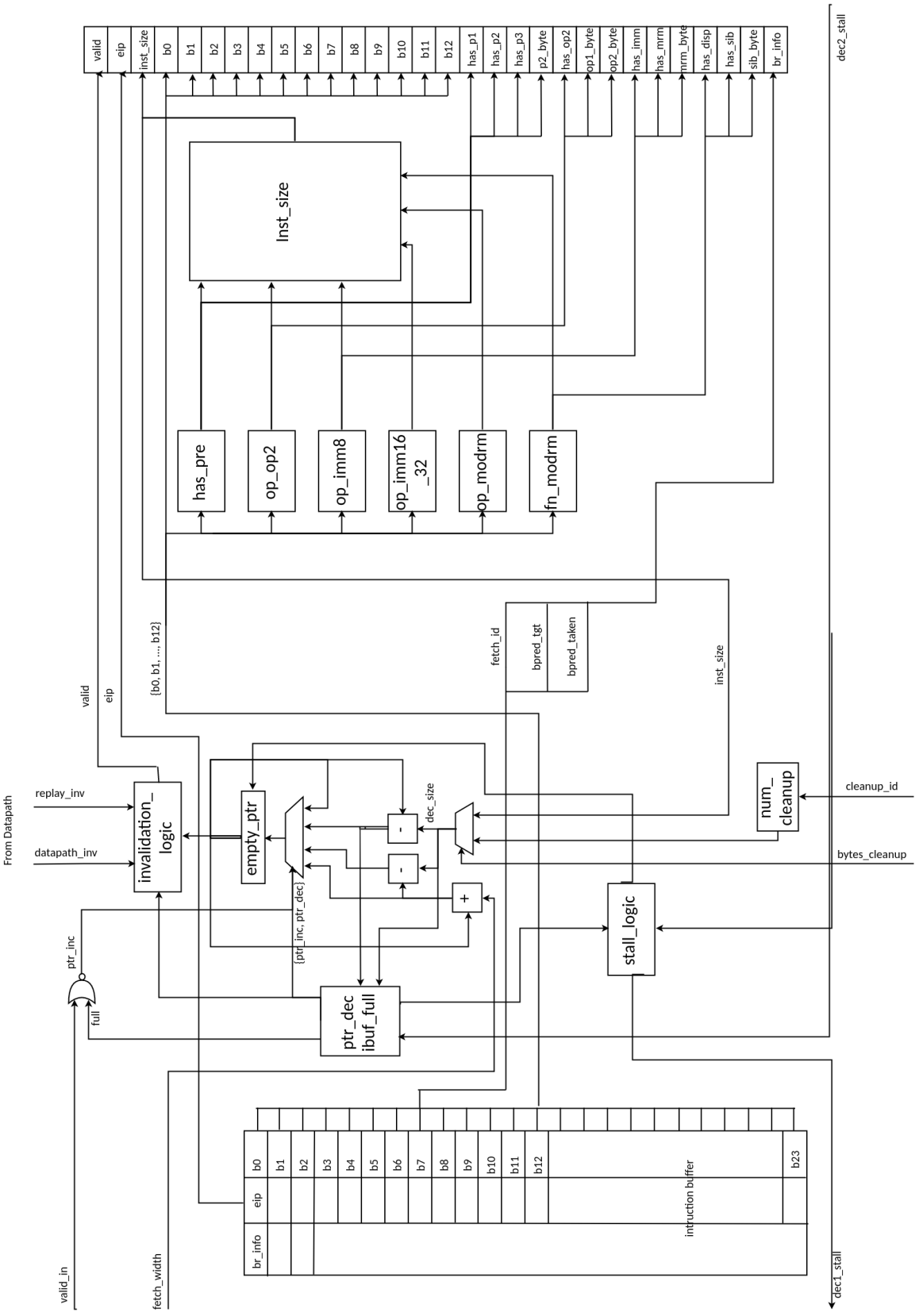


Fig. A.2 Decode 1

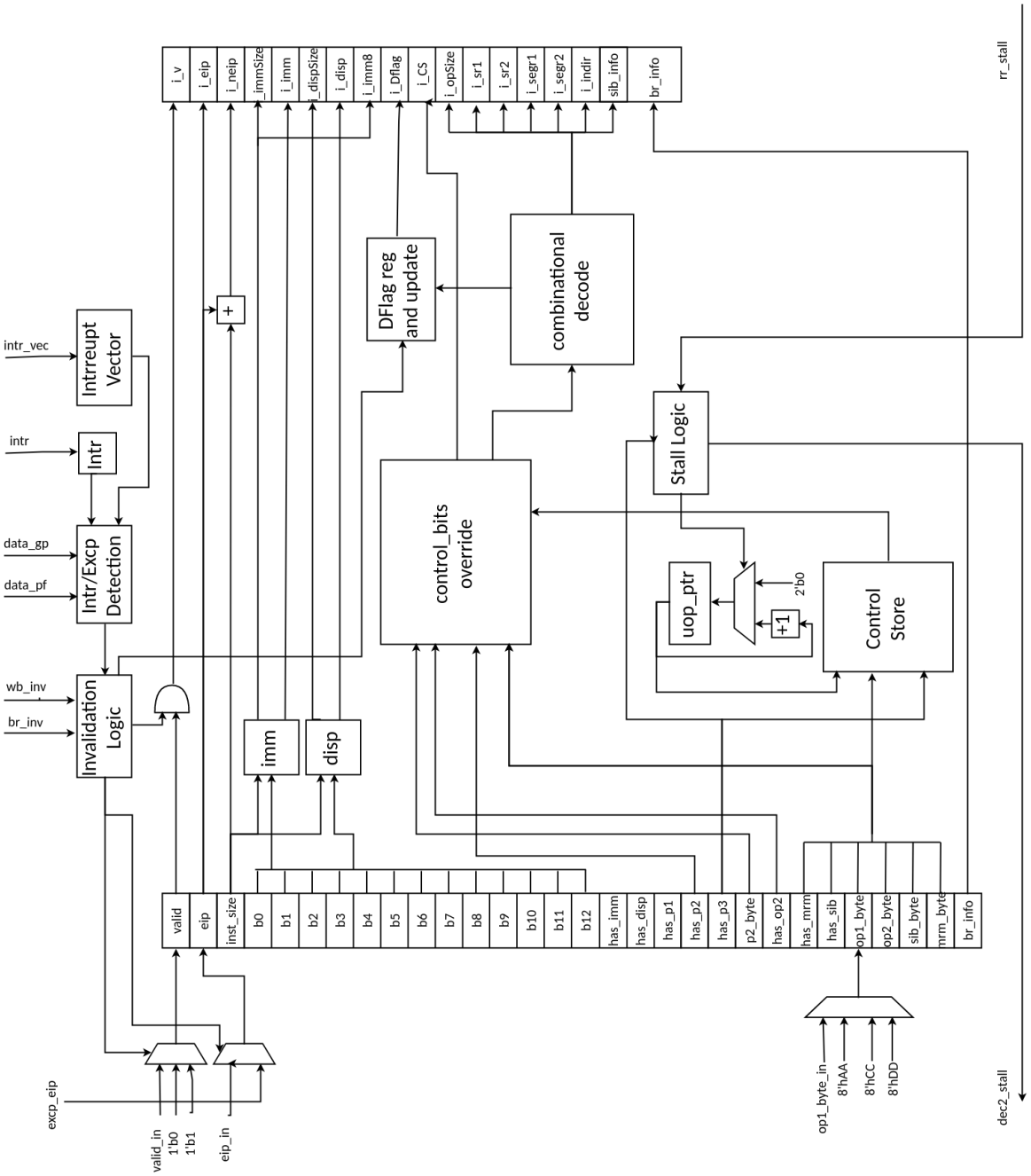


Fig. A.3 Decode 2

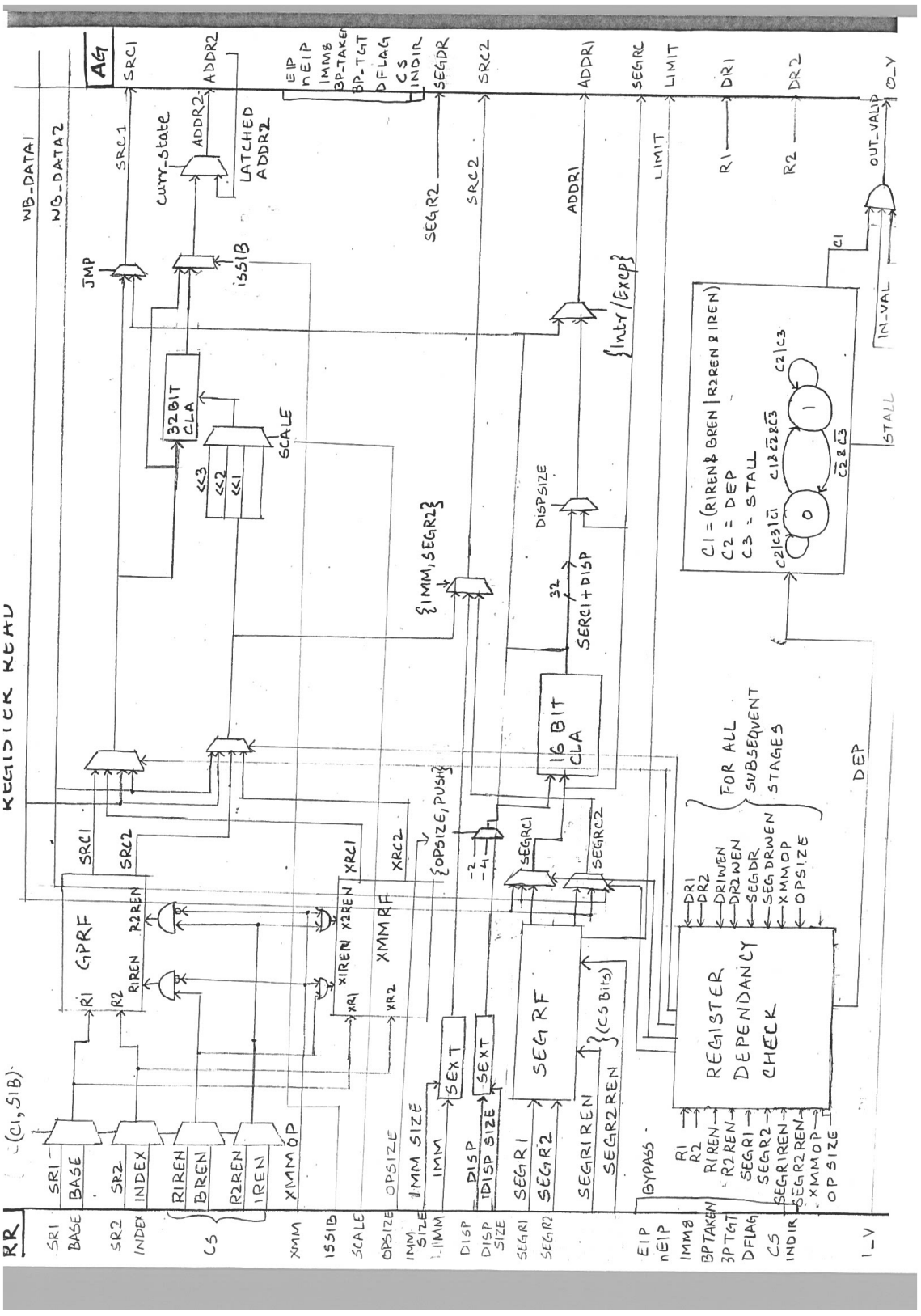


Fig. A.4 Register Read

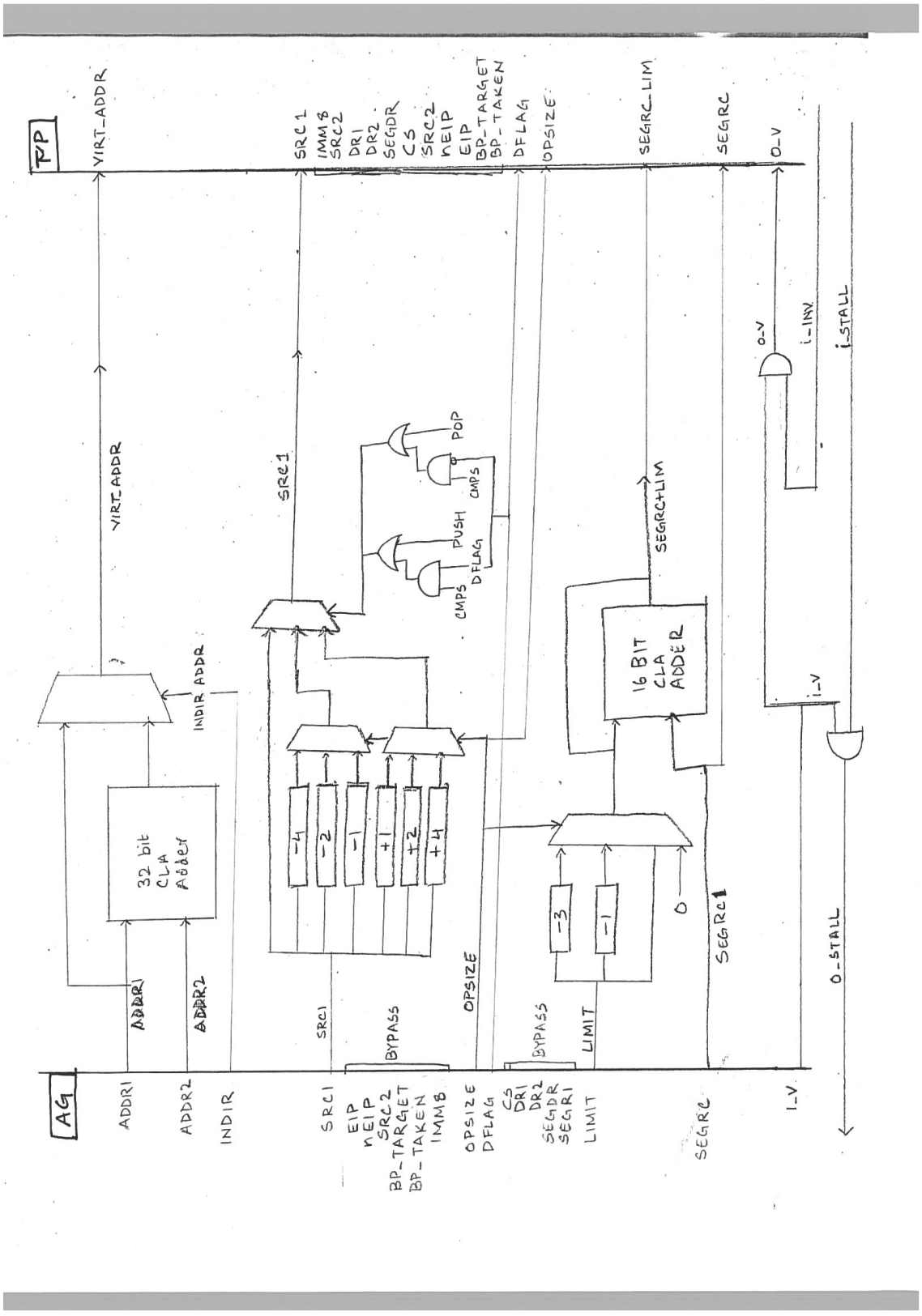


Fig. A.5 Address Generation

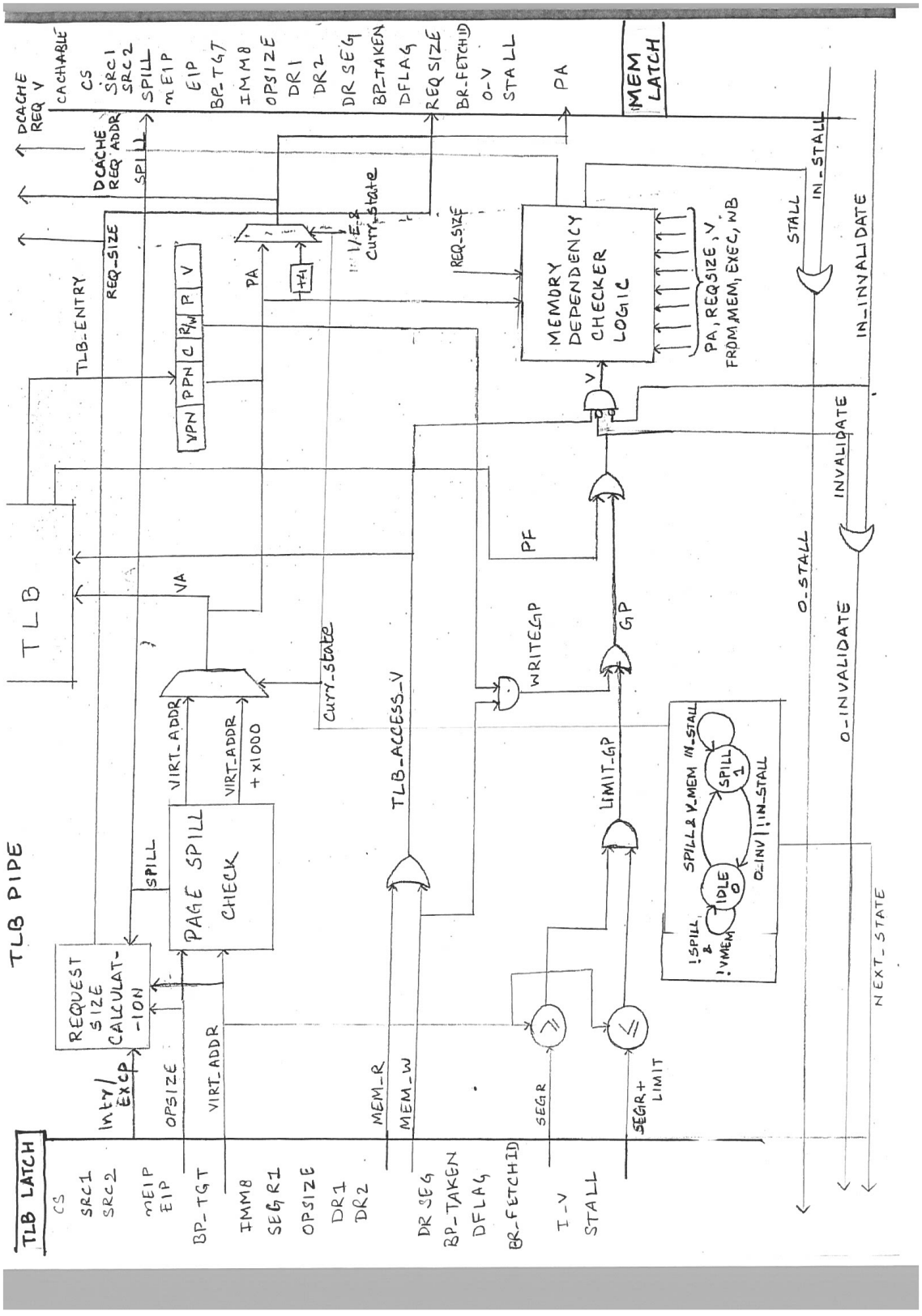


Fig. A.6 TLB Pipe

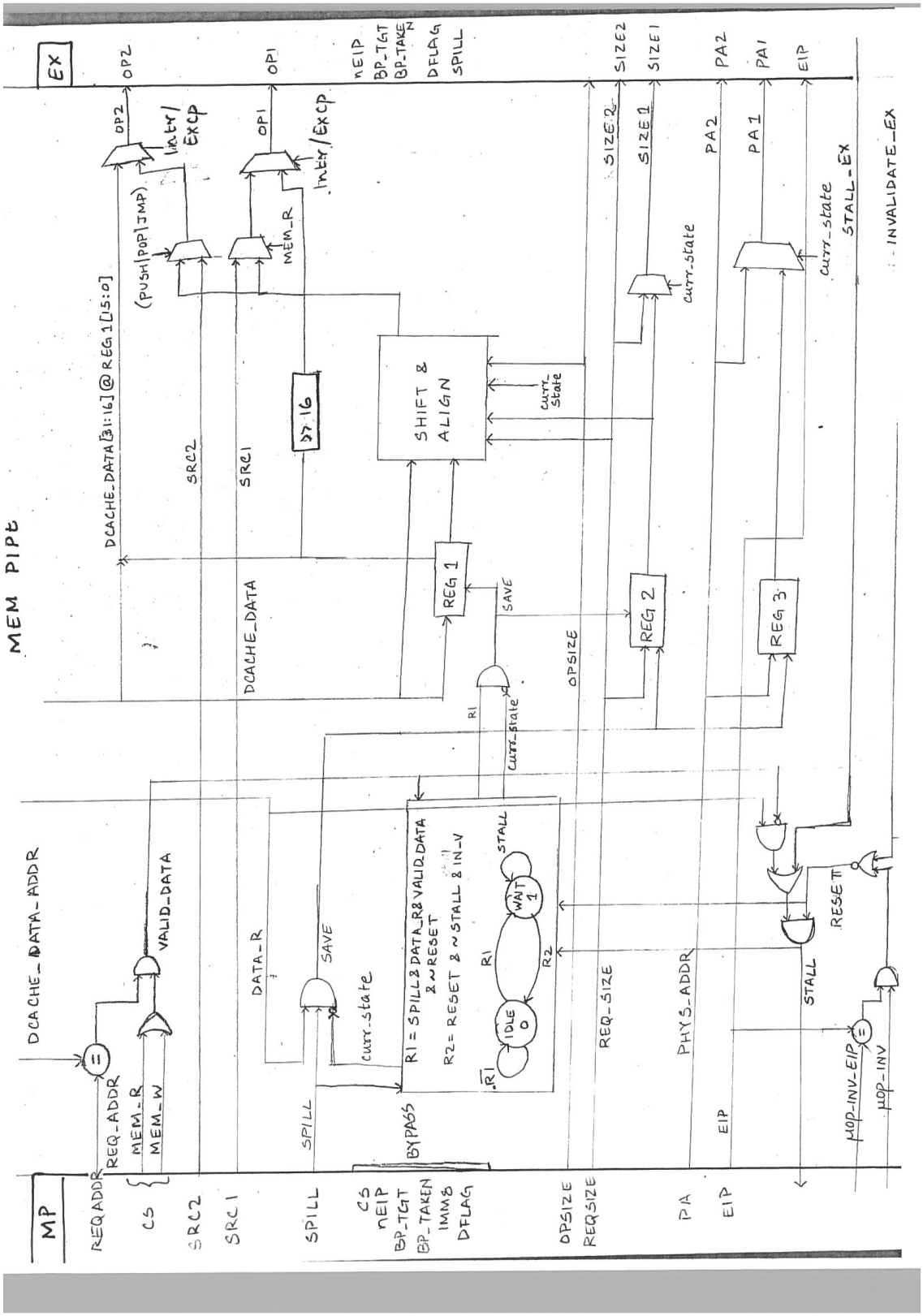


Fig. A.7 Memory Pipe

