EE382N: Final Project Report

Students

Apurv Narkhede (arn825)

Nelson Wu (ntw248)

Jiahan Liu (jl57566)

Steven Flolid (sf23839)

EE382N Microarchitecture

University of Texas at Austin, Spring 2018

Instructors

Yale Patt, Professor

Ali Fakhrzadehgan, TA

Siavash Zangeneh, TA

Abstract

This project represents the efforts of four students to learn microarchitecture by creating a gate level implementation of a simulator for a subset of the x86 ISA. Technical emphasis was placed on correctness and speed. The project challenged the team with a new standard of execution, communication, and rigor.

Table of Contents

Section 1: Introduction 7

General Requirements 7

Features 7

Design Approach 7

Section 2: Design Discussion 8

Prefetch and Fetch Stages 9

Decode 10

Address Generation Stage 1 11

Address Generation Stage 2 12

Memory Stage 1 (Memory Dependency Check) 13

Memory Stage 2 (Memory Access) 14

Execute 14

Writeback 15

Load-Store Unit 15

Instruction Fetch Unit 15

Instruction and Data Caches 16

Physical Memory 16

Bus Framework and Arbitrator 16

Memory Controller 17

DMA bus port/controller 17

Keyboard bus port/controller 18

Interrupt bus port 18

Exception and Interrupt Controller 18

REPNE CMPS Implementation 18

Section 3: Design Considerations 20

Section 4: Critical Path Analysis 22

Section 5: Final Design Decisions 32

Section 6: Conclusion 33

Appendix A: High Level Data Path 35

Appendix B: State Diagram 37

Fetch State Diagram 38

IFU State Diagram 39

LSU State Diagram 40

Cache State Diagram 41

Bus Controller State Diagram 42

Appendix C: Pipeline Schematics 43

Prefetch and Fetch Stage Schematic 44

Instruction Length Decode 45

Control Store Address from Decode 46

Prefix Checker and Decoder 47

Opcode Length Decoder 48

Modrm Detector 49

Modrm Pointer Logic 50

Offset Selector 51

Disp Selector 52

Immediate Selector 53

Decode Stage 54

Decoder 55

Decode Stage 56

uControl Store 57
Segment Override 58
Decoder (Stall Logic) 59
Control Store Override 60
Register File 61
Write Logic for Register 62
Prefix Checker 63
Address Generation Stage 1 64
Address Generation Stage 1 65
Address Generation Stage 1: Scoreboard Generator 66
Address Generation Stage 1: Register Dependency Check 67
Address Generation Stage 2 68
Address Generation Stage 2: Segment Limit Check 69
Memory Stage 1 70
Memory Stage 1: Memory Dependency Check 71
Memory Stage 2 72
Execute: operand_select_ex 73
Execute: functional_units_ex 74
Execute: cmpxchg_decision_ex 75
Execute: repne_support_ex 76
Execute: result_select_ex 78
Kogge Stone Adder 80
Kogge Stone Carry Look Ahead 81
Kogge Stone Adder subcomponents 82
Execute: alu_daa 83
Execute: Flags 84
Execute: PADDSW 85
Writeback: operand_select_wb 86
Writeback: conditional_support_wb 87
Writeback: flags_wb 88

Writeback: halt_repne_support_wb 89
Writeback: result_select_wb 90
Appendix D: Memory Hierarchy and Bus Schematics 91
Cache Controller 92
Bus Port Controller 93
Appendix E: Interrupt and Exceptions 94
Appendix G: Complete Description of Control Logic 97
APPENDIX F: Parts Listing 118

Section 1: Introduction

General Requirements

Our design met the general requirements given in the project specifications.

1. All opcodes and data paths as specified
2. All addressing modes and data types as specified
3. Instruction cache and data cache
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. One simple I/O device - a keyboard controller
7. One complex I/O device - DMA controller
8. The handling of exceptions (e.g., segment limit) and interrupts (e.g., I/O device)

Features

Our design contained the following features beyond the general requirements.

1. 8 stage pipeline
2. Register file with 4 read ports and 3 write ports
3. Prefetching the instructions. The fetch stage proactively fills a buffer that can hold 4 cache lines with instructions that follow the instruction being decoded.
4. Not-taken branch predictor. The simulator speculatively executes instructions and flushes pipeline if prediction is incorrect.
5. Speculative execution for repne cmps. The pipeline is filled with iterations of repne cmps before each iteration's termination condition is resolved.

Design Approach

The team had four members and the project was split into four tasks. First memory hierarchy, bus and I/O. Then decode, registers, formal verification unit tests, and integration. Then fetch, address generation, memory, interrupts and exceptions. Lastly execute, writeback, and individual instruction testing. The project was integrated successful late in the design process. As a result, decisions to improve to the critical path and instructions per cycle was made for each unit in

isolation. In hindsight, integration should happen early in the design process, so the performance improvements can be made in the context of the whole microarchitecture.

Section 2: Design Discussion

This section provides a brief description of the stages of the pipeline, instruction and data caches, physical memory, and bus arbitrators and controllers, exception and interrupt controls, and the implementation of repne cmps. Figure 2.1 illustrates how these units are interconnected in the simulator.

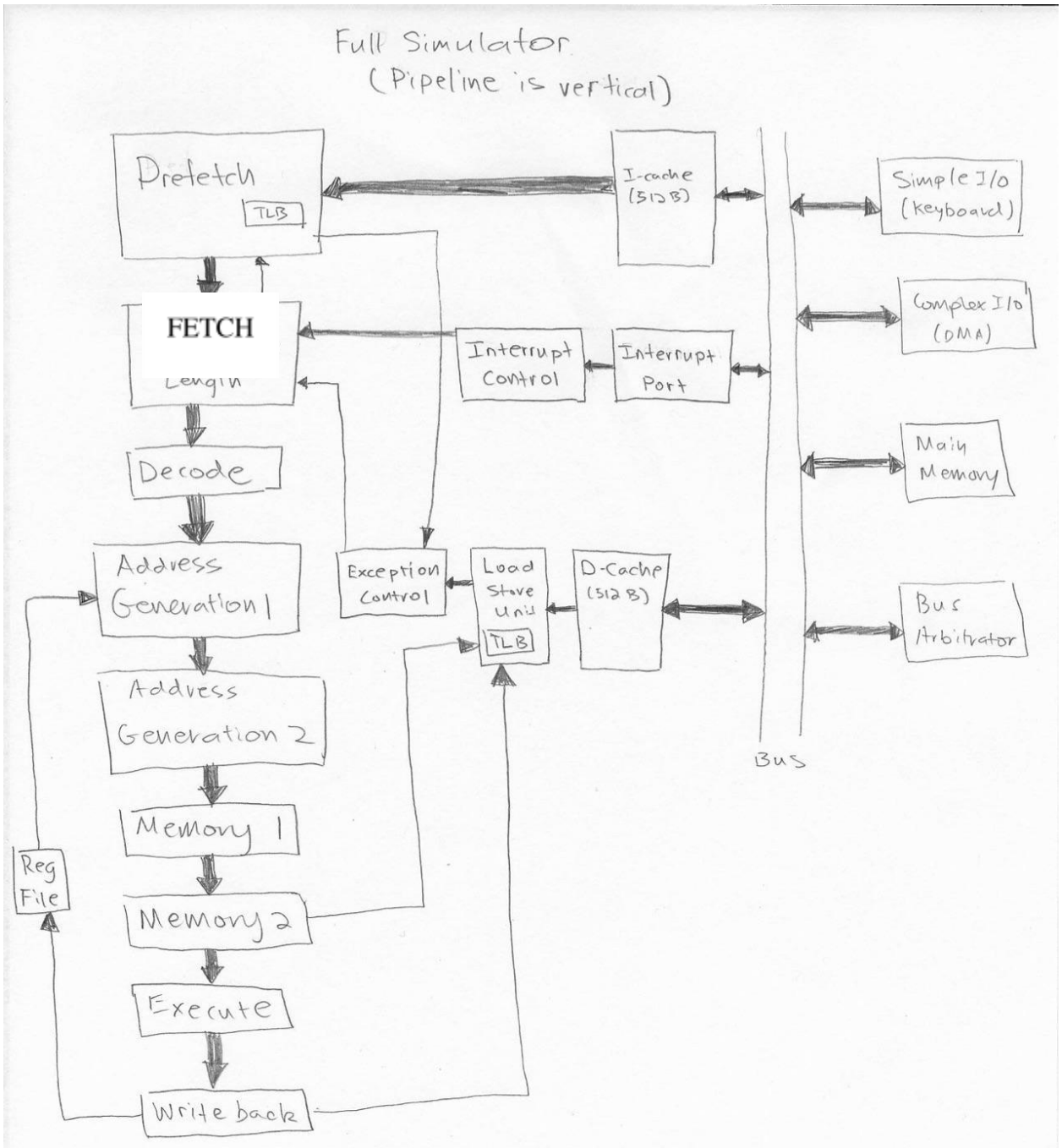


Figure 2.1

The pipeline has nine stages: prefetch (PR), fetch (FE), decode (DE), address generation stage one (AG1), address generation stage two (AG2), memory stage one (ME1), memory stage two (ME2), execute (EX), and writeback (WB).

It additionally contains an instruction fetch unit (IFU), load-store unit (LSU), an ITLB, a DTLB, and interfaces to the instruction and data caches (ICACHE, DCACHE).

Prefetch and Fetch Stages

The prefetch stage reads data from the instruction cache to be pre-decoded in the next stage. The fetch pointer passed to the ICACHE is updated independently of the instruction in decode until the buffer is full or a control flow event is detected. For the pipeline to be able to fetch a new instruction and decode the current instruction in the same cycle, the fetched instructions must be buffered. The prefetch buffer is 64 bytes, so it holds four cache lines.

The flushing and filling of instruction buffers is directed by a finite state machine. Initially, the buffer is empty. The prefetch stage will make a request to the instruction cache. Once the data is ready, the prefetch stage leaves the empty state and transitions into the fill state and writes the cache line to the buffer. A two bit next buffer pointer keeps track of the next buffer line to be written to, and a six bit read pointer keeps track of the instruction stream as instructions are decoded. The finite state machine will determine if the buffer is full based on these two pointers. The length of the instruction being decoded must be calculated in the following fetch stage so that the read pointer can be updated every cycle as long as the pipeline is not stalled. This allows the maximum bandwidth to be one instruction per cycle.

In the case of a jump instruction, return instruction, interrupt, exception, or branch misprediction, the buffer must be flushed. In the flush state, the read pointer is not updated. In the event of a page fault, the read pointer will continue updating until the buffer is empty and wait for the page fault to be serviced.

In the fetch stage, the length decoding of the instruction bytes in IR happens in parallel, where the prefix can be in the first 3 bytes of IR - IR[127:104], the opcode can be in the first 5 bytes, and so on for the other fields. The prefix checker checks the first 5 bytes of IR - IR[127:88] and gives information about whether each byte is a prefix, and is it a repeat prefix, segment override or operand override. If it is not a prefix, it can be an opcode. The prefix decoder takes these signals as inputs and tells whether a prefix is present, and gives the prefix size and the segment ID in case of segment override. The 2nd byte can be a prefix only if the 1st byte is prefix, and the 3rd byte can be a prefix only if the first 2 bytes are both prefixes. The opcode length decoder gives the opcode size and opcode select as outputs. The opcode is two bytes, only if the first byte of opcode is 8'h0F. The opcode select signal choses the correct bytes of IR as opcode using muxes. The modrm detector tells whether a modrm is present or not based on the opcode. The immediate detector takes the opcode and operand override as an input and tells whether an immediate is present and the size of immediate. The SIB and displacement detector takes the modrm, and modrm present as input and tells whether sib is present, displacement is present and the size of displacement. The offset detector takes the opcode, operand override and opcode size as input and gives information about whether the offset is present or not and the size of offset. The modrm pointer module takes opcode size, prefix present and prefix size as input and gives the location of byte to be selected as modrm from the IR. The same pointer can be used to select the SIB and offset by setting the input of muxes appropriately to choose the correct bytes. The instruction length decoder takes the sizes of all the different fields, and whether the fields are present or not and gives the instruction length for the current instruction. This instruction length is then used to update the read pointer to shift and mask the instruction buffers to get the next instruction.

The control store address based on the opcode is also generated in this stage. We use only the opcode[7:0] bits and the opcode size to address the control store Hence, some additional logic is necessary to map the opcodes having conflicts because of the same bits, and also for opcodes whose functionality is different based on the reg/opcode field of the modrm. In case of interrupt, the interrupt address for micro control store is passed on instead of the opcode mapping.

Decode

The decode stage decodes the instruction in the instruction buffer coming from fetch stage. The decode stage generates the control signals needed for the next stages based on the determined fields. The control store address is generated based on the opcode for each instruction. The final control address is selected between this address, the interrupt or exception micro-code address, and the next micro-sequencer address for instructions with multiple micro-operations. As we use only the opcode to generate the control store address, some override signals are required for control signals which are affected by modrm such as the addressing modes, operation of the execution unit for instructions which have the same opcode(ADD, AND, ..) and other such control signals. The segment ID for push and pop instructions depend directly on their opcode. In case of EBP, and ESP registers being used, the segment should be SS. Thus, the segment ID is either SS, override due to PUSH and POP, DS, or the segment override due to prefix. The schematics and the micro-code listings in the appendix detail how the fields are decoded and how the control signals generated. The EIP is updated in this stage with the instruction length generated in the previous stage.

Address Generation Stage 1

In the first stage of address generation (AG1), the pipeline does a register dependency check for all later stages except WB. The register file writes on the negative edge from WB so that AG1 can read from it in the same cycle before the pipeline is clocked at the positive edge.

The dependency check is done through a scoreboard. For GPRs, due to the different sizes of eight bits, 16 bits, or 32 bits, the vector is 24 bits wide. Each of the eight GPRs is represented by three bits - bit two indicates if WB will write the upper 16 bits of a GPR, bit one for the upper eight bits of the least significant 16 bits, and bit zero for the lower eight bits of the lower 16 bits. An 8-bit size clears the most significant bit of an ID so that AH, CH, DH, and BH map to the same 32-bit register as AL, CL, DL, and BL. 16 and 32-bit sizes map directly to ID. The segment register and MM register scoreboards are each eight bits and translate directly from the ID.

AG1 will check the GPR scoreboard from the next four stages for each of the four possible read IDs, depending on if the register is needed for the current instruction. Similarly, AG1 checks the

segment register scoreboard and MM register scoreboard for the two possible read IDs of each type depending on need.

The GPR scoreboard generates based on the three possible destination GPRs and load signals, and the SEG and MM scoreboards generate according to the single destination of each type. All three scoreboards pass through to AG2, ME1, ME2, and EX, and those stages pass their respective scoreboards back to AG1 for the dependency check. The valid signal clears all scoreboards.

AG1 also performs the first steps of generating read and write addresses. Based on the decode control signals for base register, SIB byte, displacement, and segment register, AG1 adds the appropriate register data to form the intermediate address. This is passed through to the next stage, AG2. For stack operations, the stack register can also be added to the push or pop size. The stack pointer is saved in an internal register in case of back-to-back stack operations (such as for IRETD or interrupt saves) to not stall the pipeline. WB only updates ESP on the final operation of a multi-stack access sequence.

Instructions needing to push EIP or CS on the stack select the correct data according to operand size and save them in the MM.A latch. This is done so that EIP and CS are always written from the MM.A field and can be pushed at the same time, saving one memory write if both need to be pushed (e.g. for far CALL and interrupt saves).

REPNE CMPS operations stall in AG1 if an earlier instruction writes to DF so that the correct direction gets added to the compared addresses. This is accomplished by one bit and a load signal from the following four stages if DF gets written and a repeat operation has started through the pipeline.

AG1 uses the EXC.EN signals passed from decode to choose the right interrupt vector. General protection exceptions have the highest priority, page fault exceptions the next, and interrupts the last. A priority encoder selects the highest interrupt vector and passes it to AG2.

Finally, AG1 also generates the target address of control flow instructions using a relative displacement (e.g. JMP rel and CALL rel). These are loaded into the next EIP and next CS latches if required.

Address Generation Stage 2

In the second stage of address generation (AG2), the segment limit check is done, and the final linear addresses are generated. The segment limit check uses the segment ID passed from AG1 to select the correct limit. In parallel, the intermediate address is added to the scaled index data (for SIB), and the size is subtracted from the limit. These are compared to generate the general protection exception. The incremented EIP (current EIP added to length) is also compared to the CS limit to check if the current instruction has crossed that boundary as well. Stack limit checking is not done. General protection exceptions pass to WB, which signals that the current instruction generated an exception.

The intermediate addresses are also added to the appropriate segment register data to generate the linear addresses. There are four possible read addresses: from a base register access, stack access, interrupt table read, or CMPS address. There are two write addresses: base register writes, and stack writes.

AG2 contains structures for speeding up REPNE CMPS. The ESI and EDI data and addresses are saved so that each does not need to be read for subsequent iterations. The current DF value decides whether to add or subtract from each address after having stalled in AG1 on a write to DF. We also check the segment limits for ESI and EDI offsets based on the segment ID (either DS, which may be overwritten, or ES). The REPNE instruction contains four microinstructions - the first two behave as a normal CMPS, and the last two are repeated. Count and offset registers are written in WB only on the second memory read after the comparison to facilitate flushing the pipeline.

Memory Stage 1 (Memory Dependency Check)

Using the read and write addresses passed from AG2, the memory dependency check stage (ME1) accesses the DTLB to translate linear addresses to physical addresses. Two pairs of real

addresses generate: two addresses for read, and two addresses for write. Addresses crossing a cache line boundary (16B) because of a 2B, 4B, or 8B load or store are accessed in two operations to support the data cache. This method also supports page boundary crossing (4 KB), in which case two non-contiguous real addresses get passed, along with the access sizes of each. The following two stages (ME2 and EX) pass back the real addresses and sizes generated from ME1 for the dependency check. If a read address in ME1 overlaps a write address in ME2 or EX, the ME1 stage stalls.

General protection exceptions are generated if a write address corresponds to a TLB entry set to read-only. Page fault exception are generated when a TLB entry is not found or is invalid or not present in physical memory. An MMIO address need not be present in physical memory. We chose real address range 0x7000 to 0x7FFF as the memory mapped region. All exceptions generated in ME1 are also passed through to WB, which signals what type of exceptions occurred.

WB does not stall for dependency checks in ME1 because the load-store unit (LSU) prioritizes writes over reads. Return immediate instructions add the stack pointer data to the immediate passed from decode when needed.

Memory Stage 2 (Memory Access)

The memory access stage (ME2) passes real addresses generated in ME1 to the load-store unit (LSU) if an instruction needs to read from memory. ME2 stalls as long as the data cache is not ready and a read operation is in progress. After valid read data gets returned, ME2 generates the correct data for EX. Regular instructions (e.g. ADD, AND, OR) needing to read from memory have operands passed through latch A, possibly sign extended. Control flow instructions requiring memory data (e.g. JMP r/m or IRETD) load the next EIP and next CS latches if needed. Both EIP and CS are read in the same operation for far CALLs and IRETDs to reduce the number of stack accesses. Operand size determines what part of the 64-bit data represents EIP and CS (if needed). The upper 16-bits of EIP are also cleared for 16-bit CALLs or RETs.

Execute

The execute stage contains the functional units and the logic to determine what data from the memory access stage goes into each functional unit, and which results are passed onto write back. The routing of data in and out of the functional units is controlled by microcode to make the design flexible. The functional units had to take data size into consideration when calculating the result and flags. The functional units are add, or, not, daa, and, cld, cmp, std, paddw, paddd, paddsw, pshufw, pass_a, swap_a, sar, sal, update pointer, decrement count. Refer to appendix for specifications on each of these functional units.