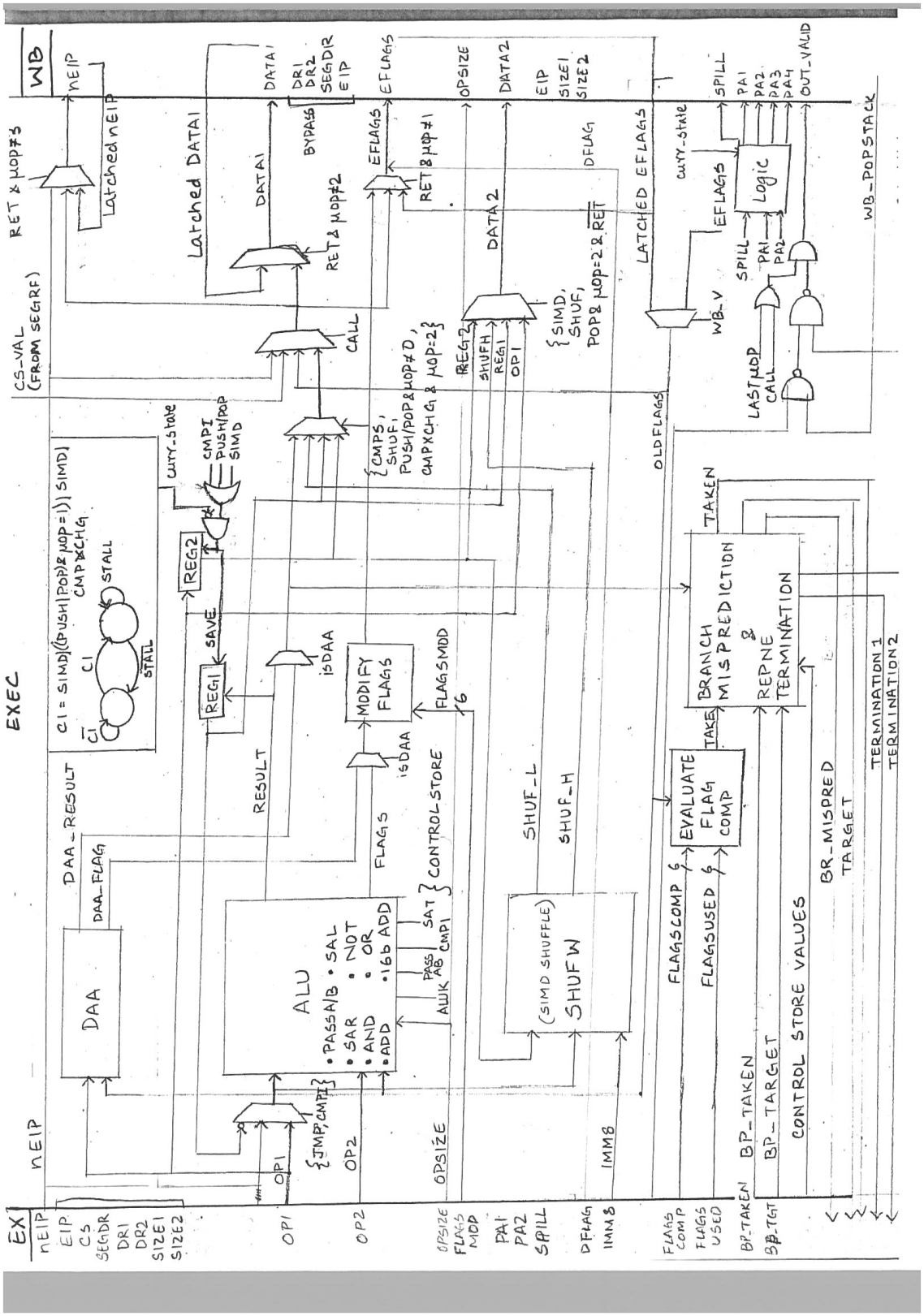


Fig. A.8 Execute

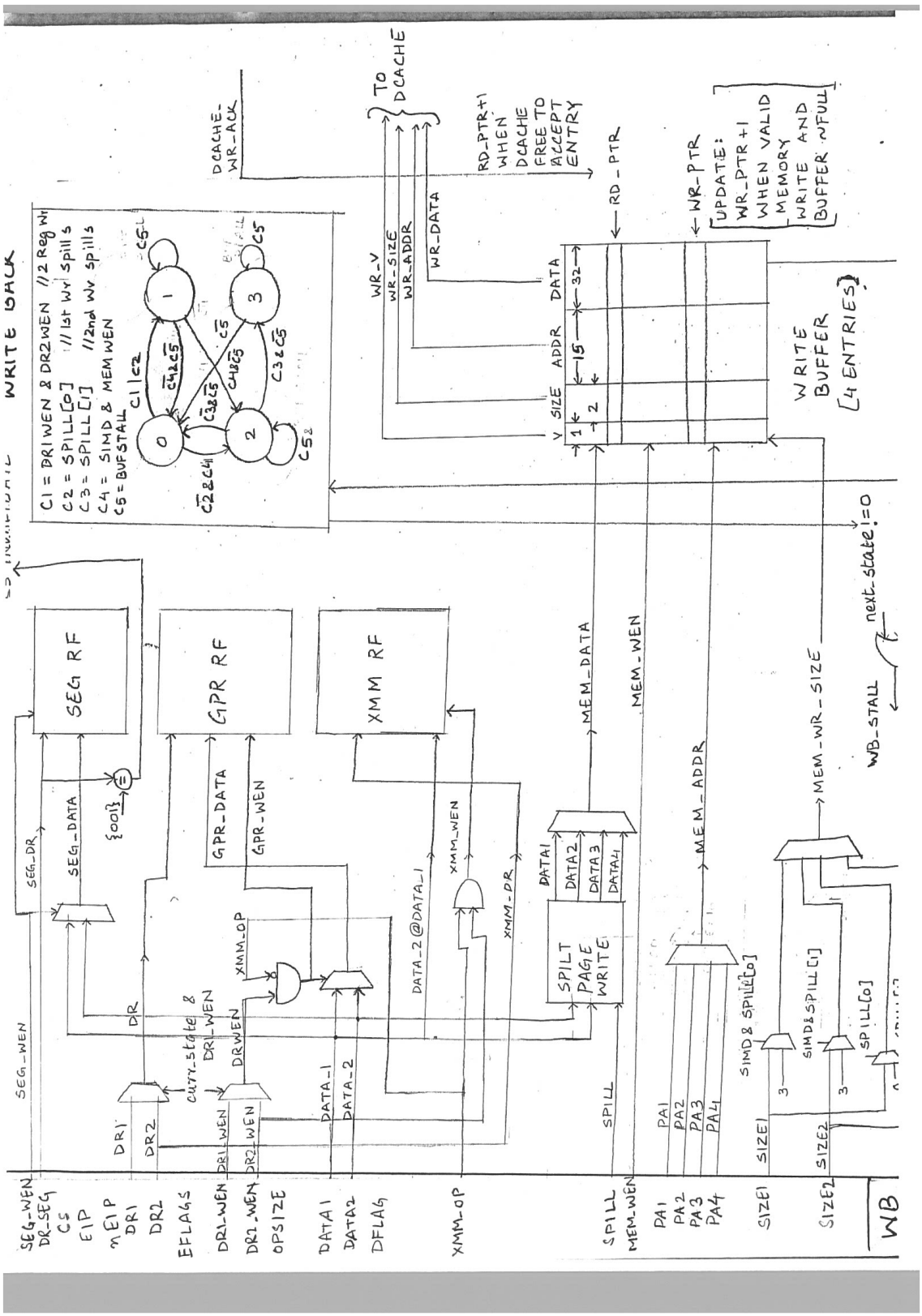
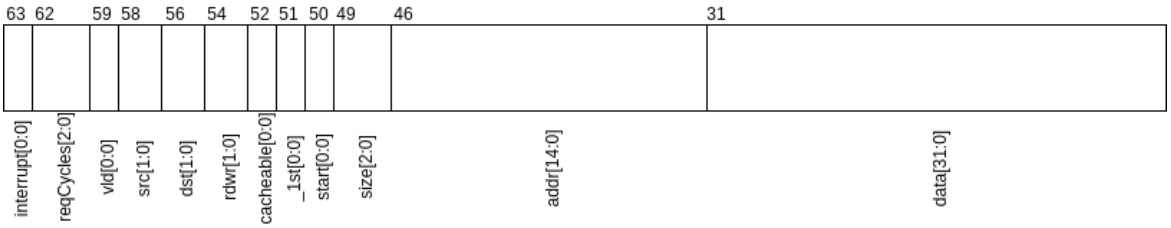


Fig. A.9 Write Back

BUS PACKET



Encodings:

src, dst -
00: I/O 0
01: I/O 1
10: Processor
11: Memory

size - (# bytes written) - 1

rdwr -
00: D-Cache read (8B)
01: I-Cache read (32B)
1X: Write (sizes 1B to 8B)

Other fields:
data [31:0] - Written / Read data or interrupt vector
addr[14:0] - Write / Read address
start[0:0] - Unused
_1st[0:0] - 1st transaction of the current burst
cacheable[0:0] - whether the current address is cacheable
vld[0:0] - is the current transaction valid
reqCycles[2:0] - # cycles for which grant is requested
interrupt[0:0] - This transaction interrupts the processor

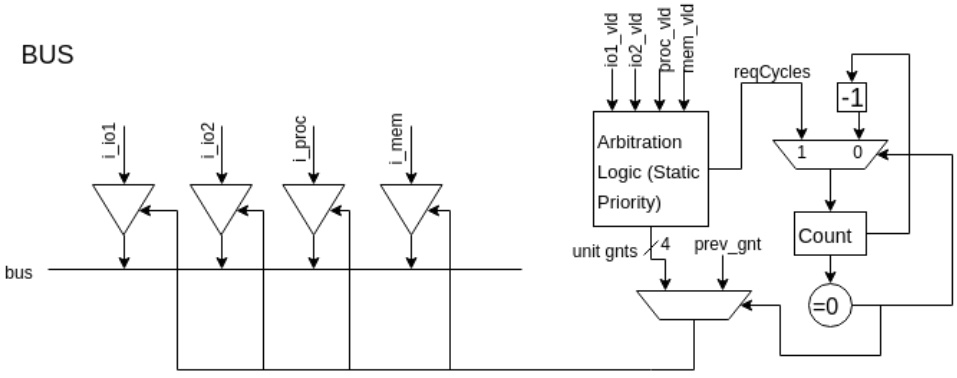