Writeback

The writeback stage determines which data from execute stage is written into the general purpose registers, segment registers, eip, cs, mm registers, dcache, and flags register. Special support was made for IRETD, JNE, JBNE, CMOVC. To make the design flexible, we saved the neip, ncs, and count to ensure that in multiple micro-op instructions, only the last micro-op would commit to the architectural state. This design decision let our simulator handle precise exceptions in instructions such as REPNE CMPS with low latency because the pipeline can be flushed at any time.

Load-Store Unit

The load-store unit (LSU) accepts requests from both ME2 and WB. Write operations are prioritized over read operations, so reads are stalled until writes complete. If a read operation is in progress when a write is needed, the write is stalled to prevent changing the address. Since write stalls, stall all previous stages, this has the side effect of forcing the read to happen twice because the data cannot be latched into the next pipeline latches, which is a performance hit. If the write also hits in the same set at a different cache line, the write would evict the read cache line, causing in the worst case three caches misses.

The LSU has a controller that issues cache writes or reads in the same cycle. If the DCACHE is not ready, the LSU enters the next state and waits before returning to IDLE. Reads and writes that cross cache line or page boundaries must read or write two different addresses in different cycles before returning to IDLE.

Instruction Fetch Unit

The instruction fetch unit (IFU) interfaces only to fetch. It contains both an ITLB and read controller. The ITLB returns two read addresses if a fetch crosses a cache line boundary or page boundary. Accesses occur in two different cycles in this case. The controller issues the read immediately, and enters the stall state if the ICACHE is not ready. When a boundary is crossed, the next read is completed before returning to IDLE.

The IFU can also return a page fault for fetch addresses not found in the ITLB. In this case, the ICACHE is not enabled, and the IFU stays in IDLE.

Instruction and Data Caches

We designed a balanced and identical cache system for the instruction and data cache. Therefore, both the instruction and data cache were 512B each. We chose a 16 byte cache line to adequately capture locality of programs while not bringing in too much data with each cache line. We implemented a Moore FSM to drive the cache's functionality. This allowed our cache to handle hits, misses, and evicts in an efficient manner. In case of a tag mismatch, the current cache line was evicted and the new cache line was brought in from the main memory. All the cache lines are invalid at reset, and they are set to valid when the data is brought from main memory. Due to the design of our FSM, a cache hit took two cycles. This accounts for the transition from idle, to recognizing the hit, to returning the data. We recognize that the cache could be improved by either increasing the cache clock or by switching to a Mealy state machine.

Physical Memory

The physical memory was implemented as a 1024x32 array of 8-bit SRAM cells. We had a row decoder to select the correct row for reading or writing. We write to the appropriate SRAM cells using the column decoder and we read the entire 32 byte row and put it into the read buffer. A selection logic chooses either to select the high 16 bytes, or the low 16 bytes of the read buffer and transfer it over the bus to the caches. Our main memory supports 1-byte, 2-byte, 3-byte, 4-byte and 16-byte writes as those were the only cases needed for this project. The 16-byte write happened while evicting a cache line whereas the other write sizes were utilized for the DMA transfer.

Bus Framework and Arbitrator

The Bus for our system is a centrally arbitrated split transaction synchronous network. A simple FSM is used by each of the bus ports that allow for straightforward communication. All bus ports can request access to the bus and the central arbitrator will decide the master for the next cycle. The arbitrator selects the next master based on priority and if there is already a bus master. Split transaction means that when reading or writing, we can split a request up into multiple parts. This allows our memory controller to service multiple requests in parallel. This is especially important when streaming data to and from the DMA. We chose to break bus accesses up into 16 byte packets, this closely mirrored our cache line size for efficient cache memory accesses.

Each port also had a state machine to track its state. The state machine had states for both being the slave on the bus and for being the master. This common state machine allowed for could code reuse for each of the bus ports. The keyboard, interrupt controller, and caches all had very similar bus ports.

Some signals did not use the main bus but were instead directly connected, this included the destination wires that signalled a incoming request. We made this decision because keying off a destination wire that was often high z led to issues with our state machine. A similar problem occurred with the acknowledge handshake signal.

Memory Controller

The memory controller required the most modifications from the base bus controller. The memory controller added two pipelined set of latches. The processing latches contain the relevant information for the request currently being processed by the physical memory. These latches include the address, size, R/W, and source of the memory request. While a memory access was processed, the Memory controller could still load an additional memory request into the on-deck latches. These propagated to the processing latches once the memory access has completed.

DMA bus port/controller

The DMA controller consisted of the common bus port controller along with a few extra registers. Each of the four memory mapped registers had a register in the controller that were used to track a DMA request. A 4KB buffer held the data to be transferred to Memory. The DMA memory accesses were handled the same as any other bus user, the only difference being that the size of access was much larger. This effectively increased the amount of time to complete the request. It is important to note that the bus only transferred data in 16 byte chunks. Therefore, the DMA controller did not block the bus while transferring data.

Keyboard bus port/controller

The keyboard controller was similar to the template bus controller. The only necessary addition was a memory mapped register to provide data from and a interrupt flag to trigger an interrupt. When an interrupt occured, the keyboard controller passed the signal along to the interrupt bus port which would forward the data to the pipeline.

Interrupt bus port

The interrupt port served as a way to bypass the cache for memory mapped I/O and for interrupt signals to propagate directly to the pipeline as well. This required the Keyboard and DMA to send their requests to the Interrupt port instead of to the caches.

Exception and Interrupt Controller

The exception and interrupt controller is logic in decode that allows interrupts to take control of the instruction pipeline. The interrupt or exception will load an address from the micro-code store and pass the control signals through the pipeline. Once the multiple microps for the interrupt/exception complete, the EIP will be ready to process the correct handler for execution.

Exception signals are passed from WB to allow all earlier instructions in the pipeline to complete before taking the precise exception. Instructions in all other stages (that entered after the faulting instruction) are flushed when taking the exception.

REPNE CMPS Implementation

We first implemented CMPS, then REPNE CMPS. Our implementation of REPNE CMPS is efficient because it speculatively fills the pipeline with iterations before the termination condition for the previous iterations are checked.

We implemented a single CMPS in two micro-ops. The first access reads the first memory location and increments the first pointer. During the first micro-op, the comparison operand is saved in a temporary register in execute, and the updated pointer is saved in a temporary register in writeback. The second access reads the second memory location and performs the comparison using the new and saved data. Both pointers are written back to the register file during the second micro-op to enable the pipeline to be flushed in the any micro-op.

We extended the single CMPS implementation for REPNE CMPS by the following solution. When a REPNE CMPS is decoded, decode stage stalls fetch and checks if any earlier instruction affects the DF flag. If there is an instruction that affects the DF flag then the pipeline stalls until no instructions in the pipeline affect the DF flag. Once the DF flag is constant, the decode stage will send out REPNE CMPS micro-ops. The REPNE CMPS micro-ops are identical to the CMPS micro-ops with the addition that they also read the count, decrement it, store it back, and perform the associated termination condition logic. The read pointers are also saved and auto-incremented in address generation stage. Once one of the termination conditions are met, then the pipeline flushes using the same flush logic as our branch predictor. Since both pointers and count are written back in the second of micro-op of every iteration, the pipeline can be flushed anytime in the case of an interrupt or exception.

Section 3: Design Considerations

The goal of the design process is to implement a correct and fast machine. Our design process involved iterative improvements on each unit of pipeline because integration did not happen until late in the process. During each iteration, we increased instructions per cycle, reduced cycle time, and wrote more tests. We will discuss the free optimizations that we made before integration and the optimizations through tradeoffs that we made after integration.

The most important goal of the project is to have a correct simulator. In the real world, no customer will buy a processor with errors. In the context of this project, the cycle time or instructions per cycle would not matter if the simulator could not pass all the test cases. The team performed formal verification methods on each unit, created test benches for every opcode, and created test scenarios for cache and main memory. We focused on a microcode centered design in the pipeline so that errors could be fixed without changing the datapath. Abstracting the pipeline into microcode also provides a structured way to think about the pipeline, thus reducing human error.

Instructions per cycle was maximized by minimizing stalls, having separate TLBs for instructions and data, and reducing the number of micro-ops for multiple micro-op instructions. To avoid stalling for flag dependencies on conditional moves and jumps, conditions are checked in writeback when older instructions have already retired. Reducing the number of micro-ops for micro-op instructions also increases IPC. We used a separate ITLB with two read ports and a DTLB with four read ports to allow instruction fetching to progress simultaneously with data reads and writes. Two read ports are needed if an instruction fetch crosses a page boundary. The DTLB has two read ports for read addresses, and two write ports for write addresses for the same reason. We added specialize logic to reduced call, ret, cmpxchg down to one micro-op, ret, iretd, interrupt/exception, each cmps in repne cmps down to two micro-ops.

Cycle time was minimized by splitting pipelines stages and using a faster adder. The cycle time was limited by address generation and memory access stages because checking register and

address dependencies were on the critical path. We split the longest stages, address generation and memory access, into four stages. We also implemented a Kogge-Stone adder to decrease the critical path of all the stages that used an adder.

For REPNE CMPS, we initially chose to stall during each iteration to avoid having extra logic dedicated to servicing the repeat. Since it was just one instruction, probably rarely encountered, it did not make sense to devote extra structures for that purpose.

We also initially chose to stall on all conditional jumps as well as unconditional to avoid having to flush the pipeline on a mispredicted branch.

Section 4: Critical Path Analysis

This section provides the critical path analysis of each stage of the pipeline done by hand calculations.

The critical path for the entire pipeline was in Fetch. Theoretical critical path for Fetch was 13.89 ns. Actual was around 14.0 ns.

Prefetch:
28-bit add for fetch pointer (always add 0x10) = 4.80 ns

- 0.22 ns for (1) 3-to-1 mux
- 0.14 ns setup time for fetch_ptr register

= **5.16 ns**

Fetch:
Critical path of pre-decode is 12.24ns (to get the instruction length update from IR)

(3) 4-to-1 muxes, 128-bits (for shifter) = 0.66 ns

- 10.69 ns for length update
- 2.4 ns 6-bit add (worst case)
- 0.14 ns setup time for read_ptr register

= **13.89 ns (actual ~14.0 ns)**

Decode: 3.73 ns for DE_SEG1_ID (Access for control store = 1.68ns)

Address Generation Stage 1:
shifter has (5) 2-to-1 muxes = 1.0 ns

- 0.2 ns for (1) 2-to-1 mux
- 3.0 ns for 16-bit add (16-bit seg register)

= **4.2 ns AFTER register write on neg edge and new read data**

register read about 1.0 ns after write (in custom GPR file)
= **5.2 ns total in AG1**

Address Generation Stage 2:
segment limit check is (1) 8-to-1 mux (32-bits) = 0.5 select signal + 0.2 (2-to-1 mux)

- 4.2 ns for 32-bit adder
- 2.31 ns for 32-bit mag comp
- 1.05 ns for (3) and3
- 0.40 ns for (1) or3

= **8.66 ns in AG2**

Memory Stage 1:
memory dependency check is TLB lookup 32-bit comparator = 2.31 ns

- 0.76 ns pencoder

- 0.7 ns (8-to-1 mux select signal)
- 4.2 ns for 32-bit add
- 0.24 ns for bufferH16
- 0.7 ns for (2) and3
- 0.40 ns for (2) or3

= **9.31 ns in ME1**

Memory Stage 2:
cache ready at beginning of cycle
after memory read is bufferH64 = 0.30 ns

- 0.22 (1) 4-to-1 mux
- 0.2 (1) 2-to-1 mux

= **0.72 ns AFTER memory returned**
minimum 32-bit add = 4.2 ns

- 0.2 ns for (1) 2-to-1 mux

= **4.4 ns**

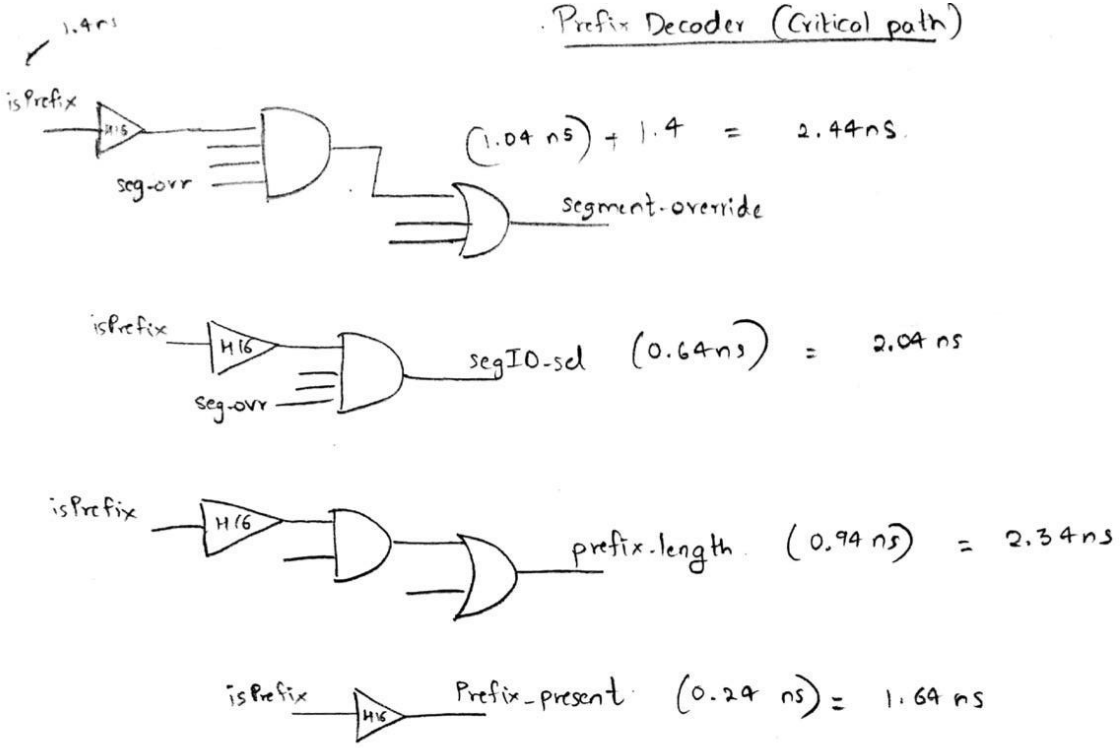
Execute: 8.29ns (cumulative time, note that the previous calculations were not cumulative)

- 0.3ns seconds operand select
- 8.04ns after ALU32 (ALUK = DAA)
- 8.29 on ex_ld_gpr1 after cmpxchg decision.

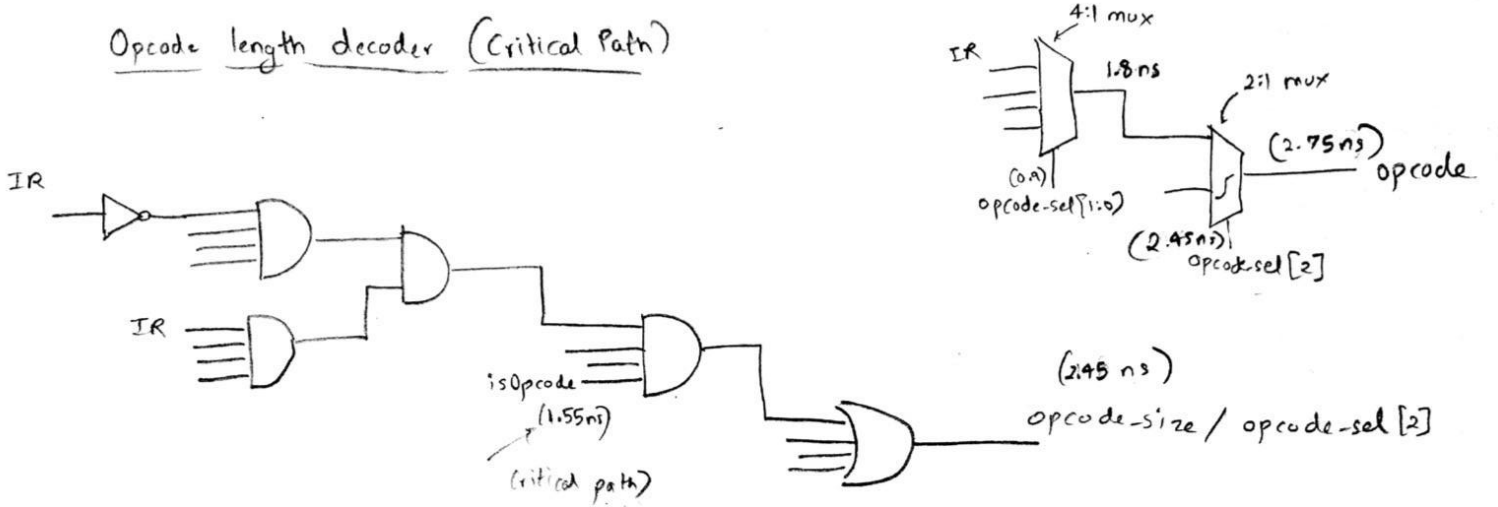
Writeback: 2.75ns

- 2ns after comparing count to 0 in
- 2.75ns after wb_repne_terminate_all

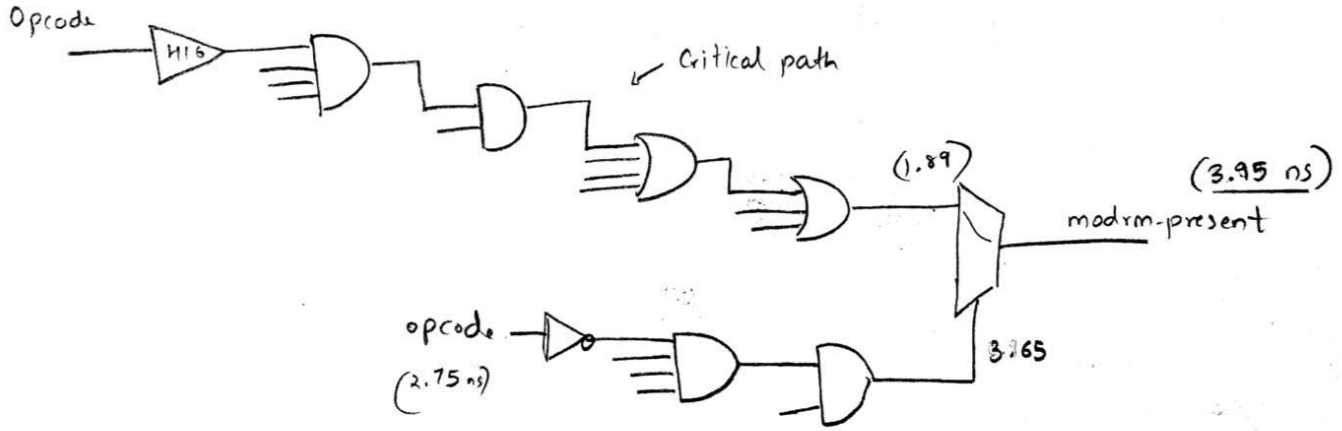
Prefix Decoder (Critical path)



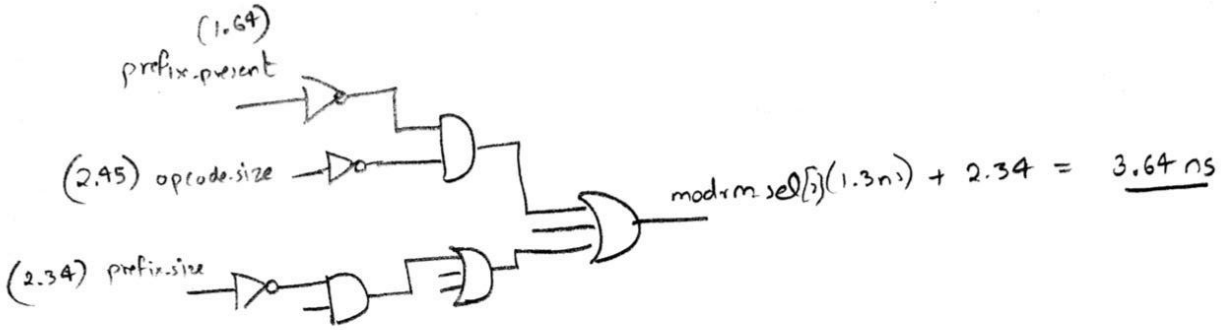
Opcode length decoder (Critical Path)