Fig. A.10 Bus Packet

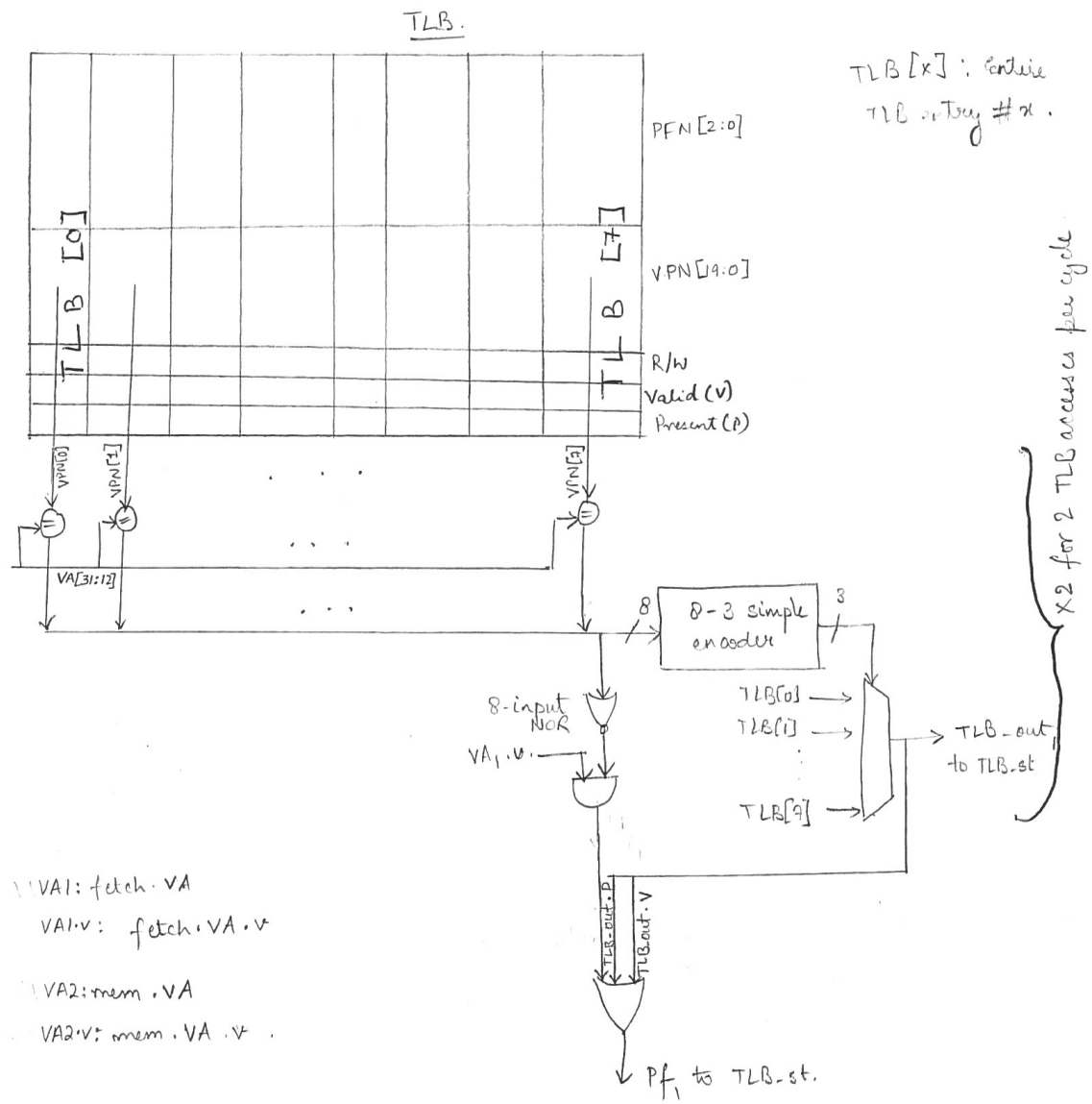

TLB.
TLB[x]: Entire TLB entry #x.
PFN[2:0]
VPN[9:0]
R/W
Valid (V)
Present (P)
TLB[0]
TLB[7]
VPN[0]
VPN[1]
VPN[9]
VA[31:12]
8
8-3 simple encoder
3
8-input NOR
VA1.V.
TLB[0]
TLB[1]
TLB[9]
TLB_out to TLB_st
X2 for 2 TLB accesses per cycle
VA1: fetch.VA
VA1.V: fetch.VA.V
VA2: mem.VA
VA2.V: mem.VA.V
TLB_out.P
TLB_out.V
Pf, to TLB_st.

Fig. A.11 TLB

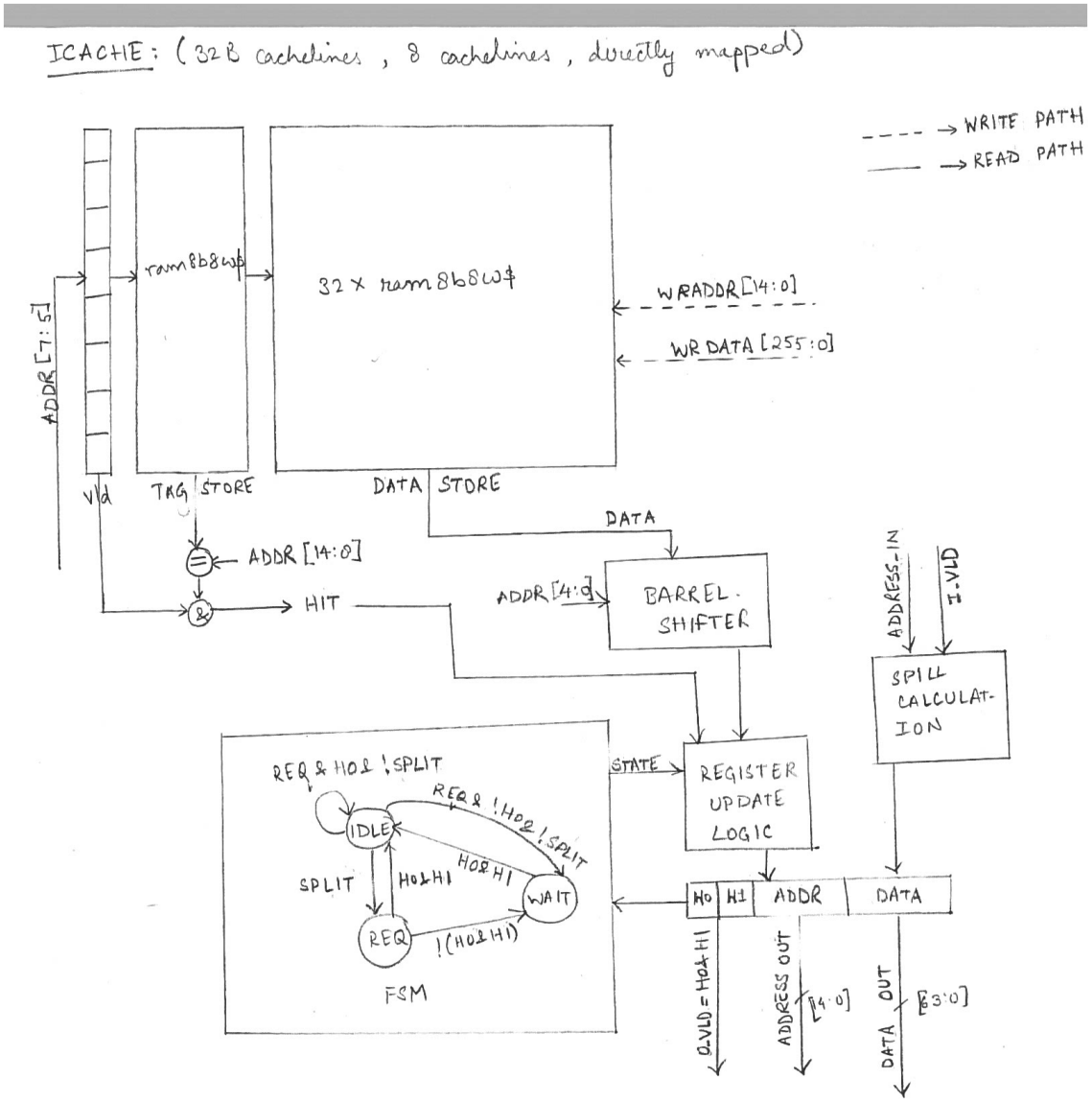


Fig. A.12 I-Cache Schematics

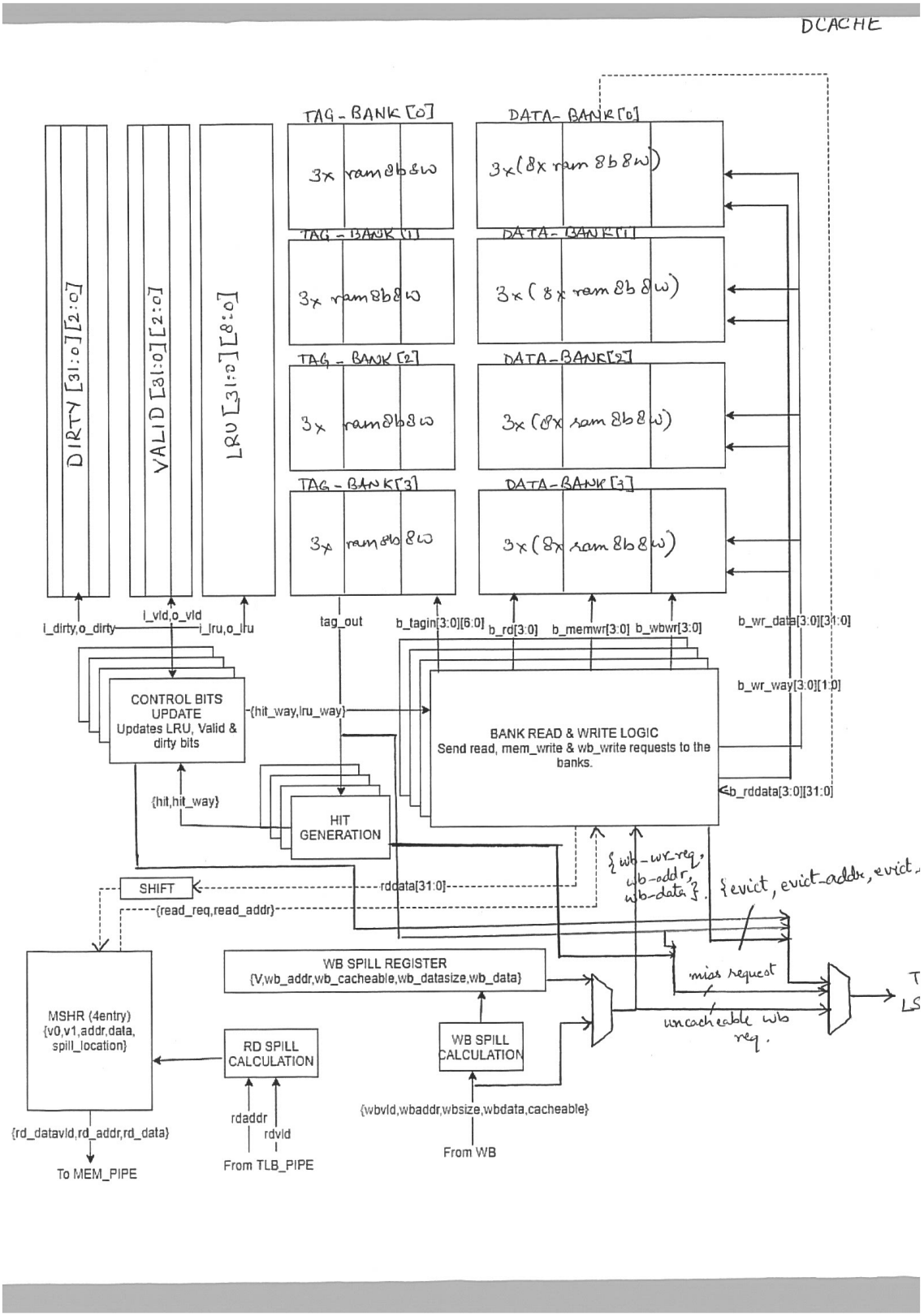
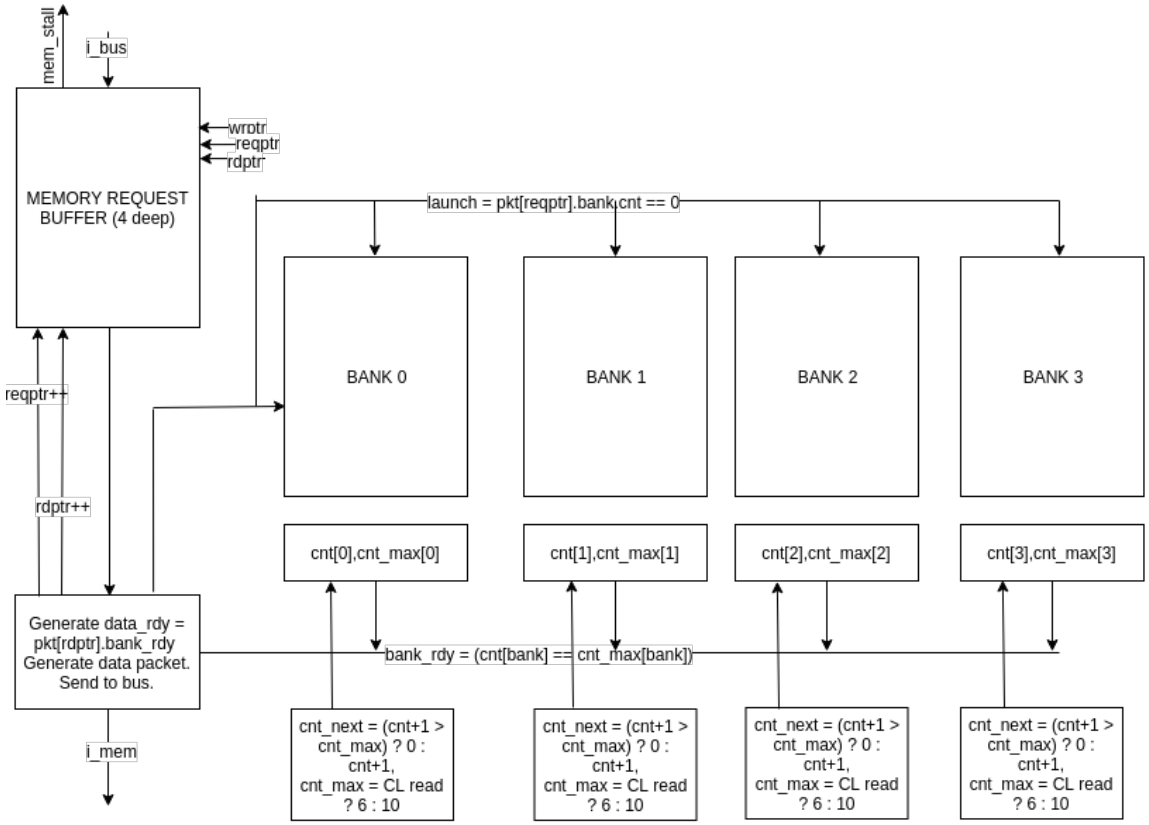


Fig. A.13 D-Cache Schematics



Address mapping: addr[2:0] : chip # within bank
addr[4:3]: bank #
addr[11:5] : row # in chip
addr[14:12]; page #

Memory is constructed of sram8b128w$ parts.
Each bank is 8B wide, and consists of a 8x1 grid of chips
Each page consists of a 32x1 grid of chips..