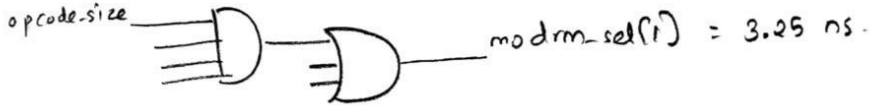


ModRM detector

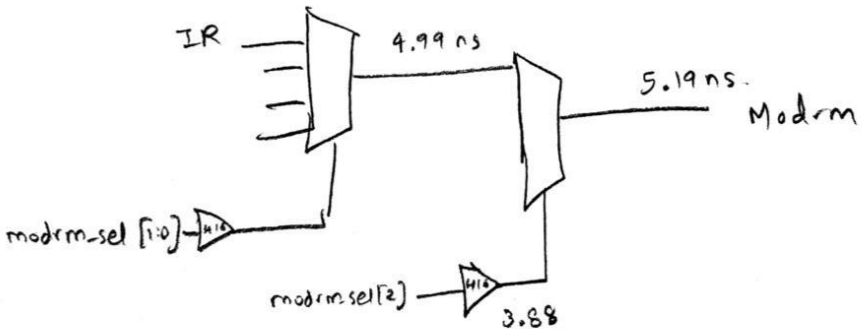


Modrm



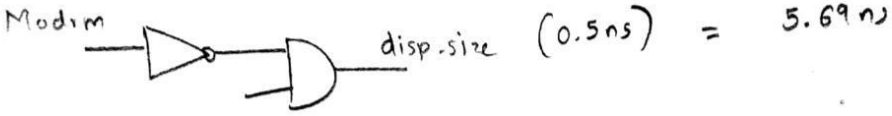


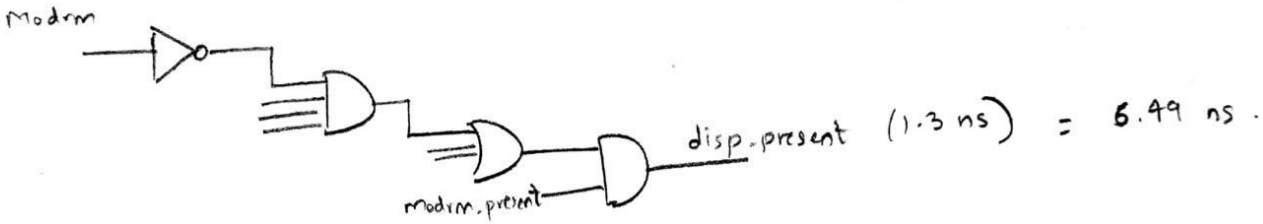
Modrm selection

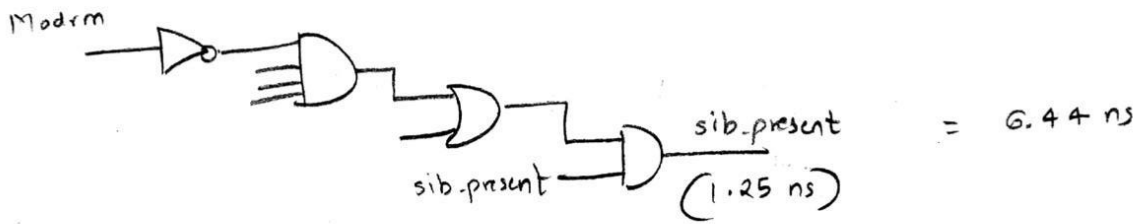


Sib_disp_detector (critical path)

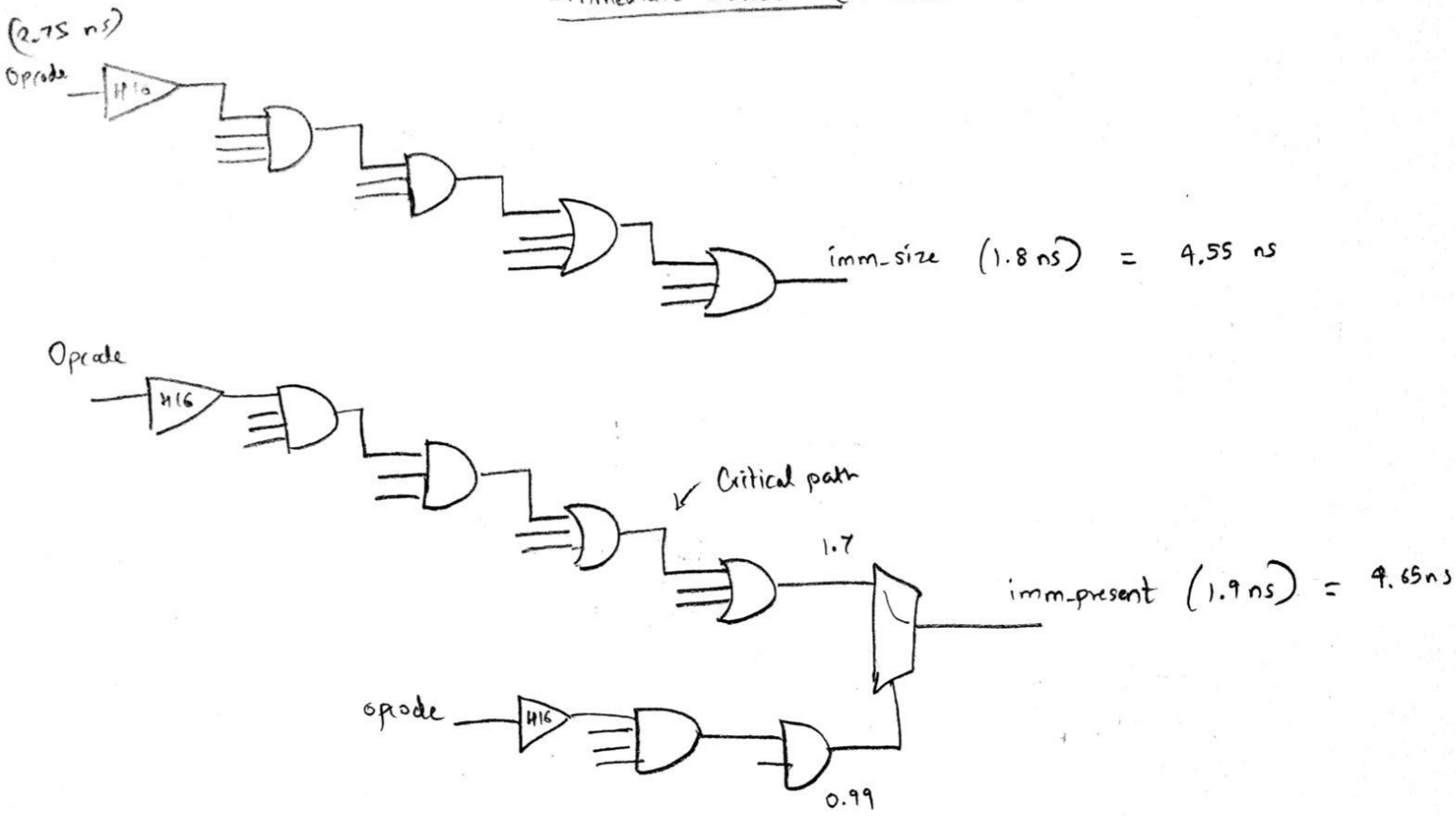
(5.19)



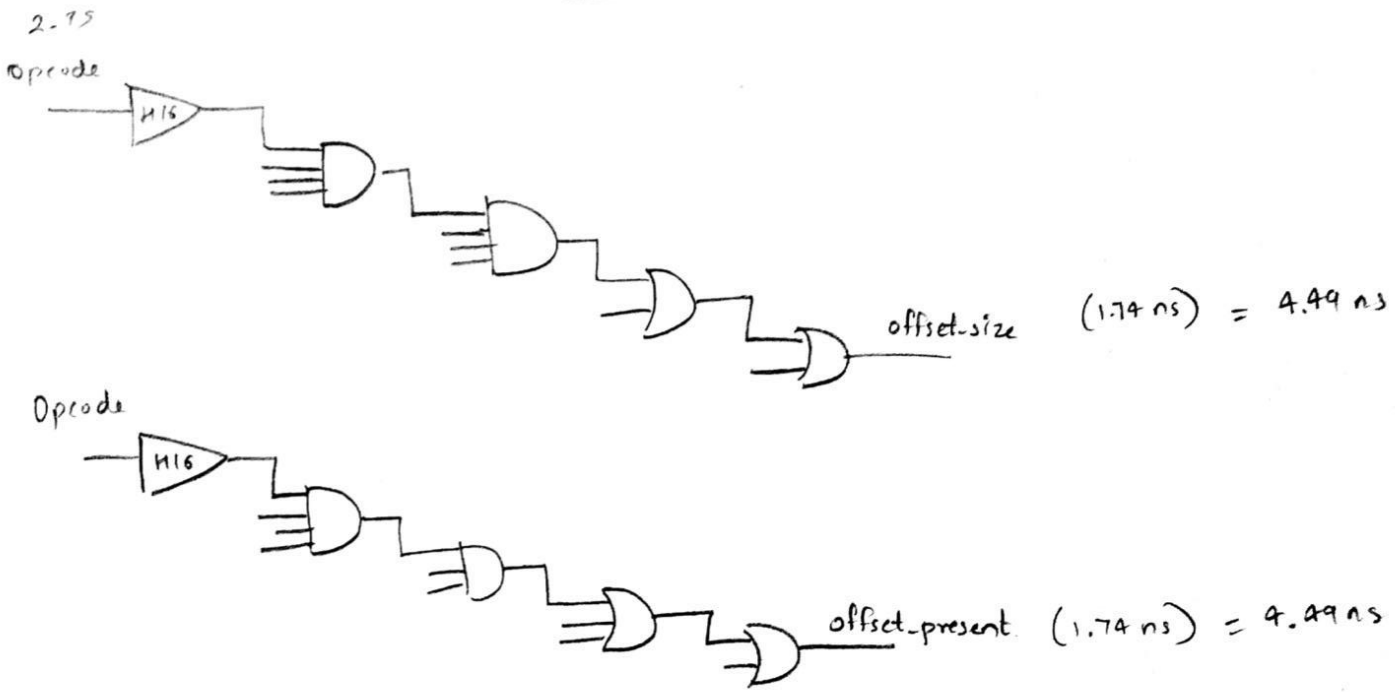




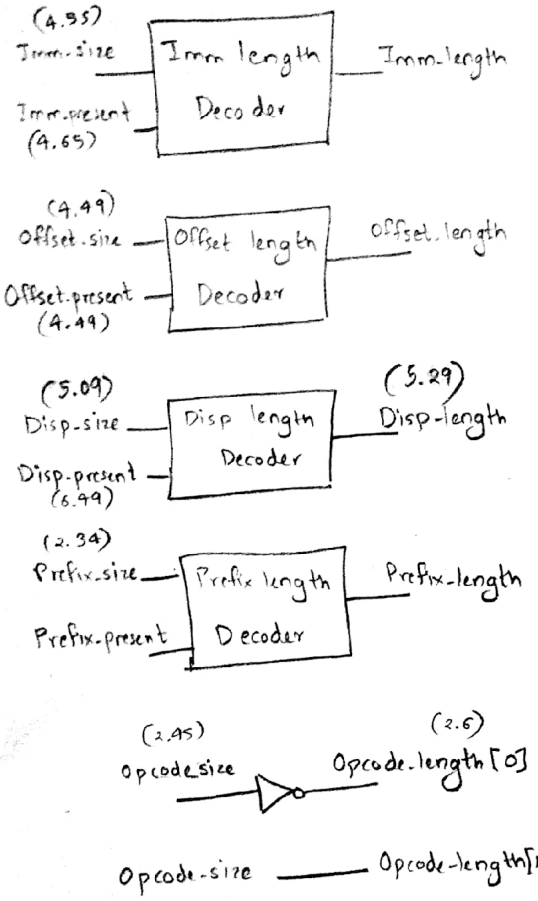
Immediate Detector (Critical Path)

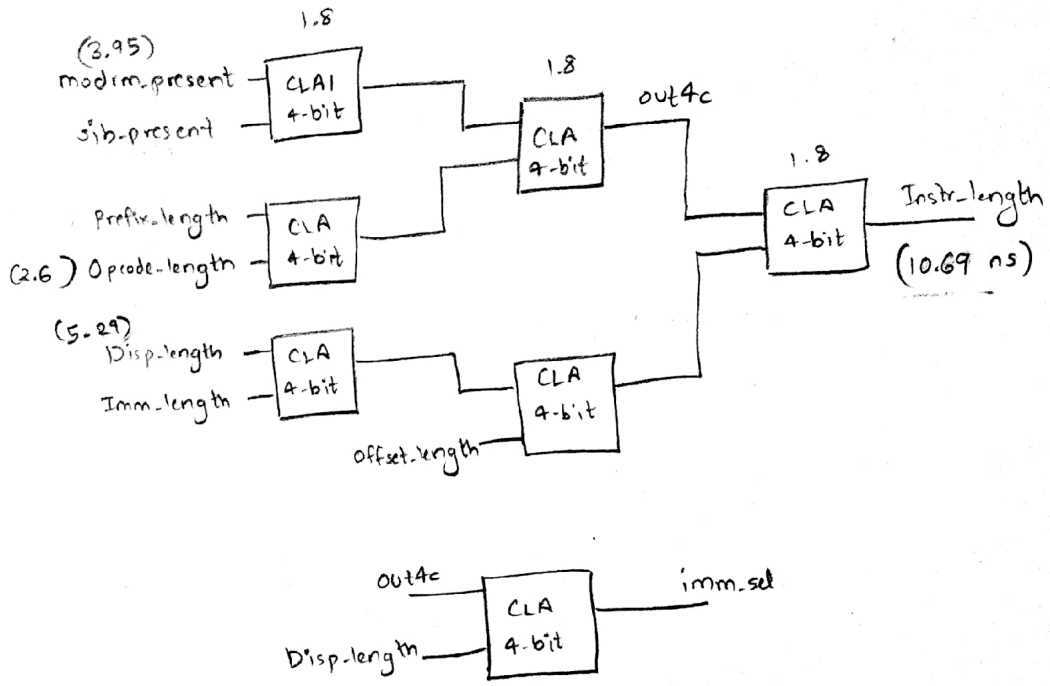


Offset Detector (critical path)



Instruction Length Calc





disp_sel = out4c [2:0]

Section 5: Final Design Decisions

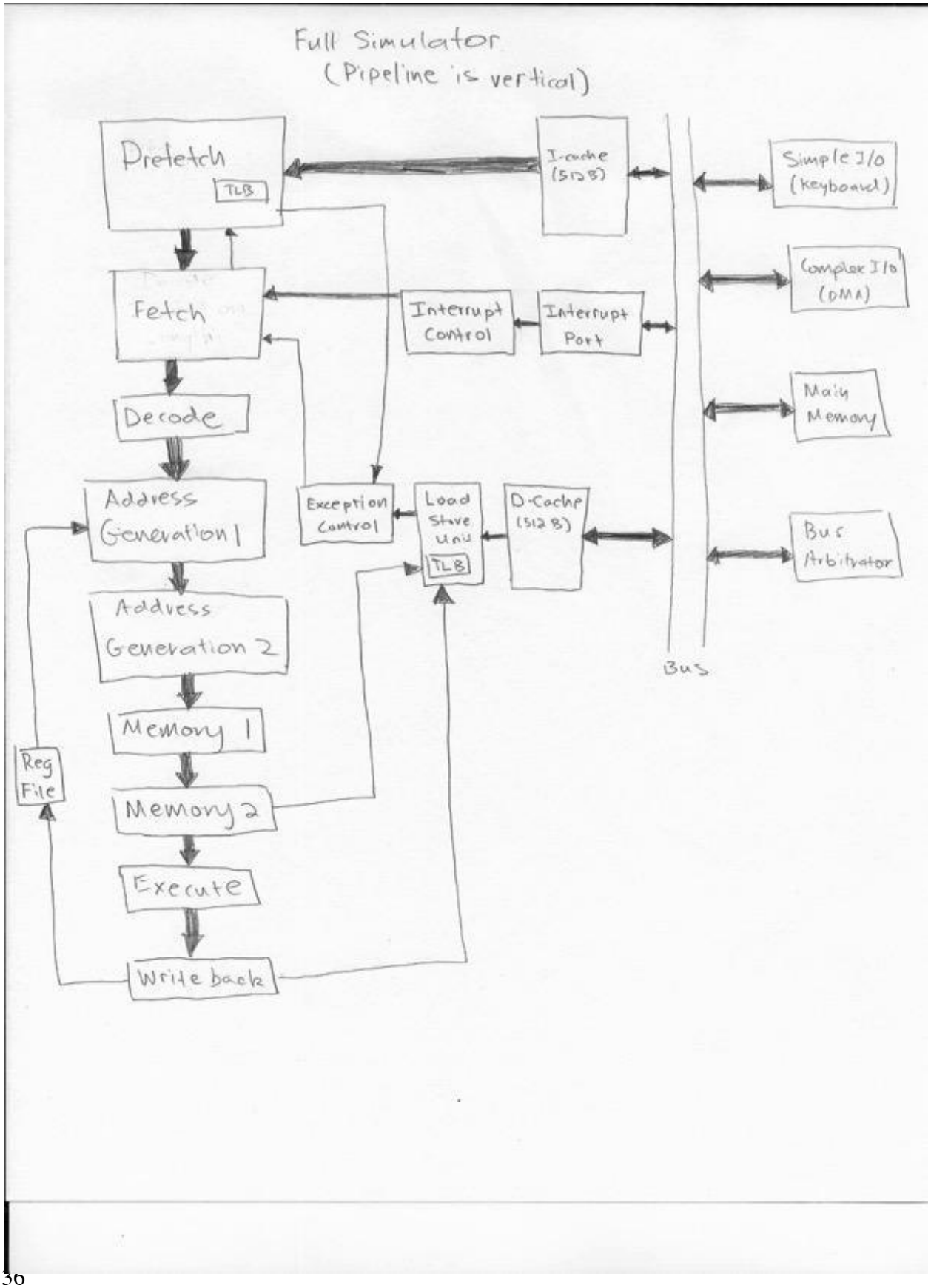
Most of our initial design did not change. We decided to implement dedicated structures for REPNE CMPS since they did not increase our critical path, which was in Fetch. This increased the throughput for repeats and avoided stalling on every iteration, waiting for the updated addresses to be stored back. Also, we implemented a simple not-taken branch predictor because most of the logic was already in place to handle flushing for interrupts and exceptions. Continuing to fetch on a conditional branch seemed natural and did not require any additional structures other than a signal indicating a conditional branch was taken instead of not taken.

Section 6: Conclusion

Our design was focused mainly on correctness and performance, and implementing such a design would be expensive in terms of money and power. We had time constraints in implementing our design, and it was difficult to implement complex design structures and get them properly verified. We had direct mapped caches for both instruction and data cache. A direct mapped cache is acceptable for an instruction cache because of the mostly sequential nature of instruction fetches, but we could improve our performance with a set associative data cache. Additionally, a read hit in the instruction or data cache took a minimum of three cycles, with data returned at the beginning of the third cycle. This is unnecessarily long and could have been reduced using only combinational logic on a hit to compare tags. The cache controller and state machine should only have been used for misses and evictions. As a a result, our performance suffered when running instructions needing to access memory. The ICACHE read performance impact was minimized by prefetching instructions into a large instruction buffer. Our recommendations for the next design would be to include a set associative cache, banking of memory to allow consecutive accesses for non-conflicting addresses, decrease the cache latency to get data faster, and having a MSHR for the data cache. A write buffer would also improve performance so that write stalls would not stall the whole pipeline and cause multiple reads of the same address. Finally, we could add a true branch predictor because the logic was already added to flush the pipeline on a taken branch. Instead, we would fetch from the target of the prediction and not the next sequential instruction after a conditional branch.

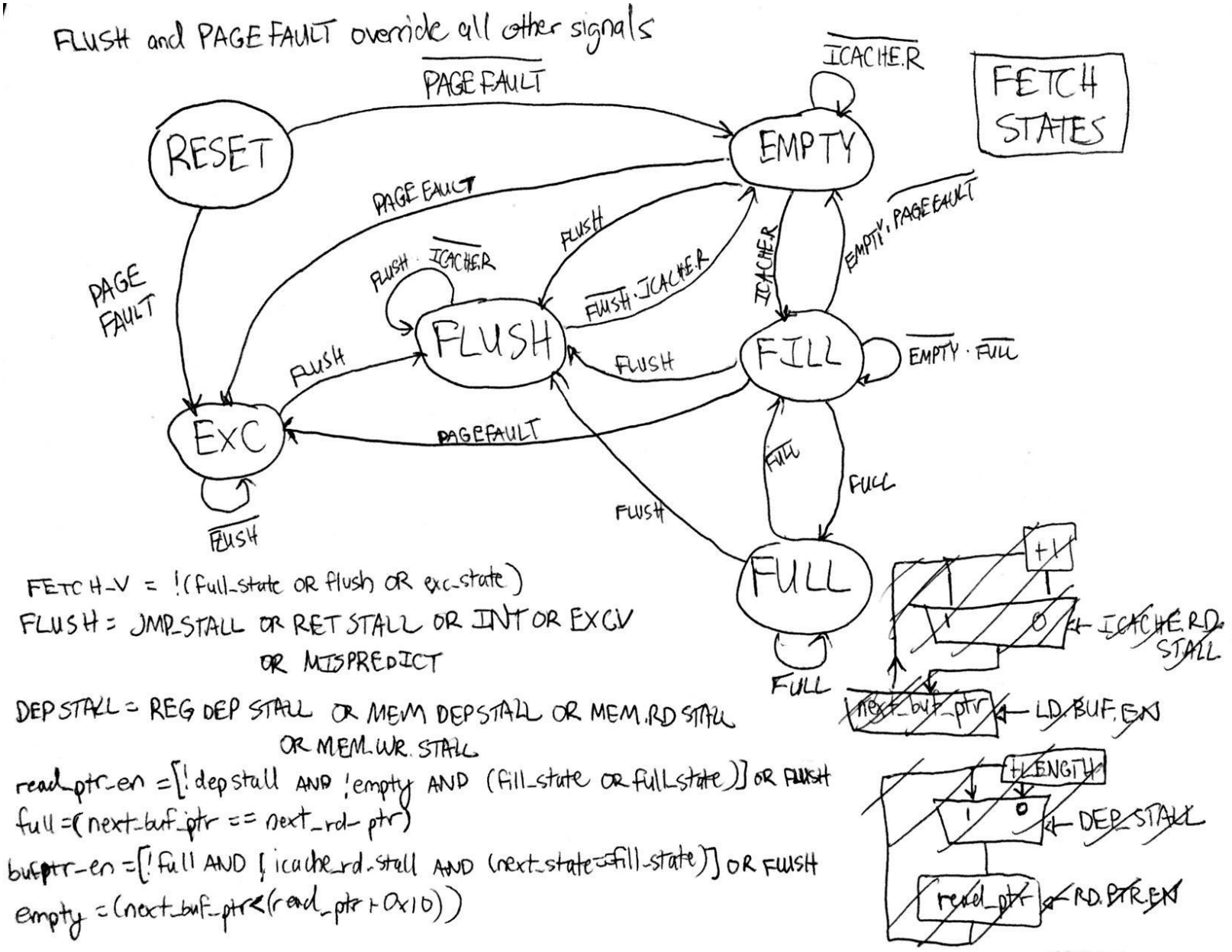
Appendix

Appendix A: High Level Data Path

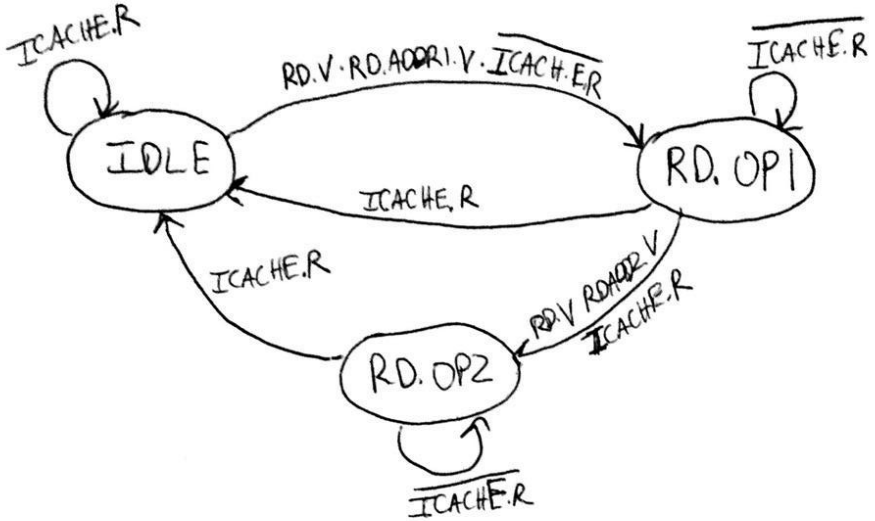

Full Simulator
(Pipeline is vertical)
Prefetch
TLB
I-cache
(512B)
Simple I/O
(keyboard)
Fetch
Interrupt
Control
Interrupt
Port
Complex I/O
(DMA)
Decode
Main
Memory
Address
Generation 1
Exception
Control
Load
Store
Unit
TLB
D-Cache
(512B)
Bus
Arbitrator
Address
Generation 2
Memory 1
Reg
File
Memory 2
Execute
Write back
Bus

Appendix B: State Diagram

Fetch State Diagram



IFU State Diagram



IFU STATES

load_buffer = (buf_ptr == buf_idx AND ! ICACHE.RD.STALL AND ((next_state == fill_state AND (current_state == flush_state OR empty_state)) OR (current_state == fill_state AND next_state == full_state))

ITLB SAME AS DTLB LOOKUP

LSU State Diagram

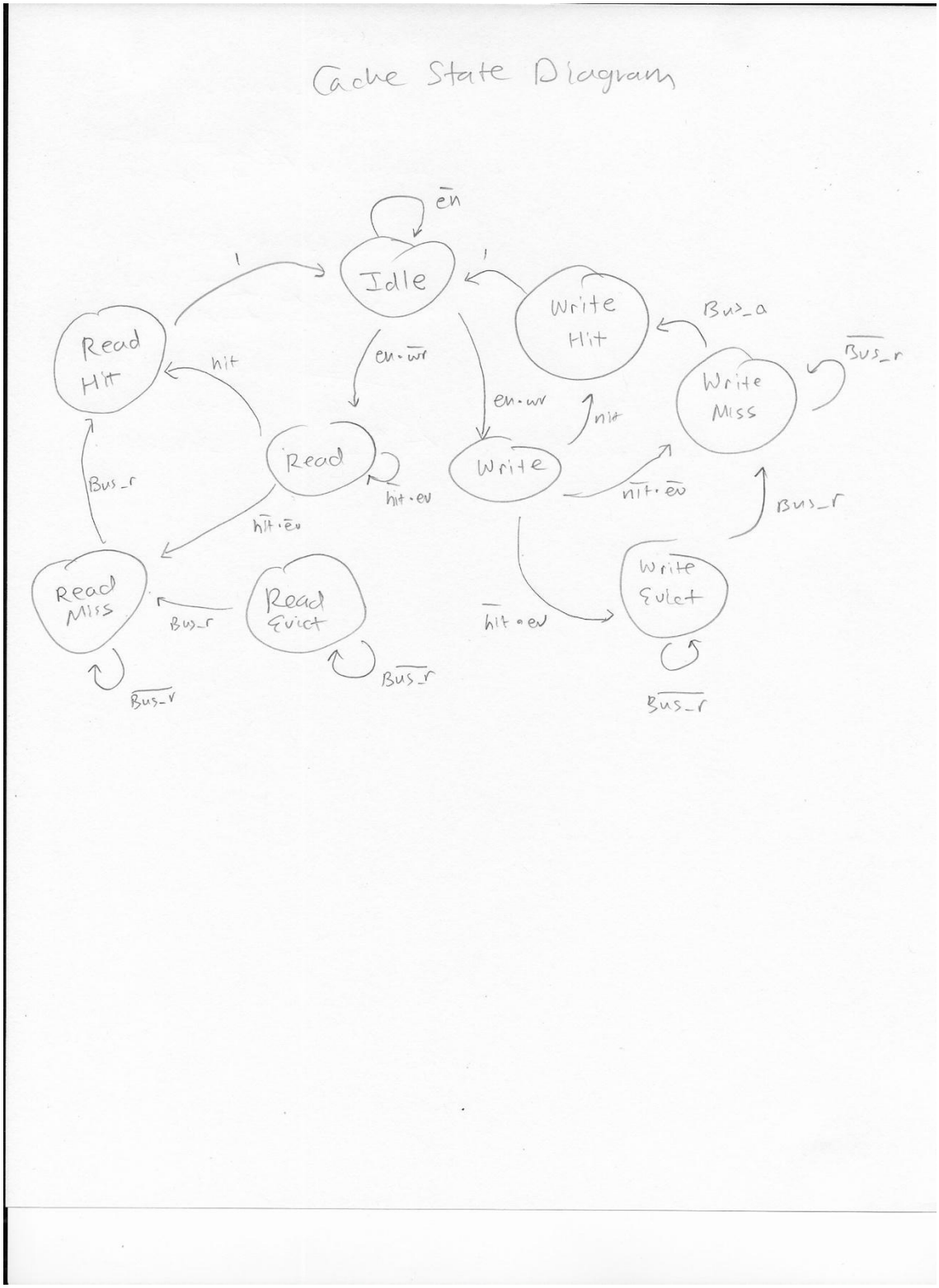
RD STALL IF WR IN PROGRESS
OR RD.OP1 OR 2 STATE

WR STALL IF RD IN PROGRESS
OR WROP 1 OR 2 STATE

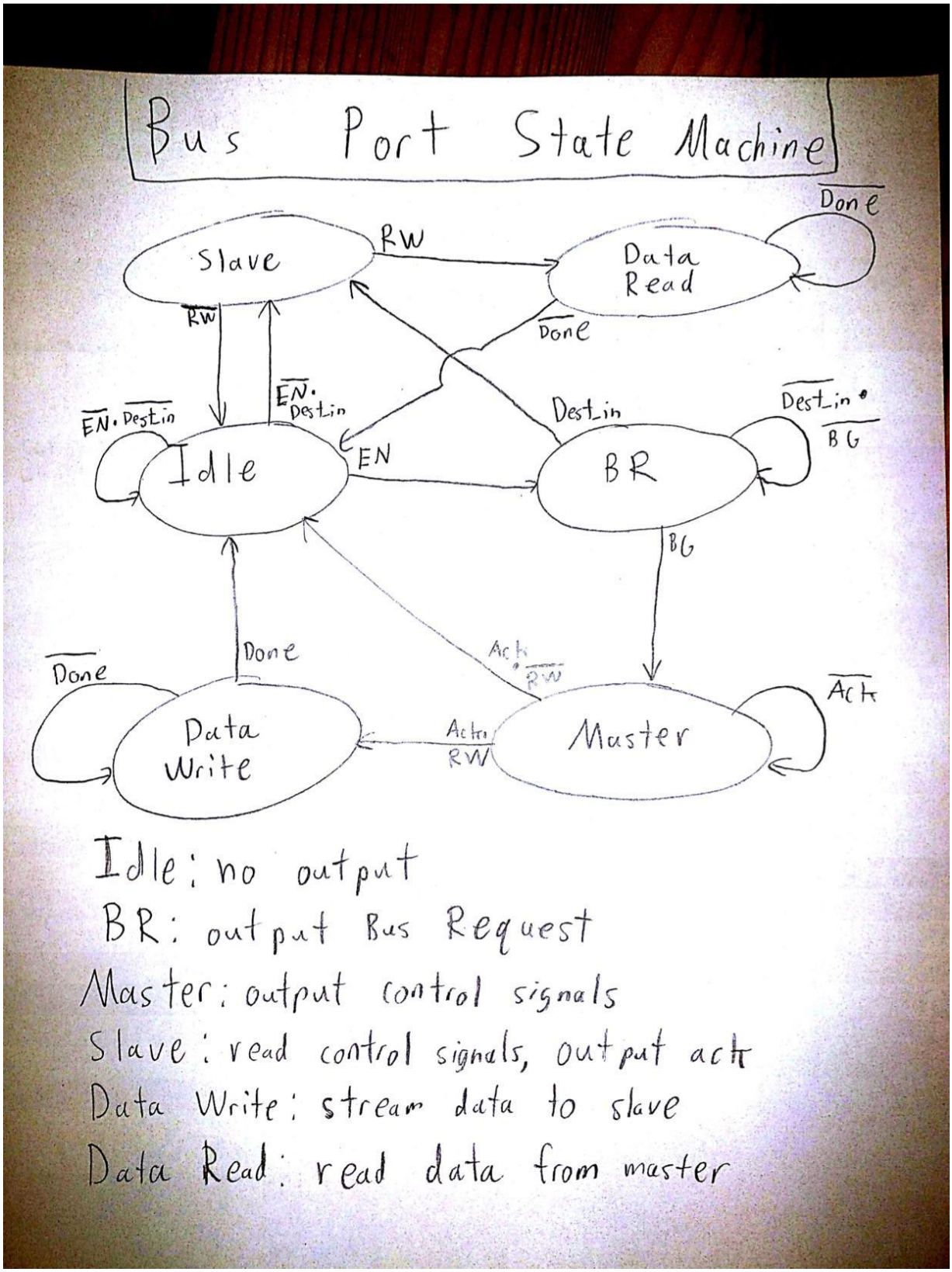
WR PRIORITIZED OVER RD

ADDRESSES PASSED FROM ME2 AND WB
AND SIZES

Cache State Diagram

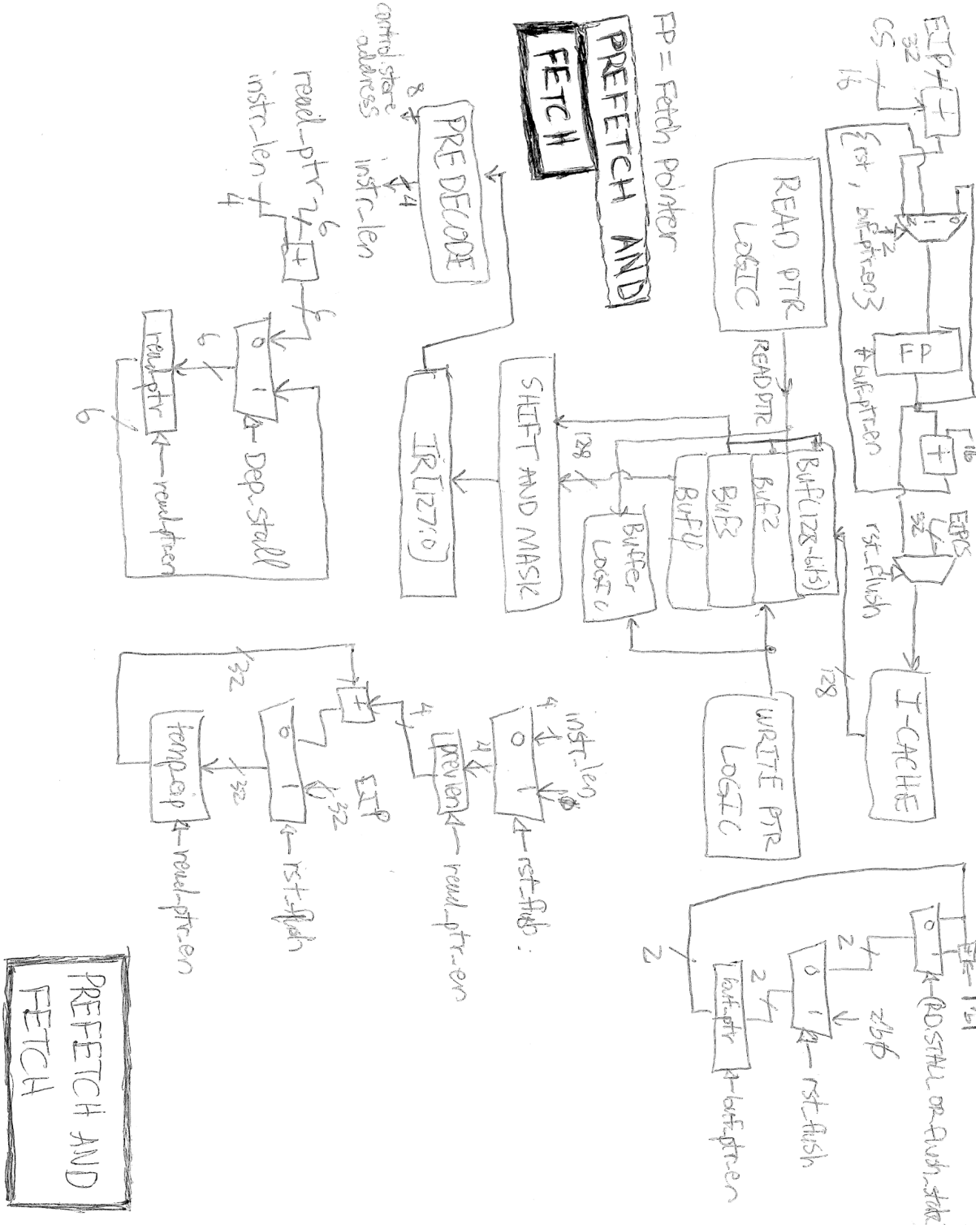


Bus Controller State Diagram

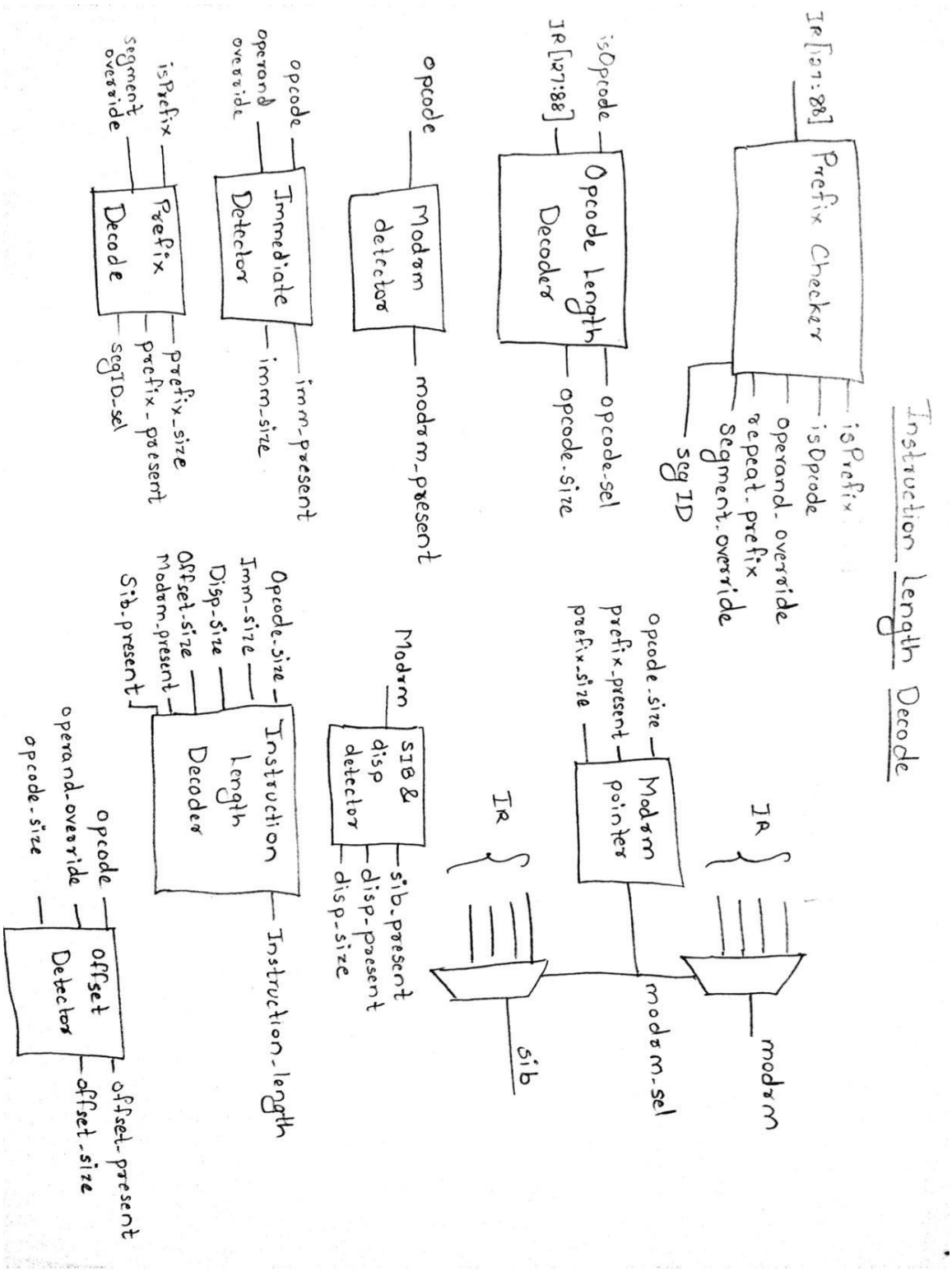

Bus Port State Machine
Slave
RW
Data Read
Done
RW
Done
EN · Dest_in
EN · Dest_in
Dest_in
Dest_in · BG
Idle
EN
BR
BG
Done
Done
Ack · RW
Ack
Data Write
Ack · RW
Master
Idle: no output
BR: output Bus Request
Master: output control signals
Slave: read control signals, output ack
Data Write: stream data to slave
Data Read: read data from master