For ICache read: launch = (cnt[0] == 0 && cnt[1] == 0 && cnt[2] == 0 && cnt[3] == 0).
bank_rdy = (cnt[0] == 6) (All banks are ready at the same time).
For DCache read/write: launch = (pkt.reqptr.bank.cnt == 0)
For read, bank_rdy = (bank.cnt == 6)
For write, bank_rdy = (bank_cnt == a)
So, memory latency = 6 processor cycles (for read), 10 procecssor cycles (for write).

Fig. A.14 Main Memory

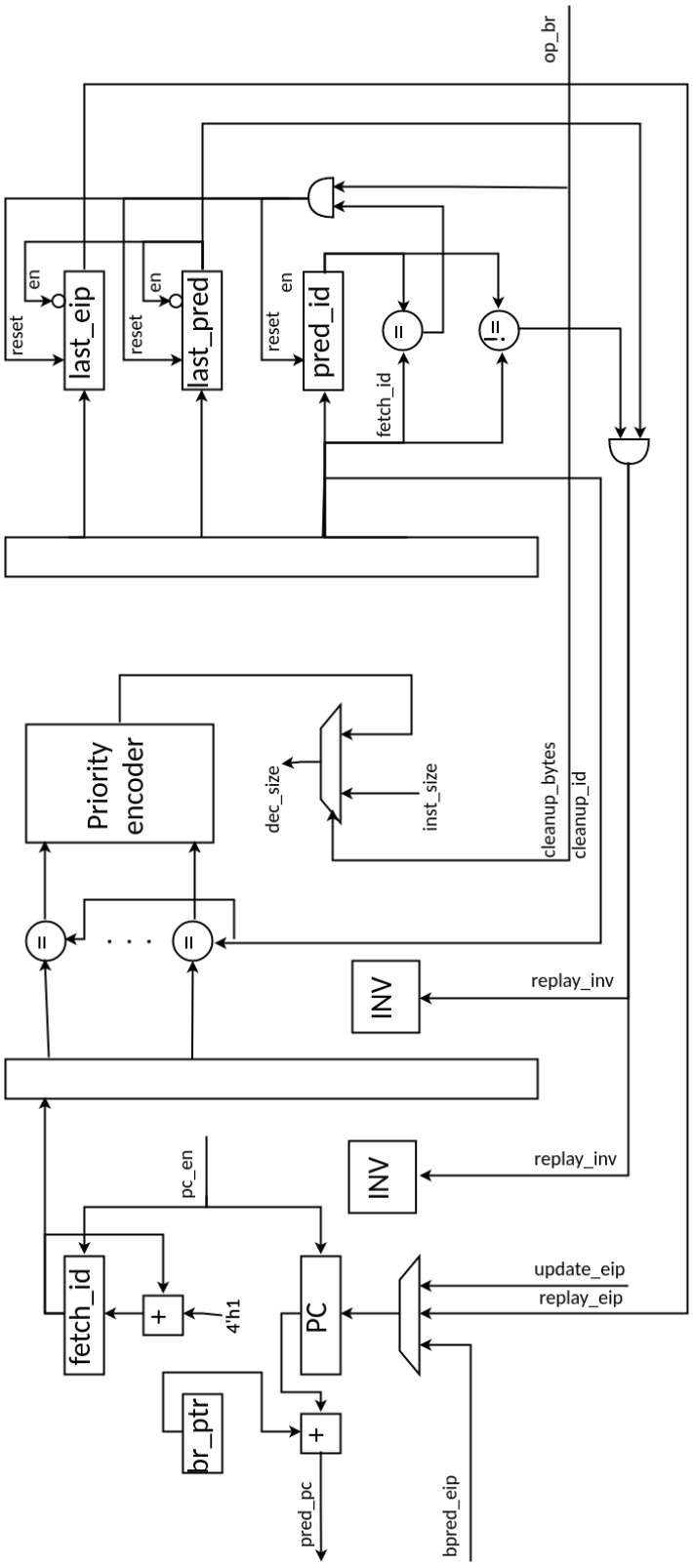


Fig. A.15 Branch Predictor Update

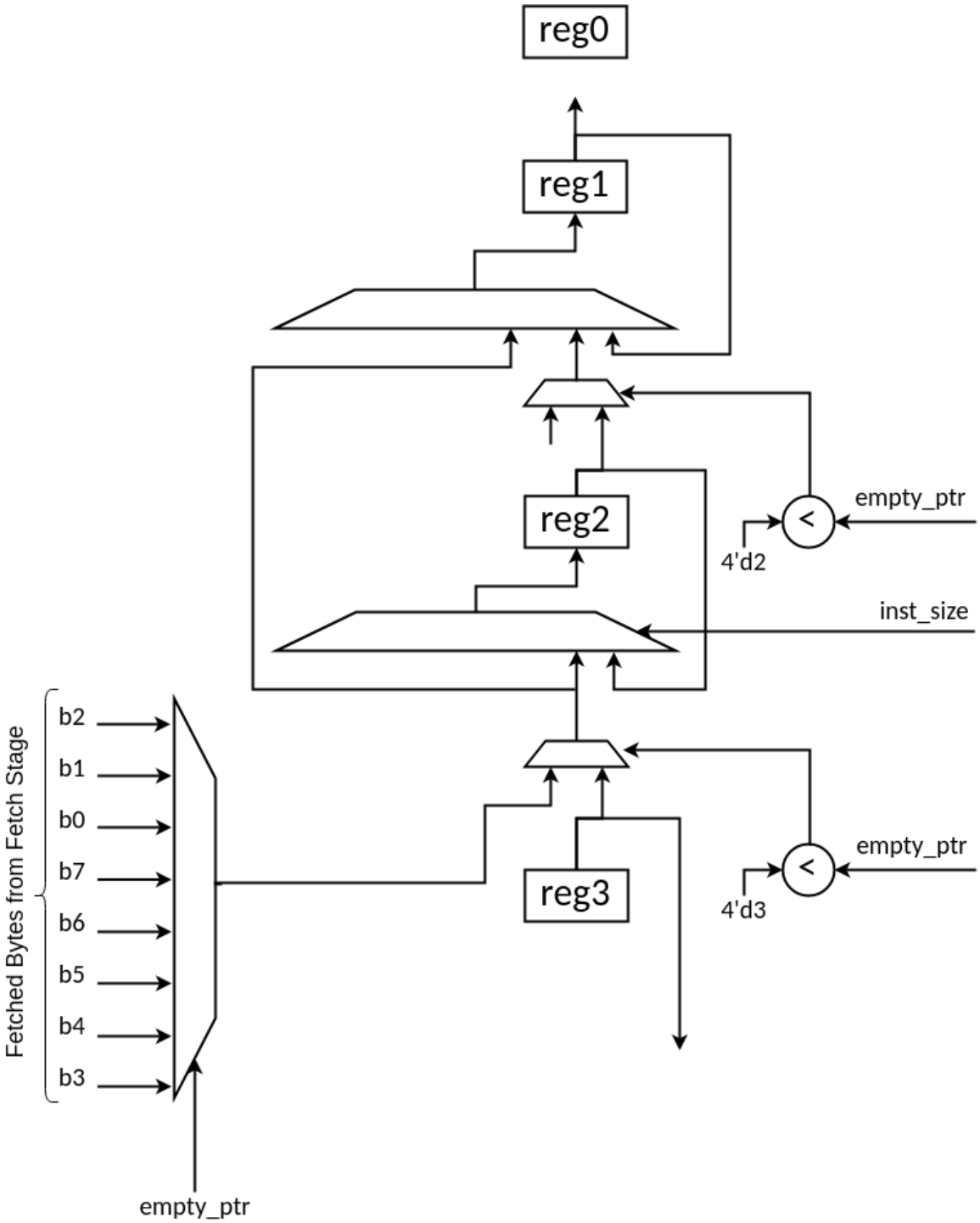


Fig. A.16 Shift Tree Schematics

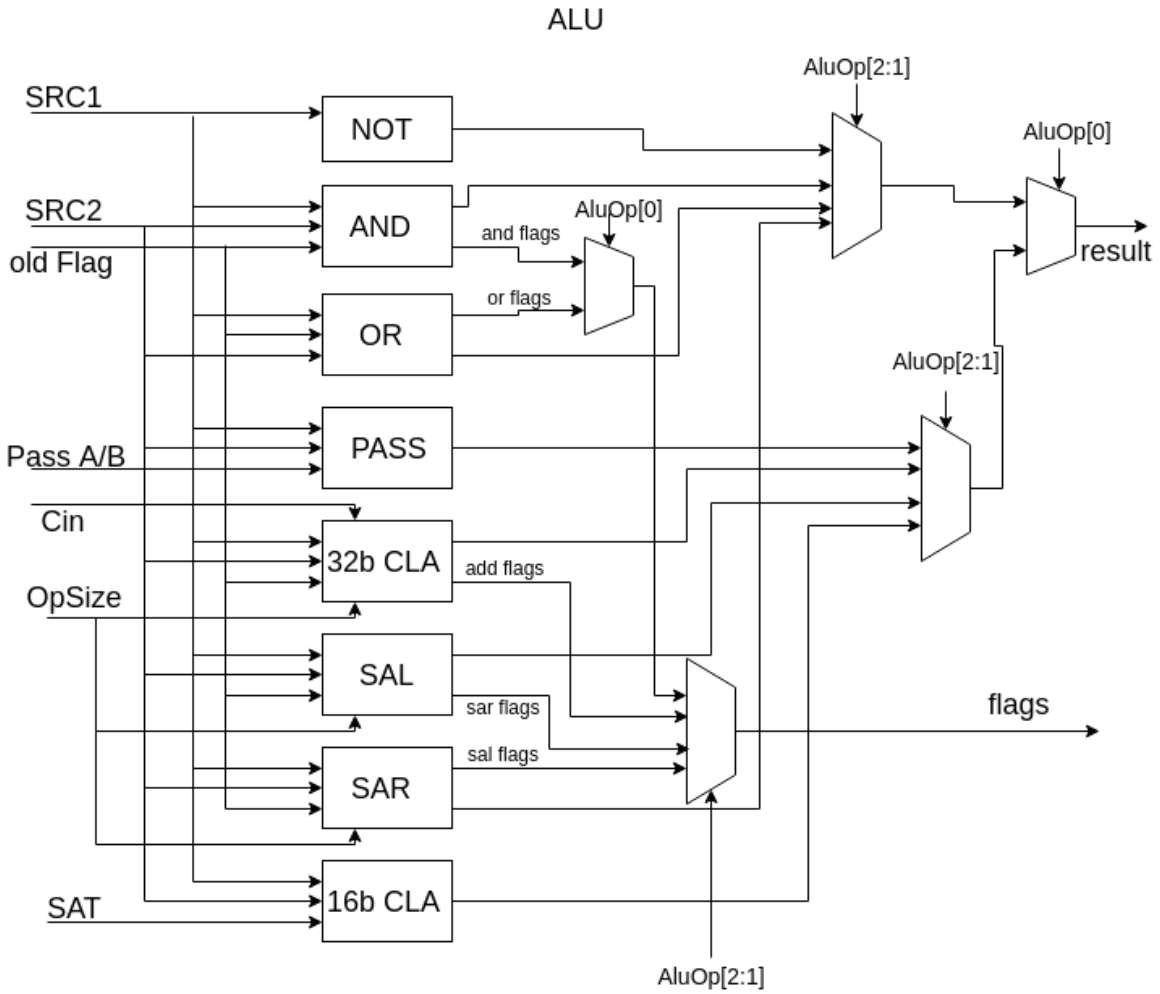

ALU
SRC1
SRC2
old Flag
Pass A/B
Cin
OpSize
SAT
NOT
AND
OR
PASS
32b CLA
SAL
SAR
16b CLA
and flags
or flags
add flags
sar flags
sal flags
AluOp[0]
AluOp[2:1]
AluOp[0]
AluOp[2:1]
AluOp[2:1]
result
flags