Appendix C: Pipeline Schematics

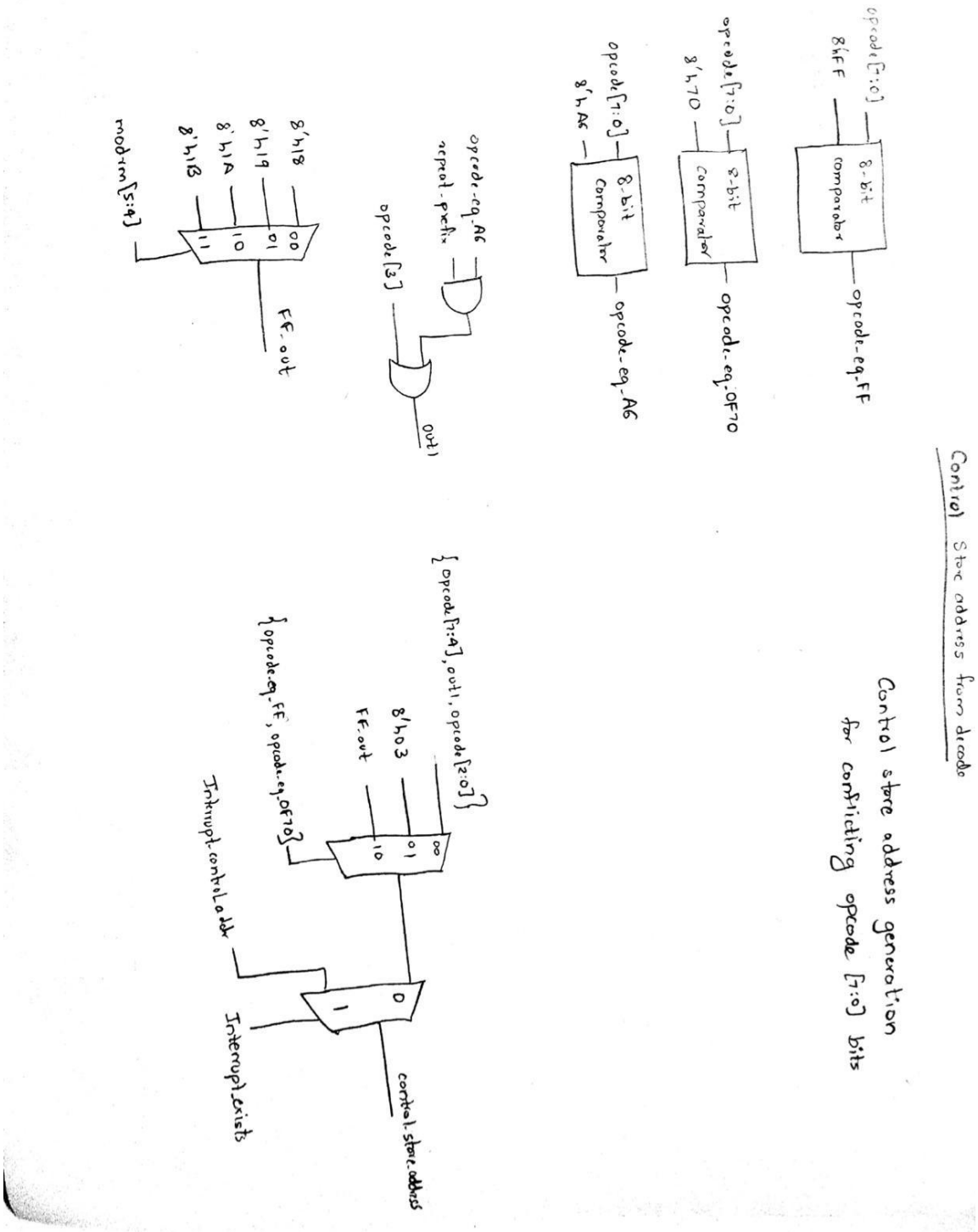
Prefetch and Fetch Stage Schematic



Instruction Length Decode



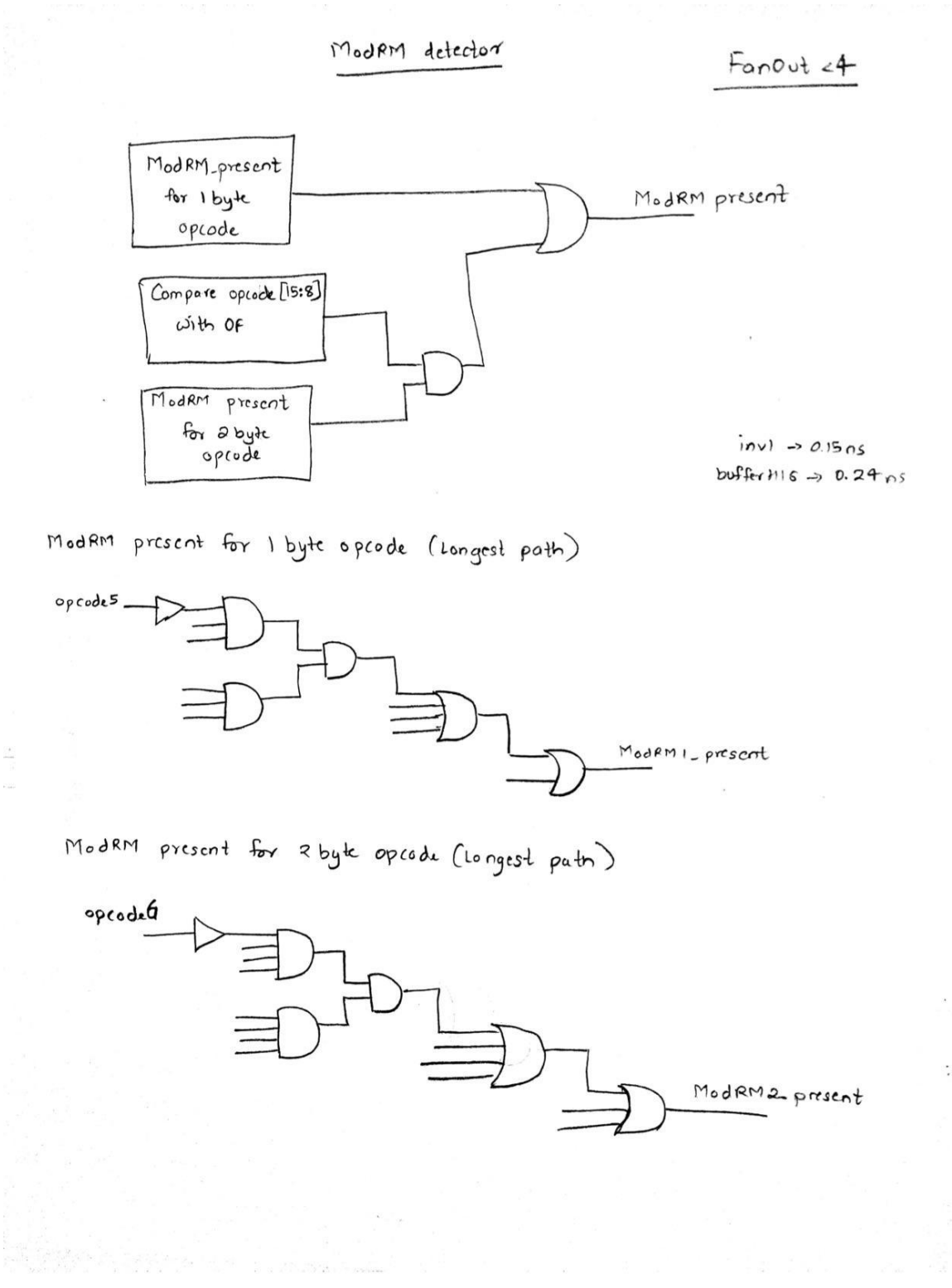
Control Store Address from Decode



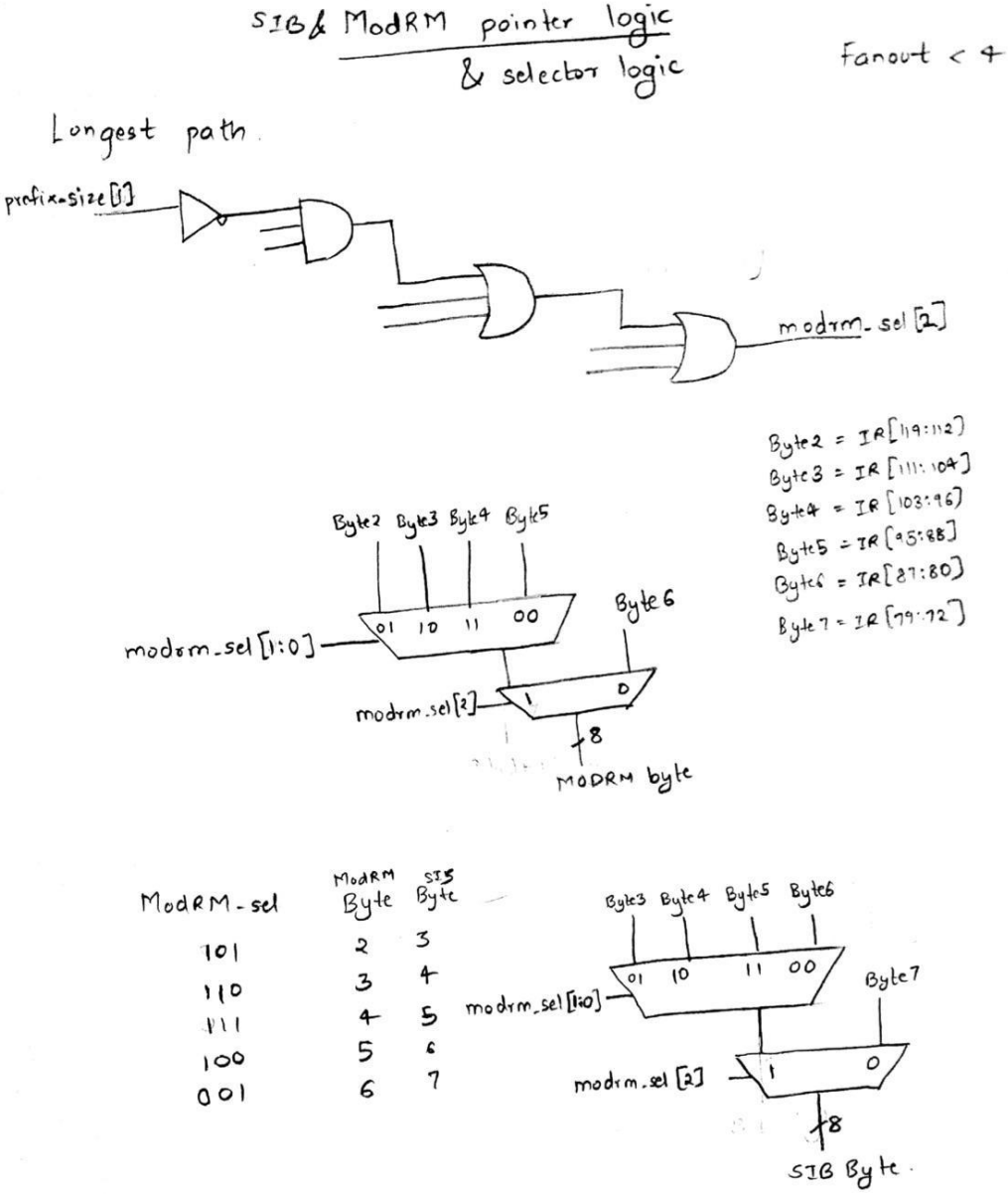
Prefix Checker and Decoder

Opcode Length Decoder

Modrm Detector



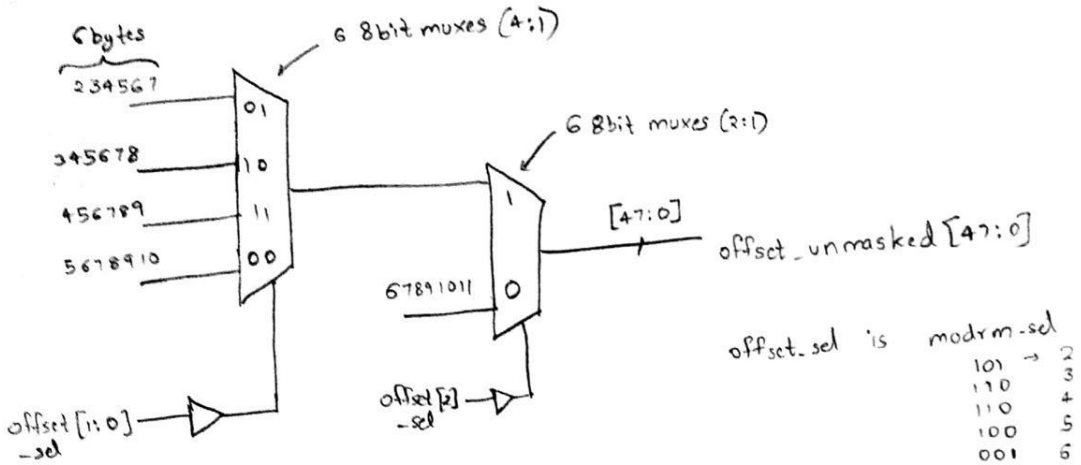
Modrm Pointer Logic



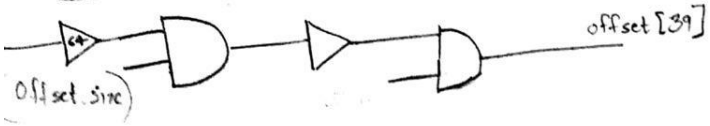
Offset Selector

Offset Selector

Fanout <4

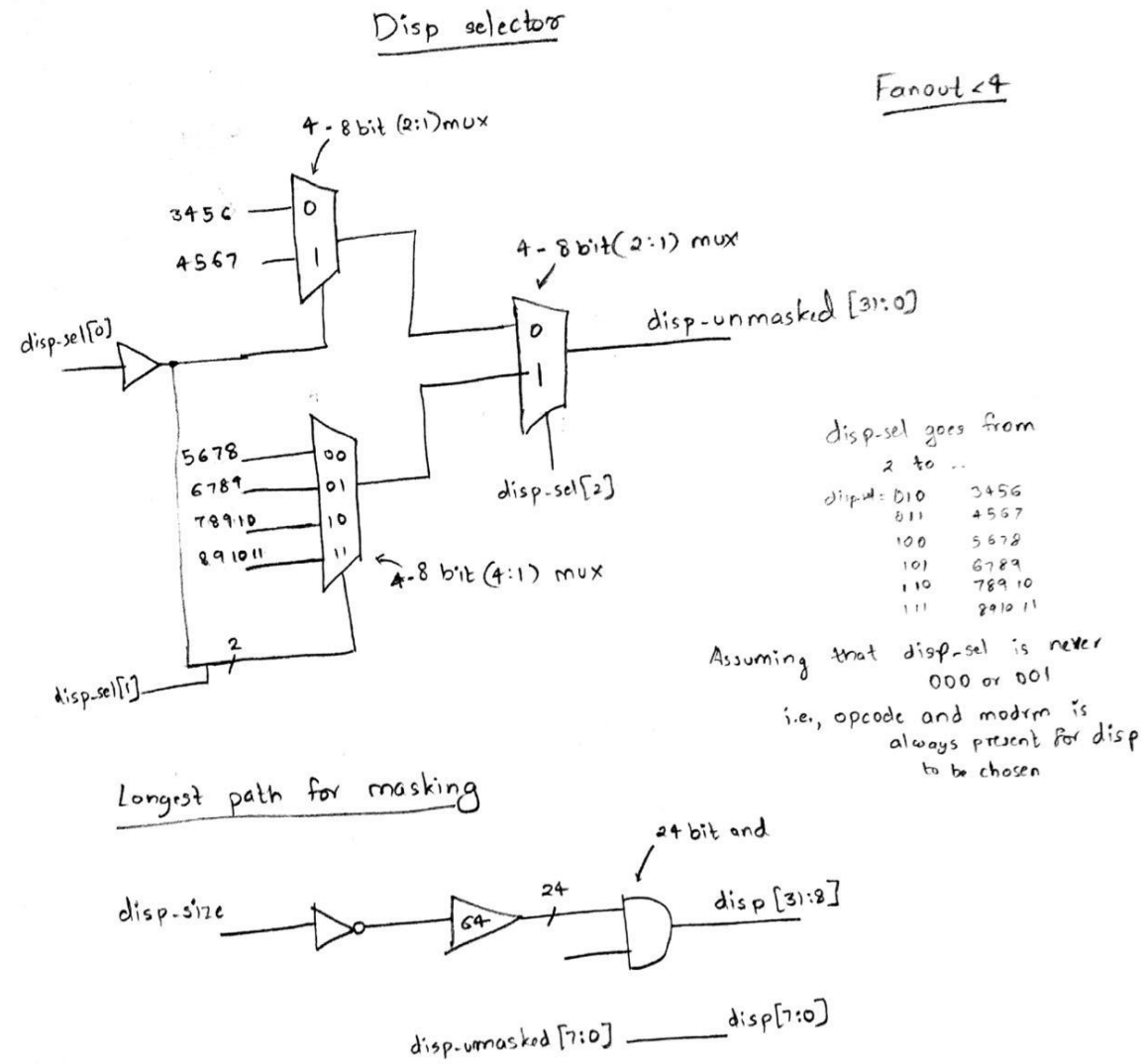


Longest path for masking