Fig. A.17 ALU

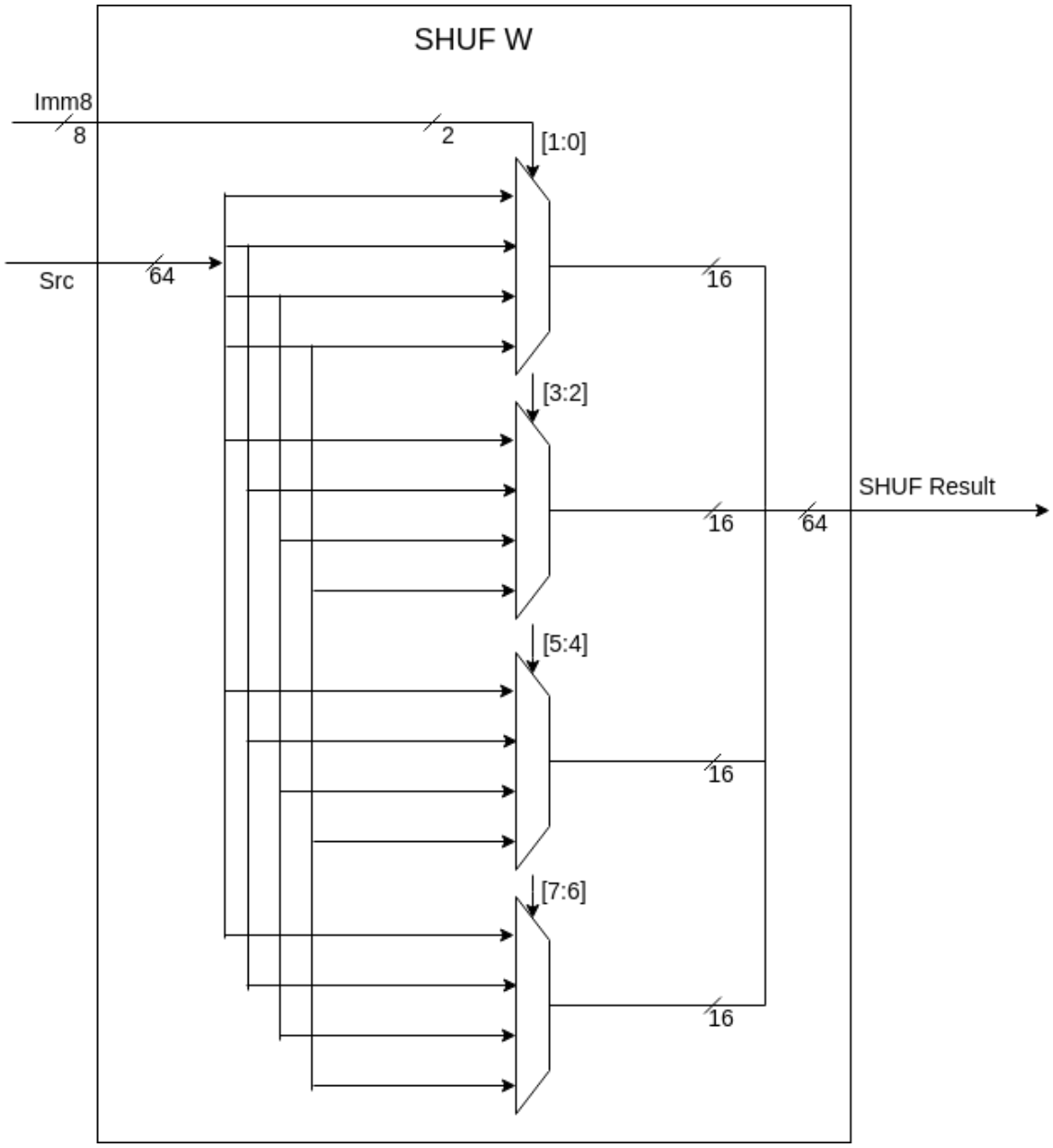


Fig. A.18 PSHUFW Implementation

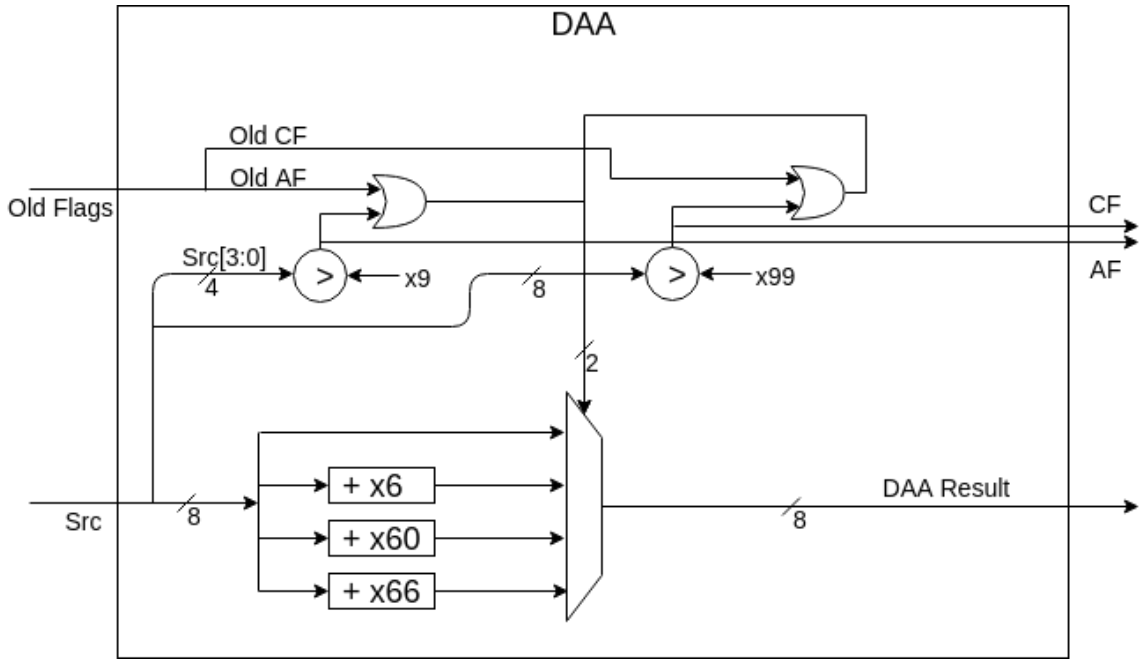


Fig. A.19 DAA

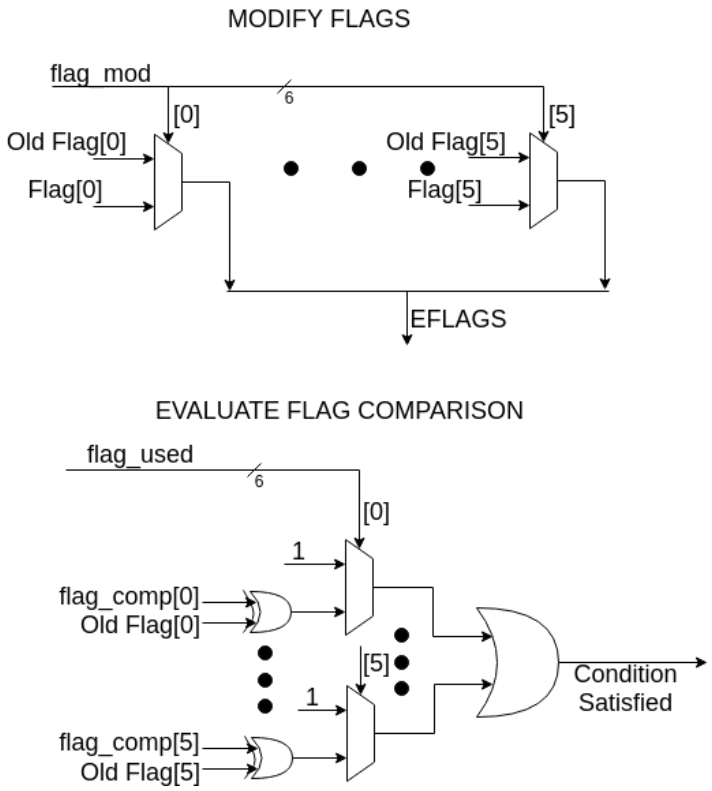


Fig. A.20 Flags Modification and Conditional Execution Paths

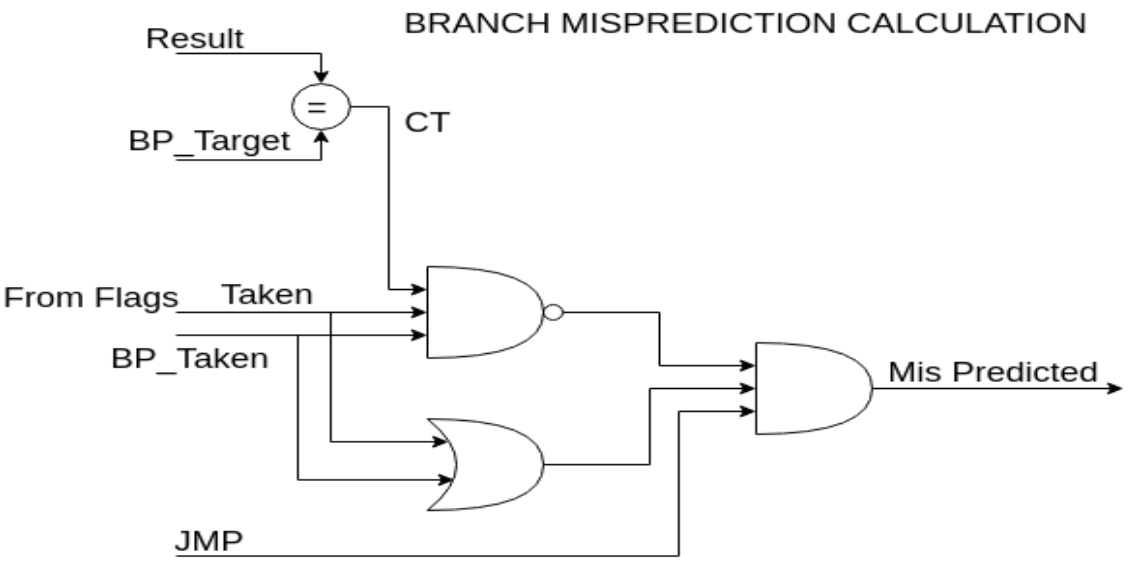