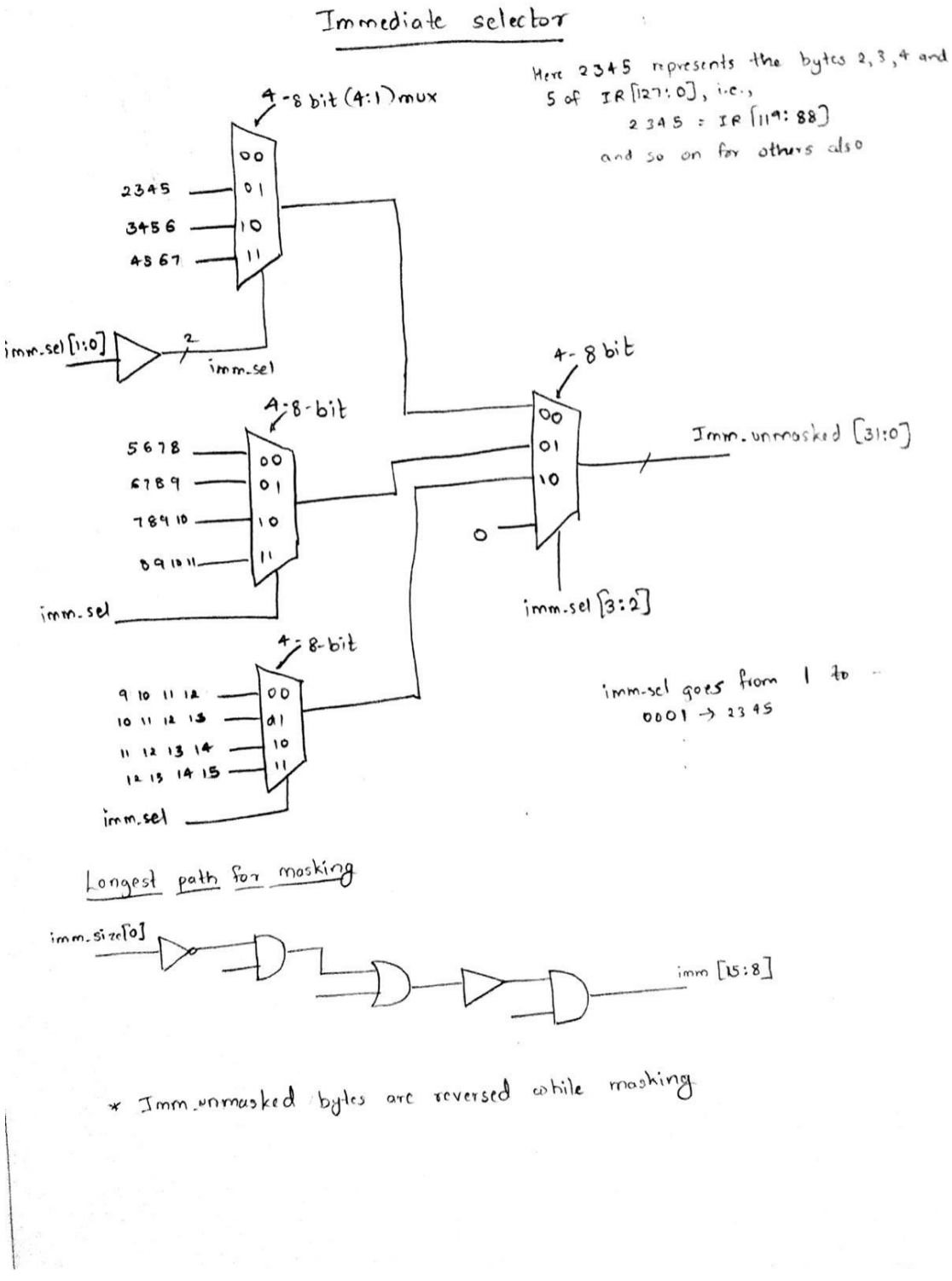
* The bytes are reversed while putting into offset value.
* Unless specified all buffers are buffer H16.

Disp Selector



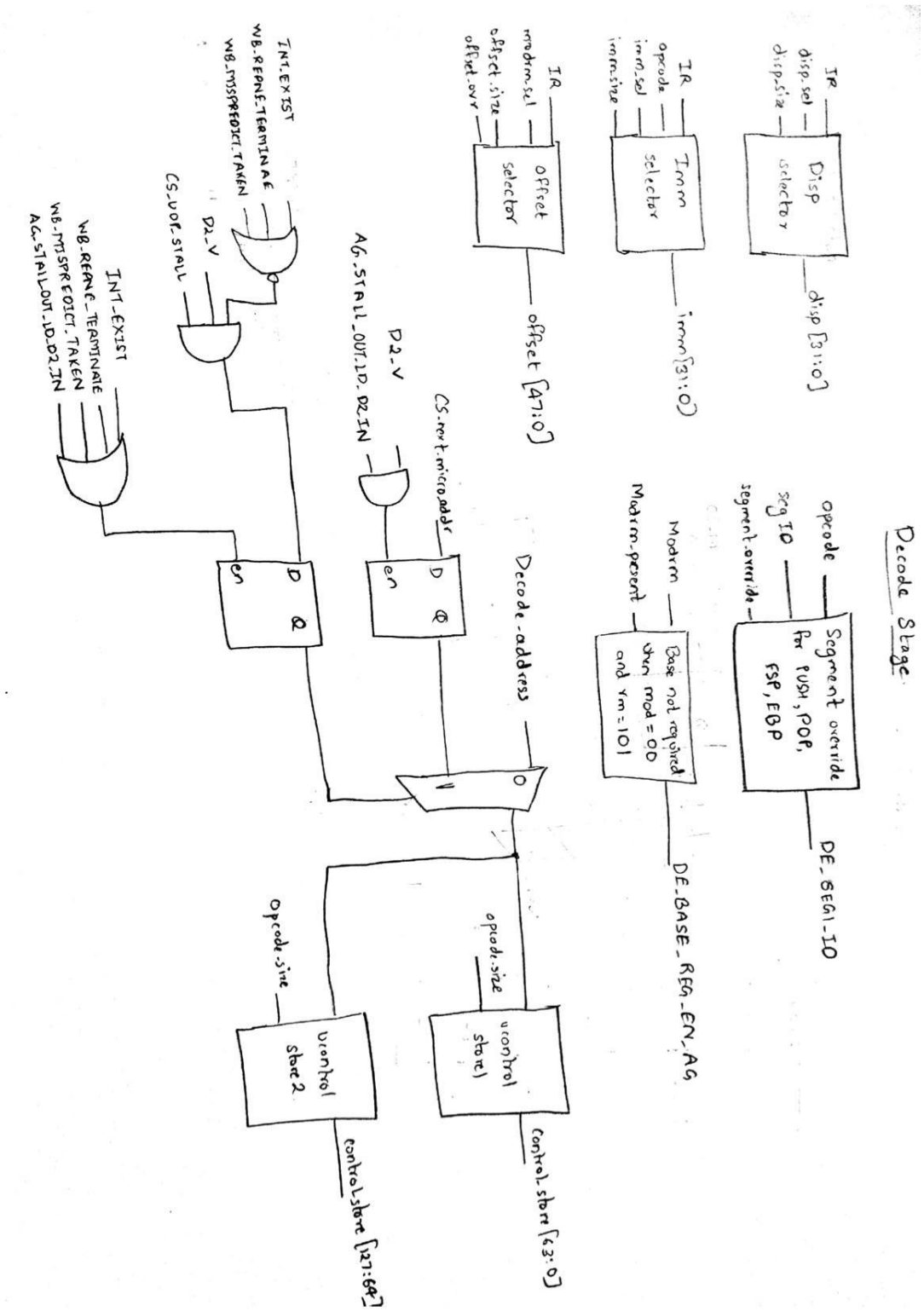
* Displacement bytes are reversed while masking

Immediate Selector

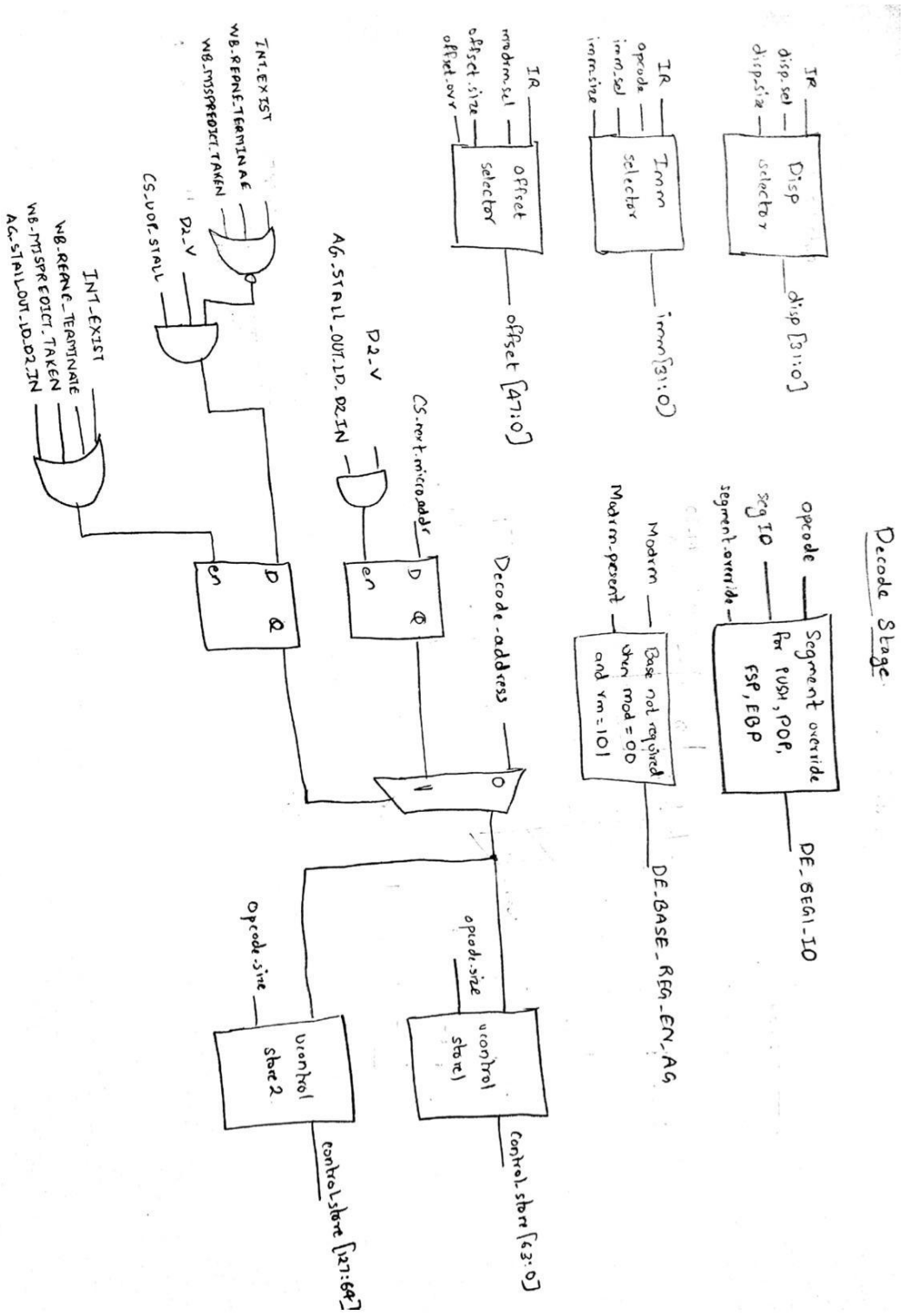


Decode Stage

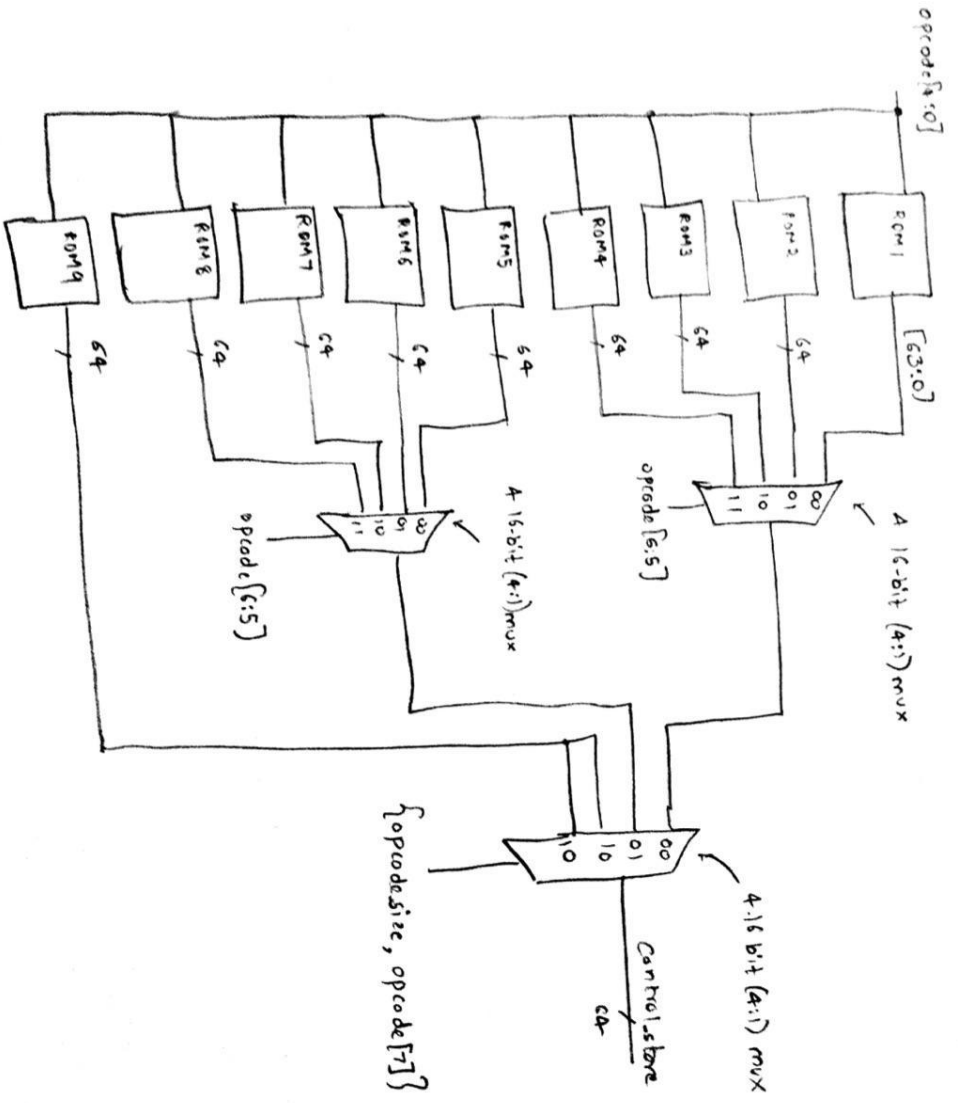
Decoder



Decode Stage



uControl Store

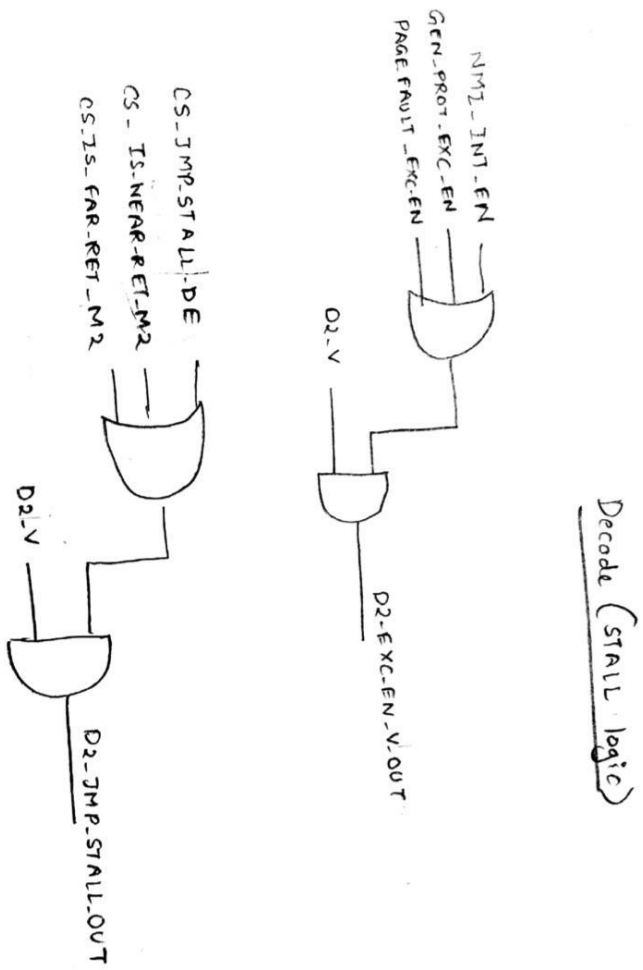


Scanned by CamScanner

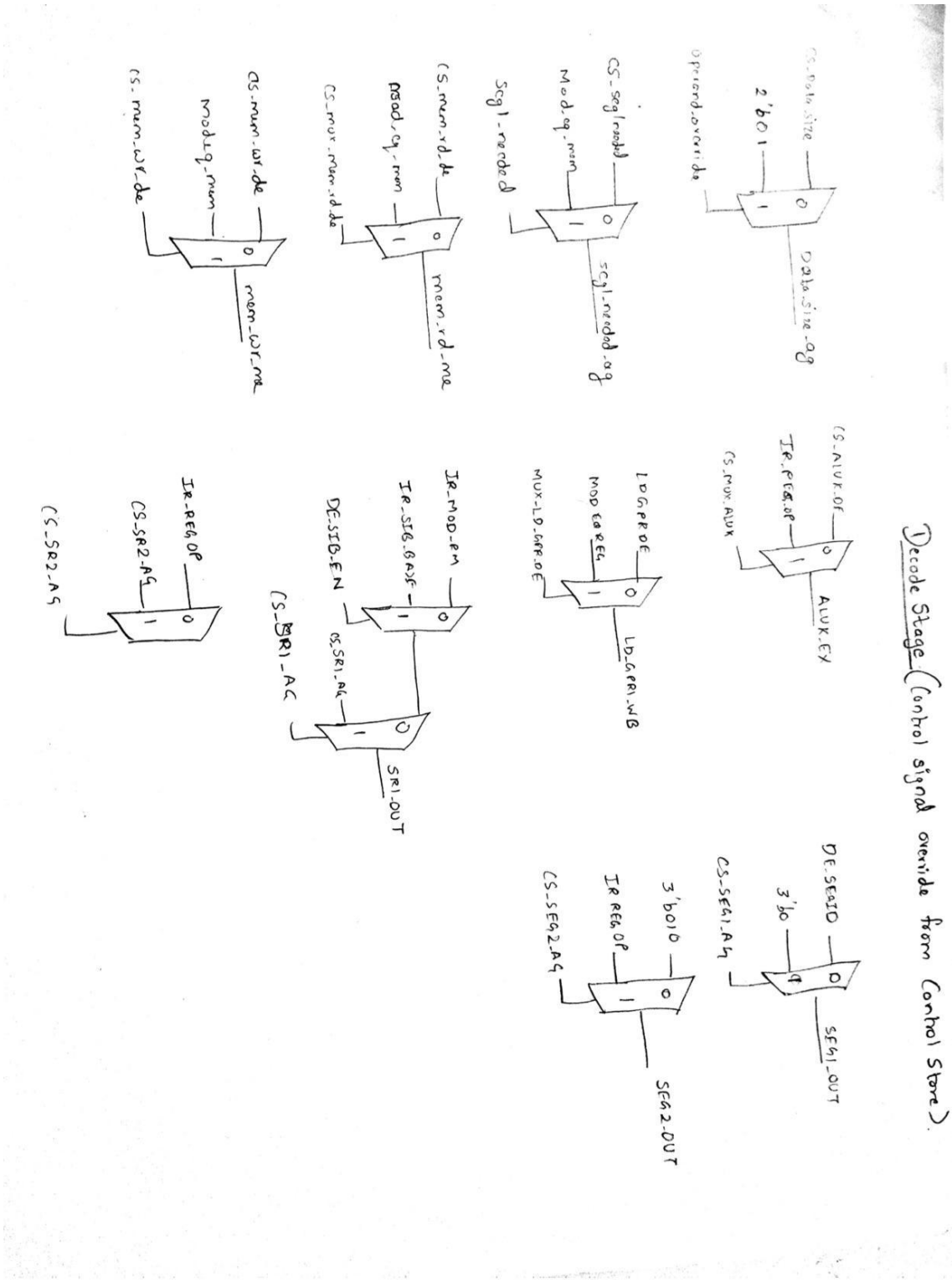
Segment Override

Segment override DE2.

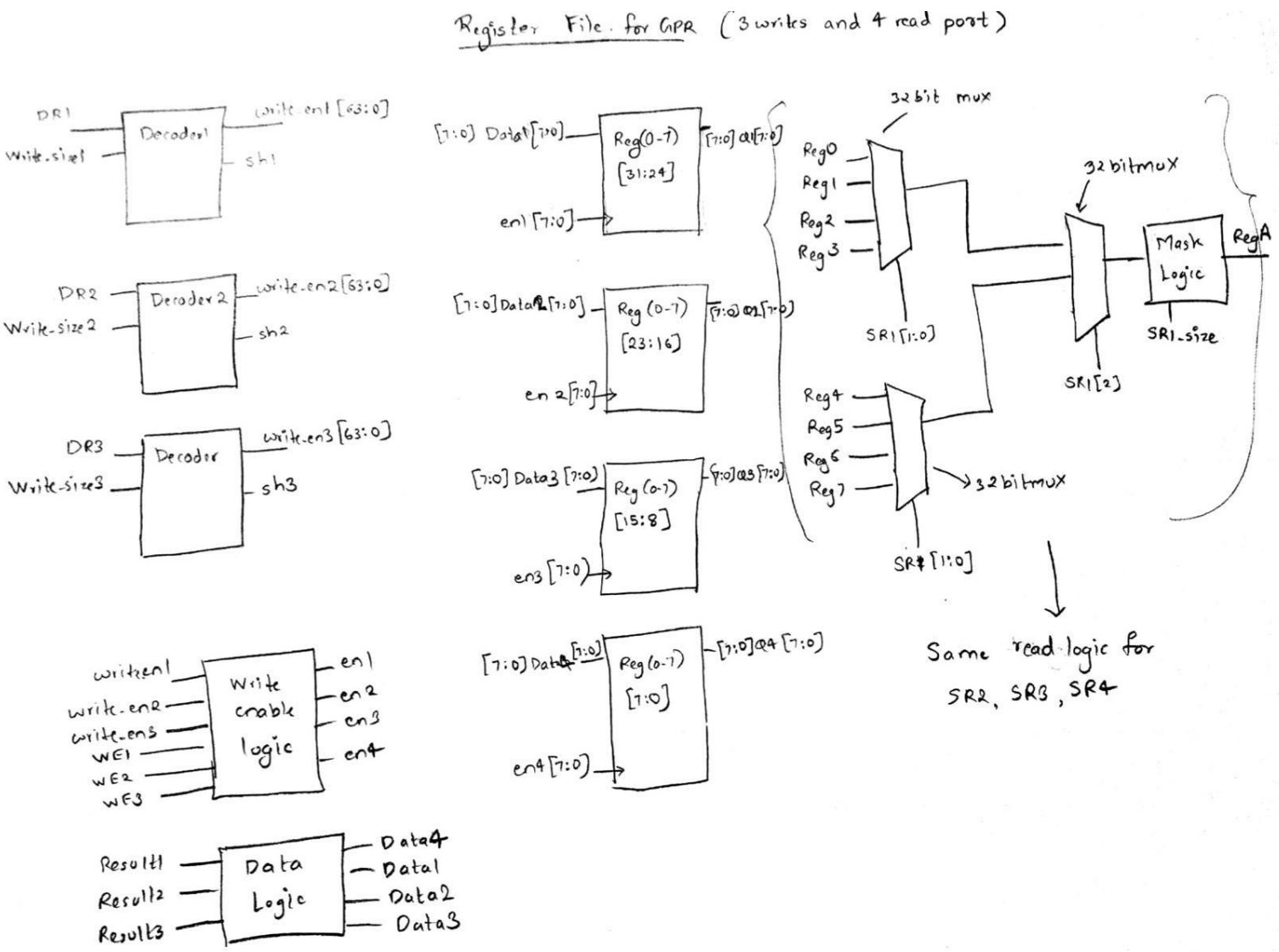
Decoder (Stall Logic)