Fig. A.21 Misprediction Calculation Path

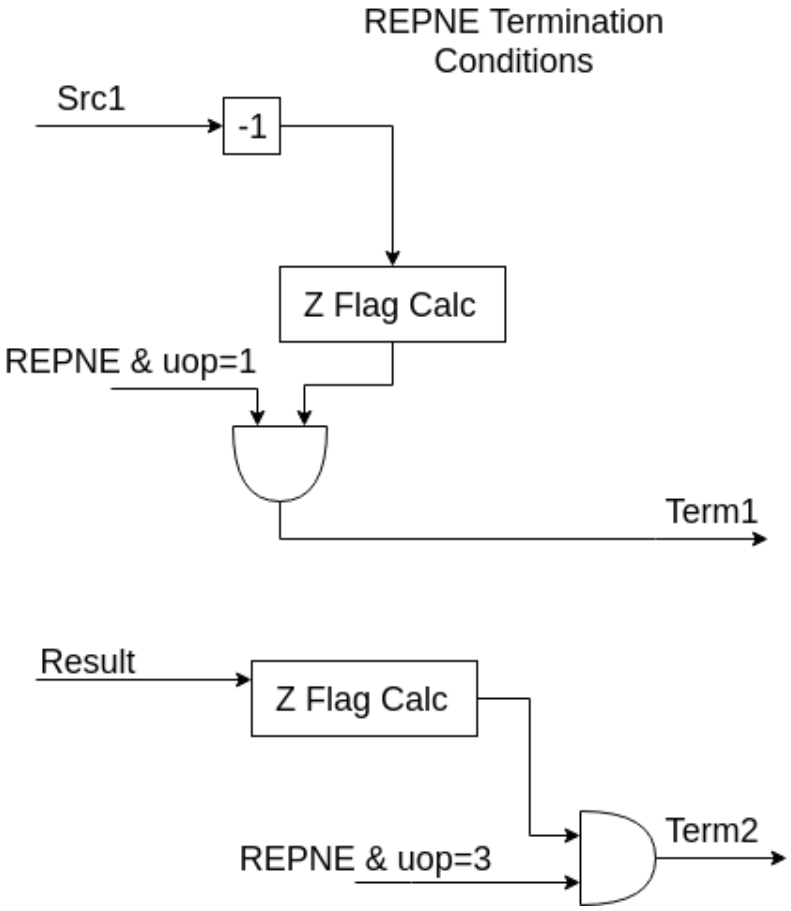


Fig. A.22 Repne Termination conditions evaluation

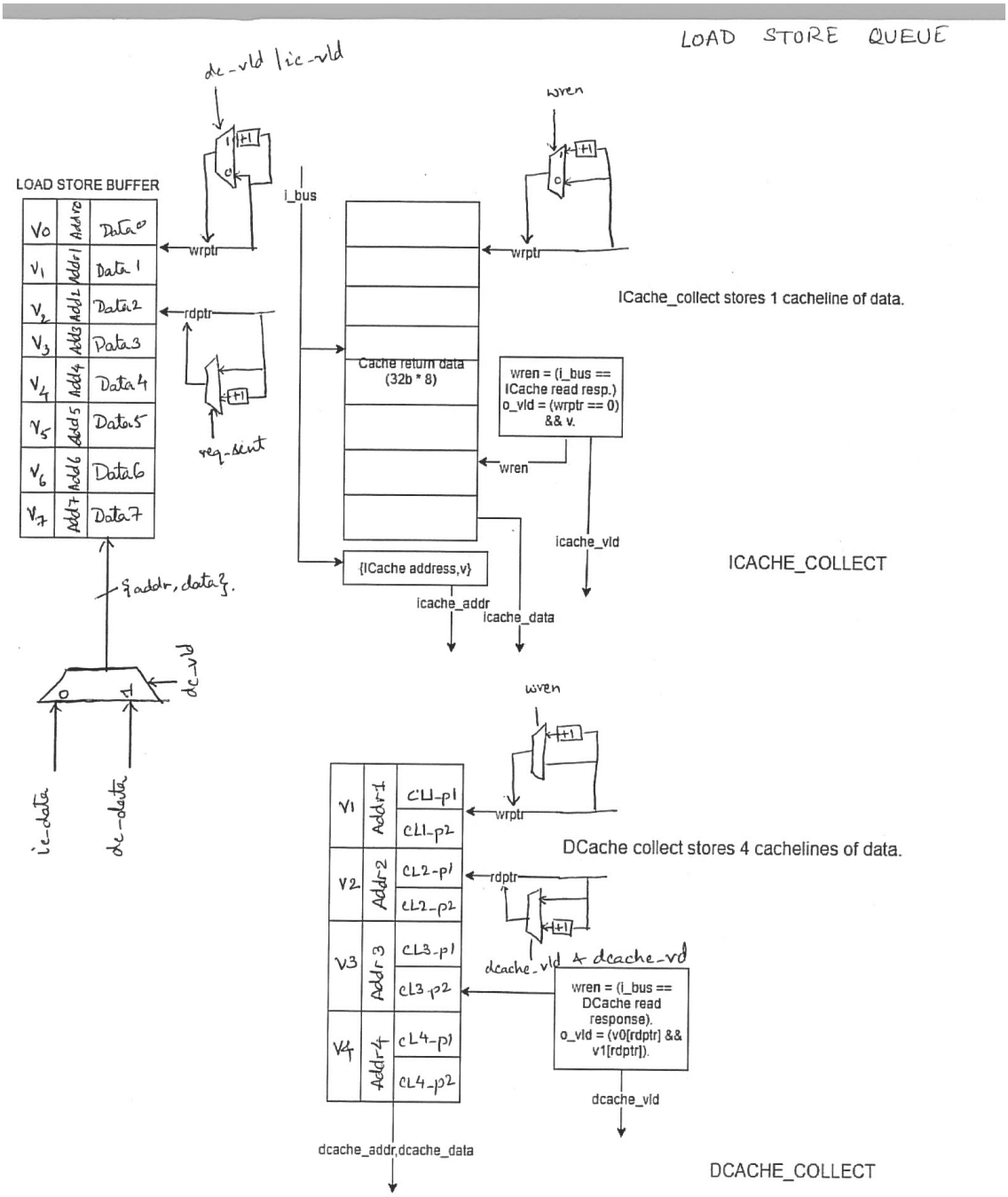


Fig. A.23 Load Store Queue Schematics

Appendix B

Micro-Instruction Format and Micro-code Listings