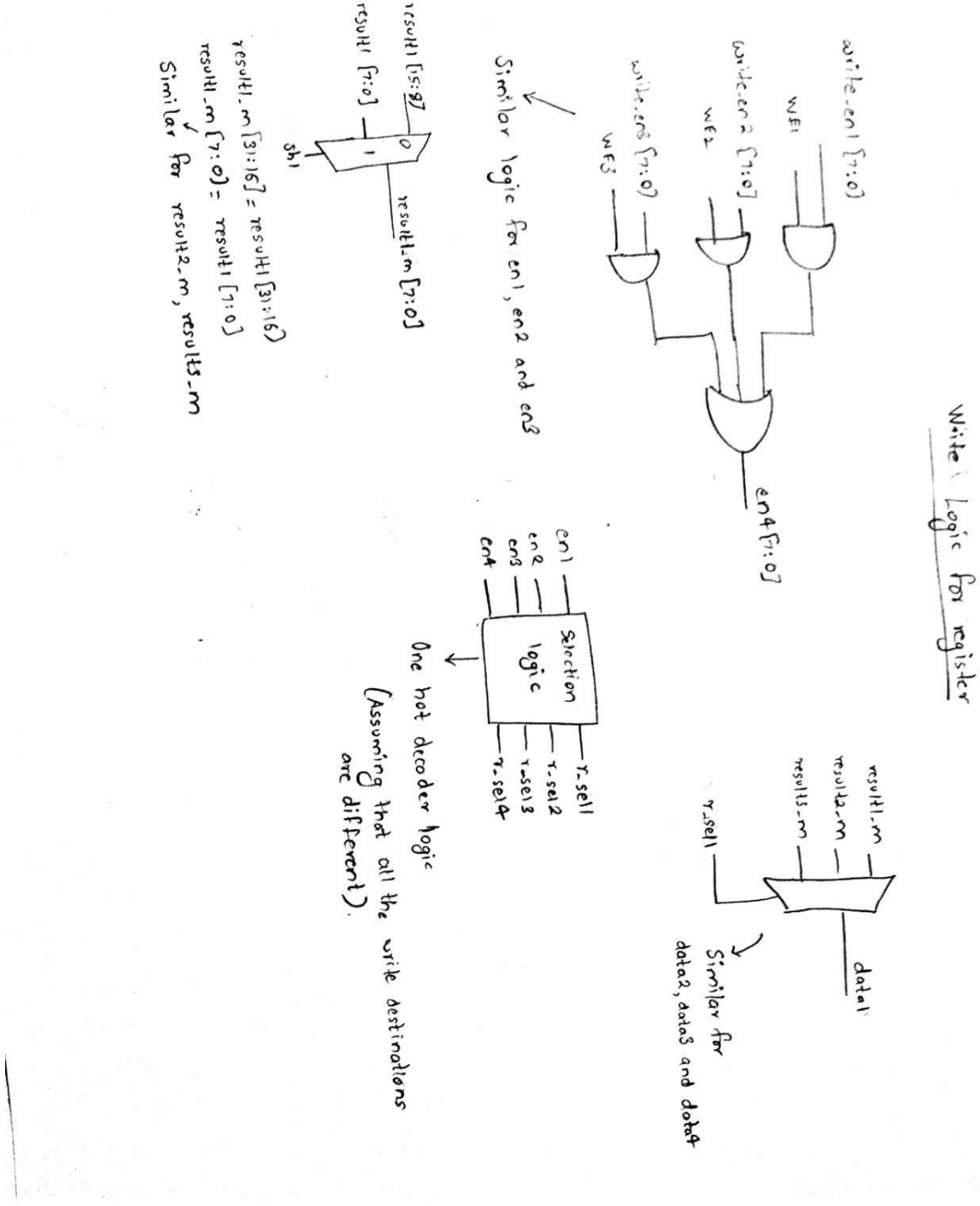
Control Store Override



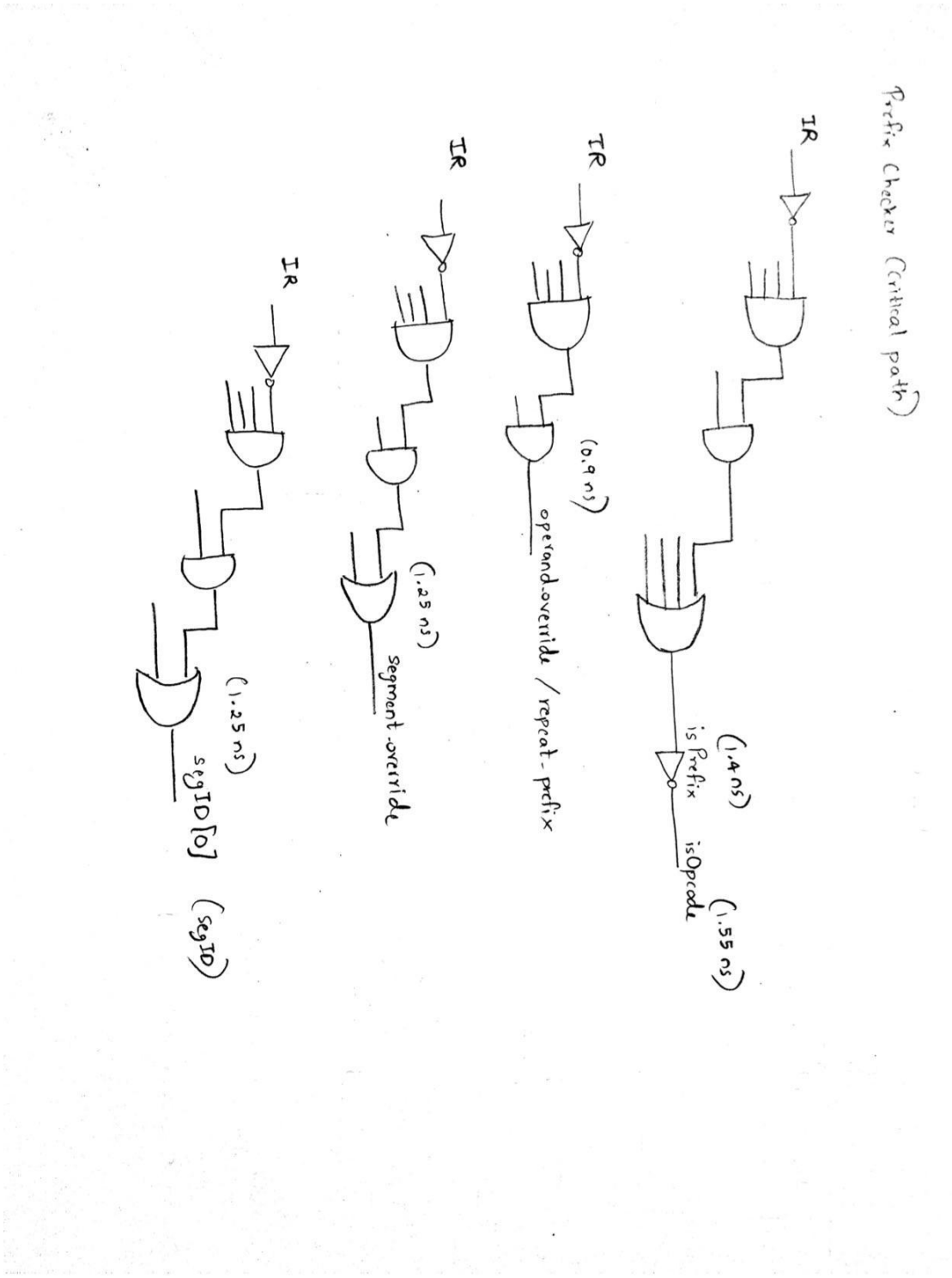
Register File



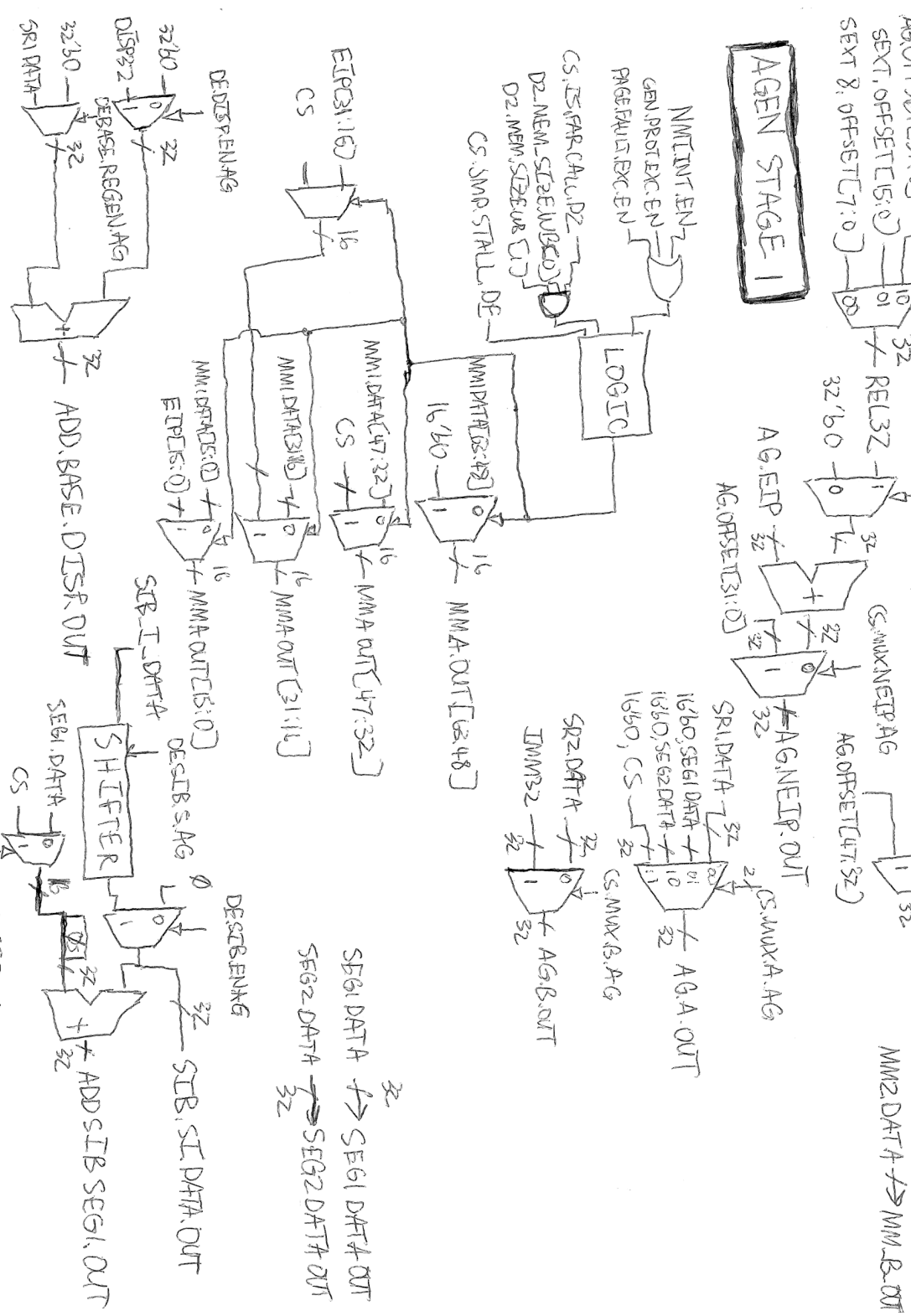
Write Logic for Register



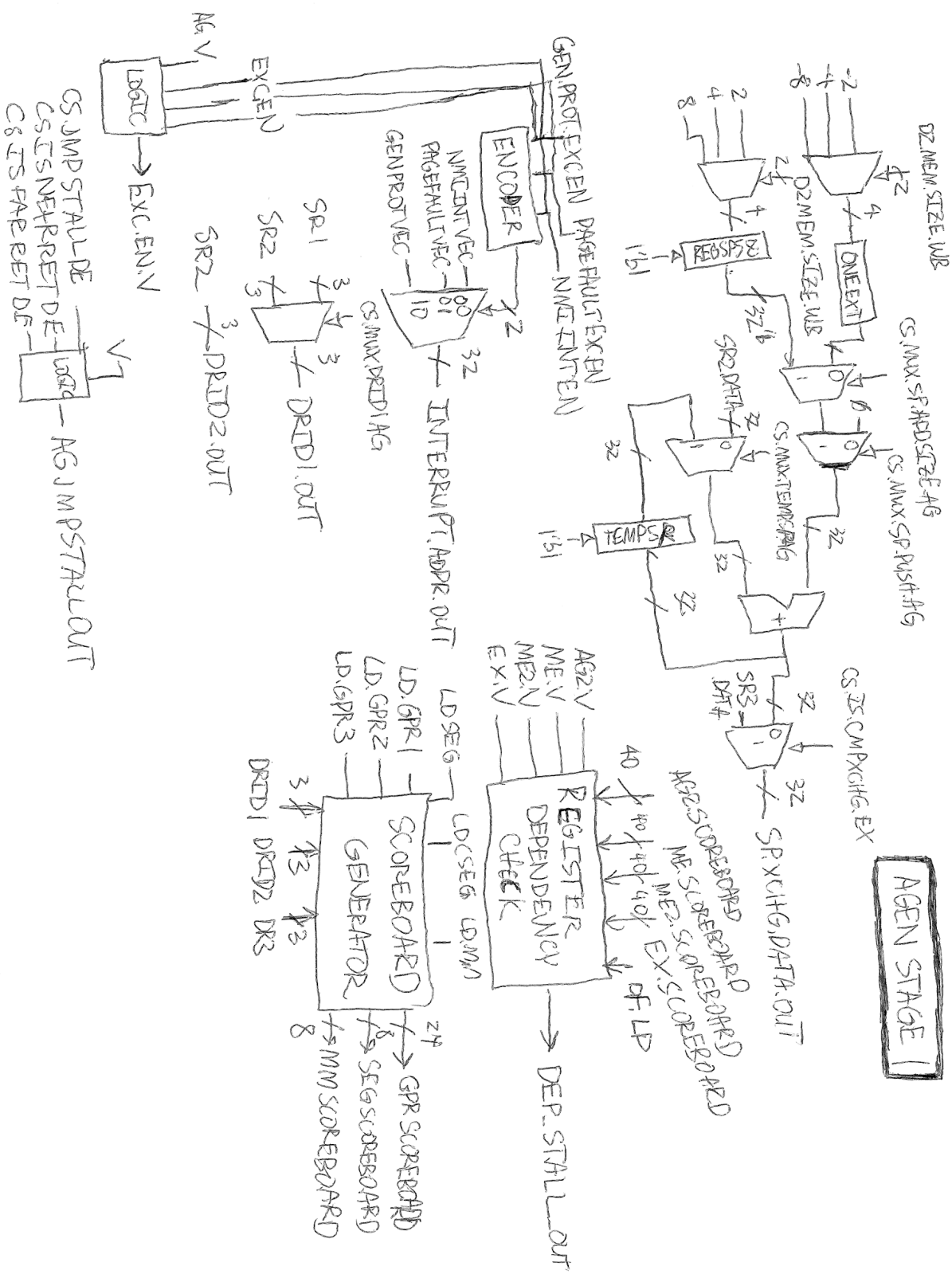
Prefix Checker



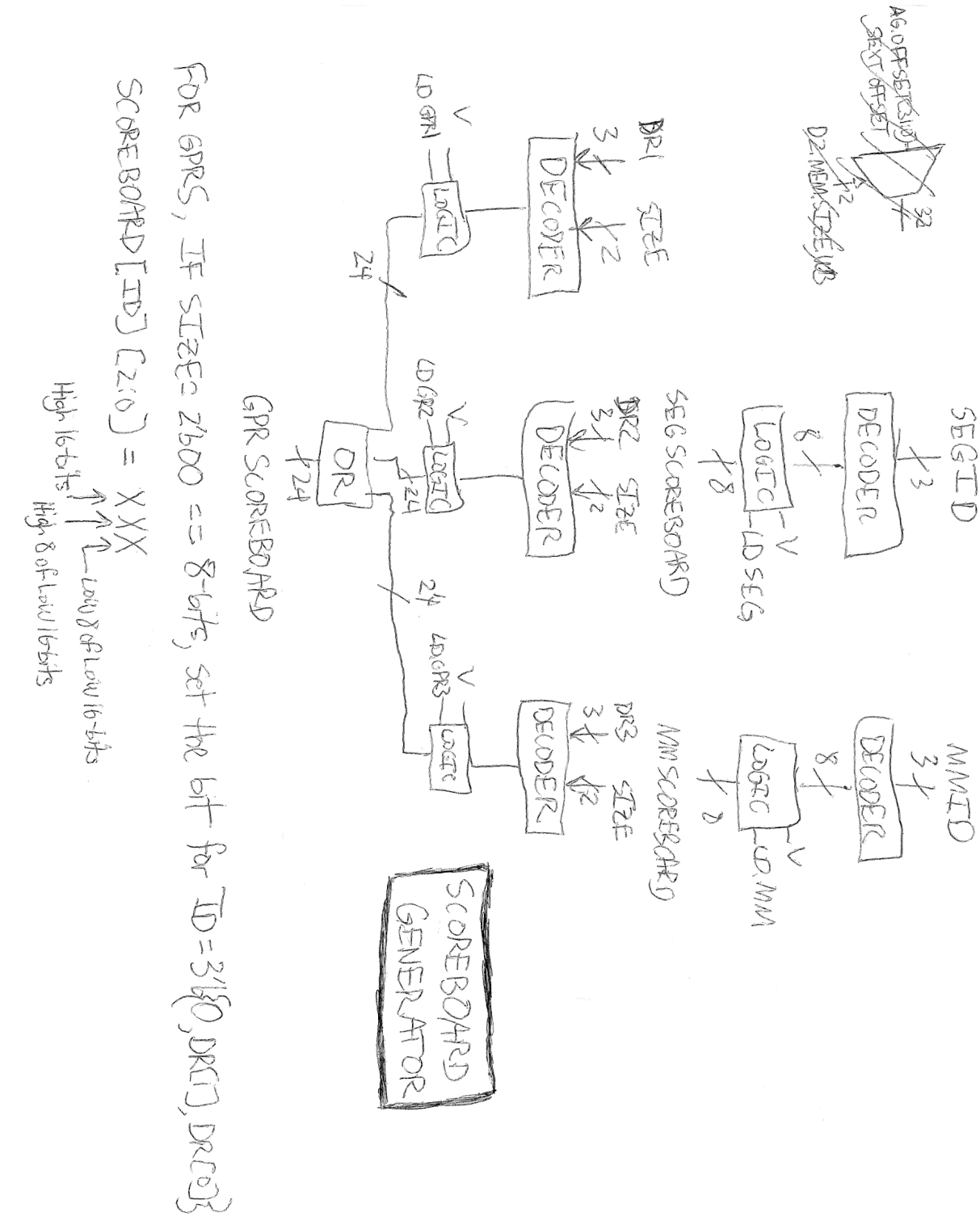
Address Generation Stage 1



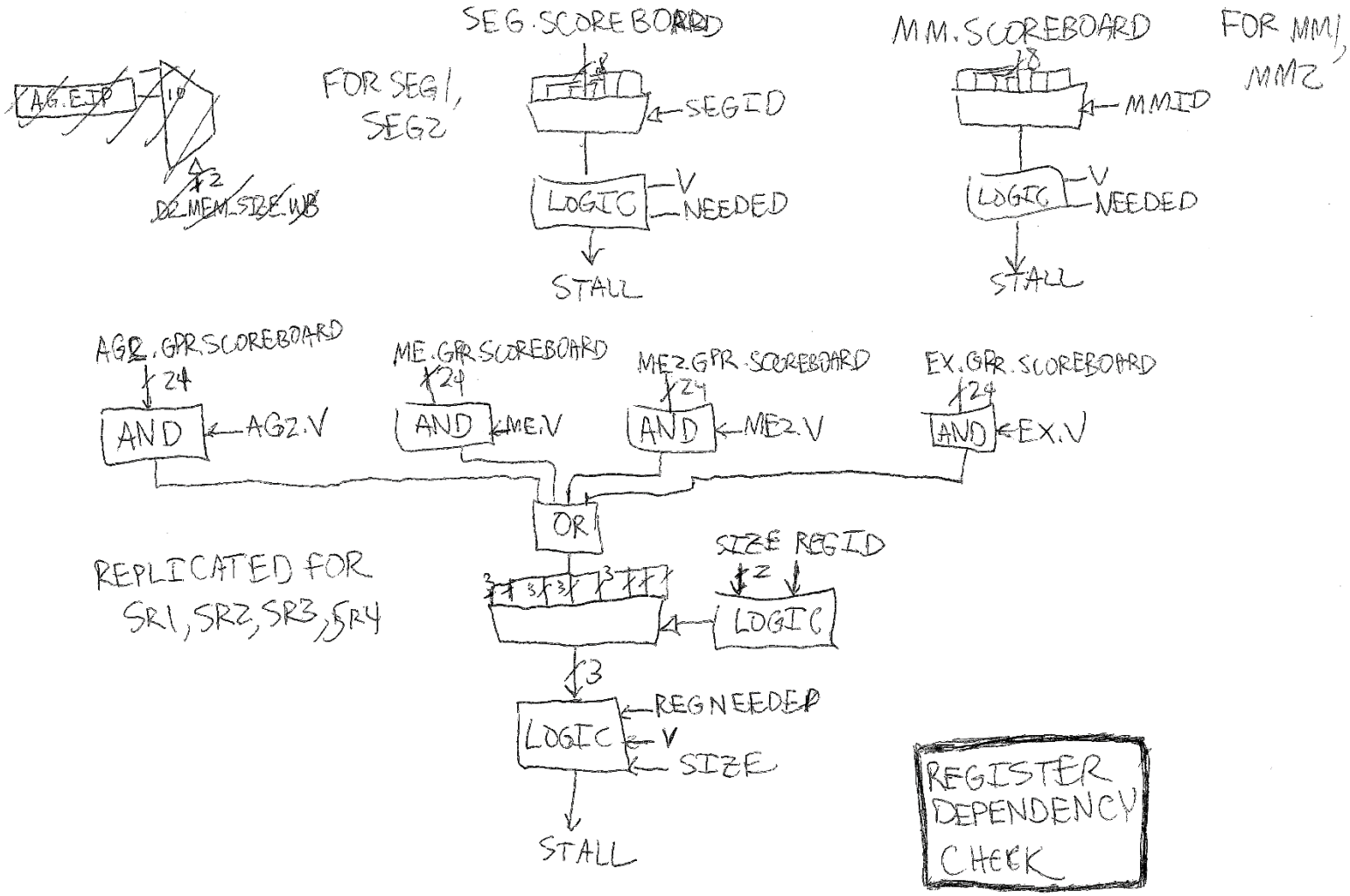
Address Generation Stage 1



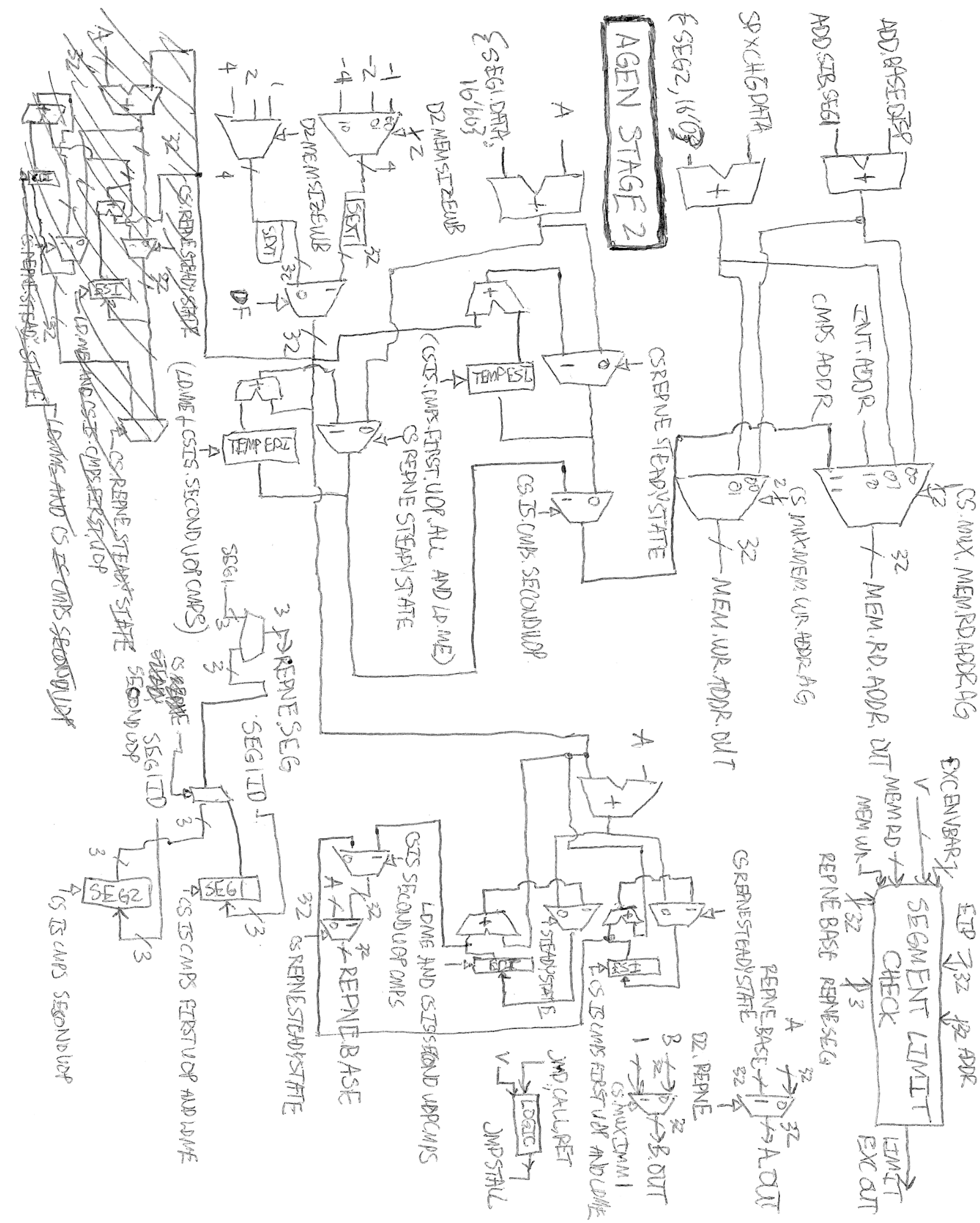
Address Generation Stage 1: Scoreboard Generator



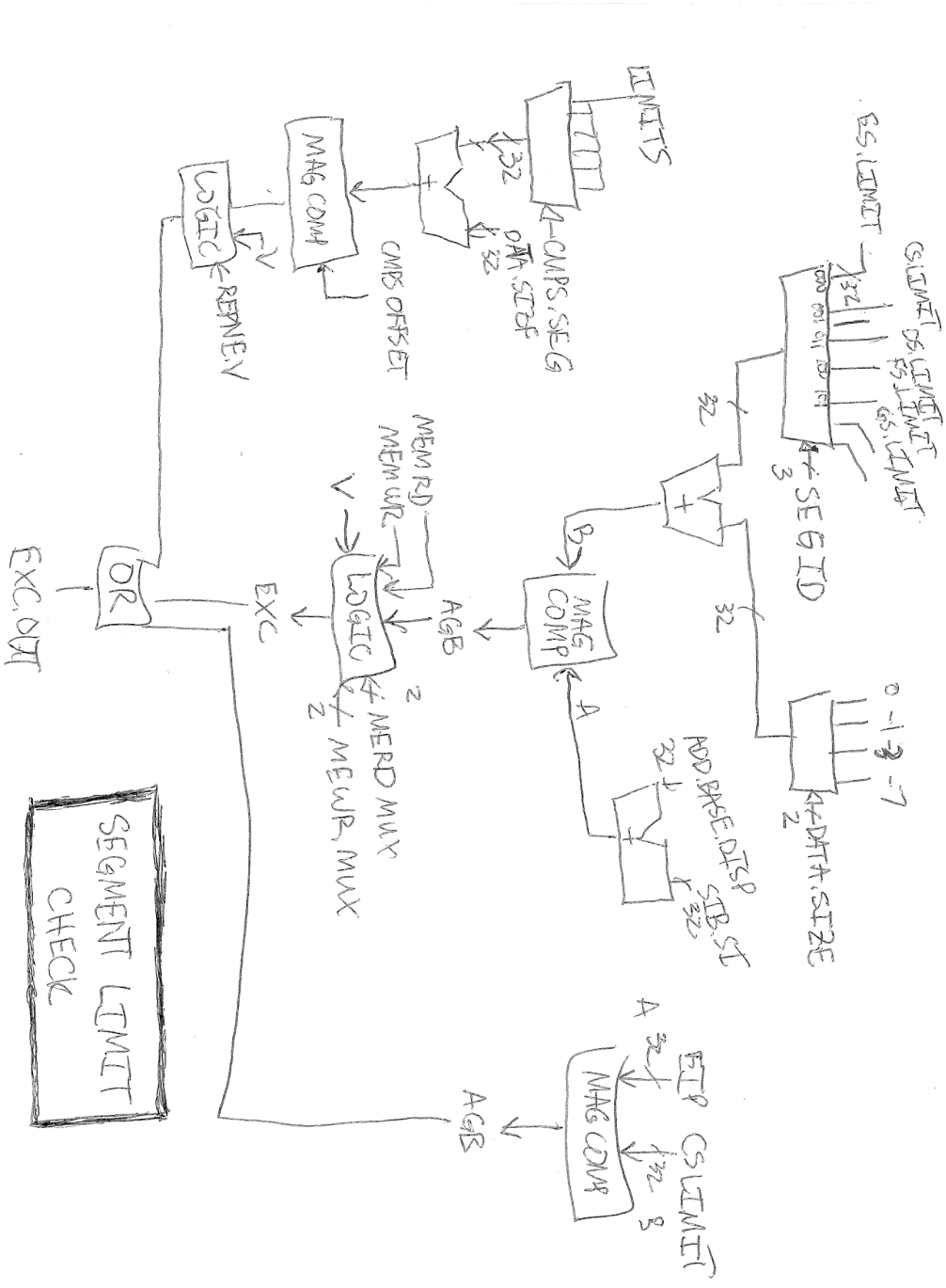
Address Generation Stage 1: Register Dependency Check


SEG.SCOREBOARD
FOR SEG1, SEG2
SEGID
LOGIC
V
NEEDED
STALL
MM.SCOREBOARD
FOR MM1, MM2
MMID
LOGIC
V
NEEDED
STALL
AG2.GPR.SCOREBOARD
ME.GPR.SCOREBOARD
ME2.GPR.SCOREBOARD
EX.GPR.SCOREBOARD
24
AND
AG2.V
ME.V
ME2.V
EX.V
OR
REPLICATED FOR SR1, SR2, SR3, SR4
SIZE REGID
2
LOGIC
3
REGNEEDED
LOGIC
V
SIZE
STALL
REGISTER DEPENDENCY CHECK

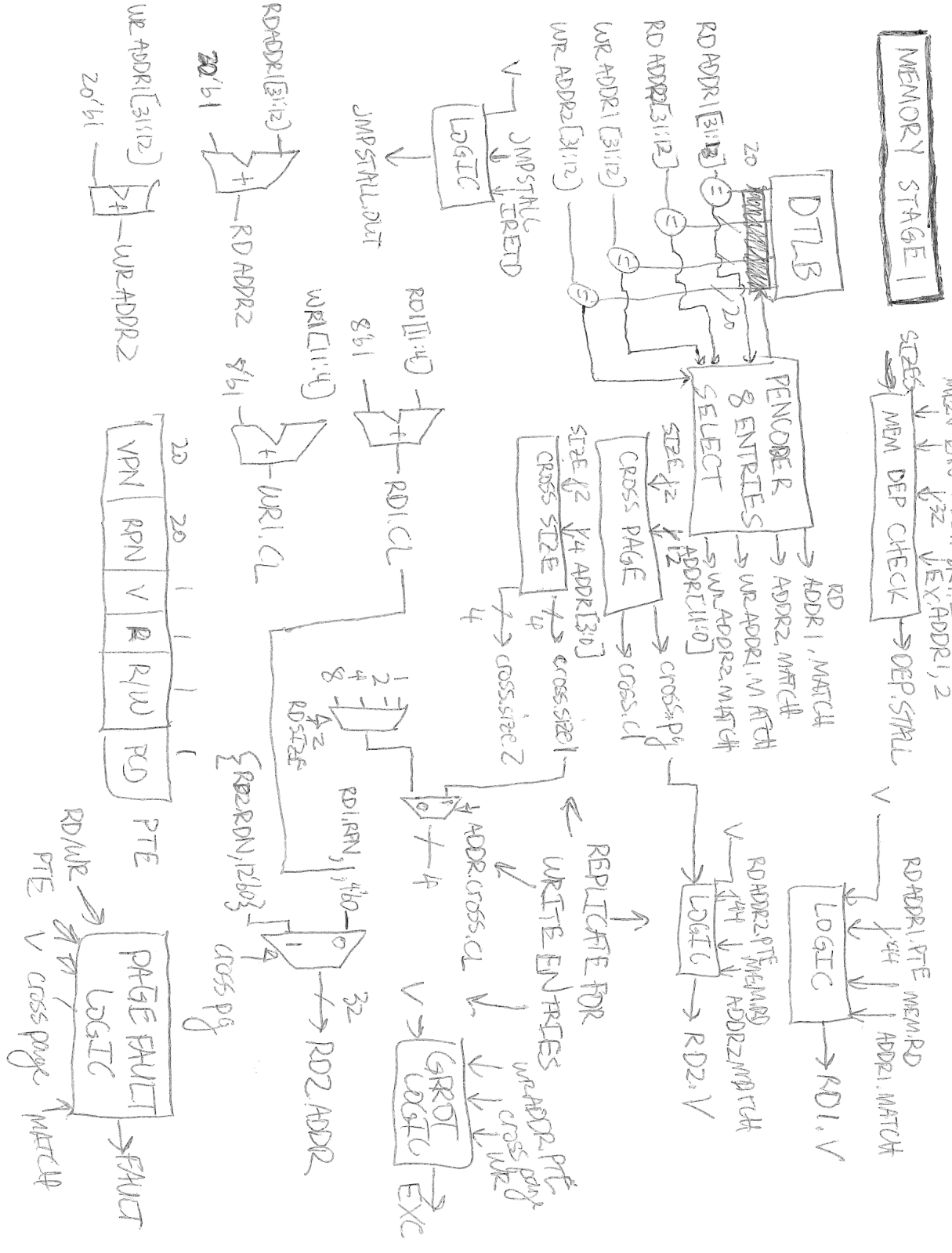
Address Generation Stage 2



Address Generation Stage 2: Segment Limit Check

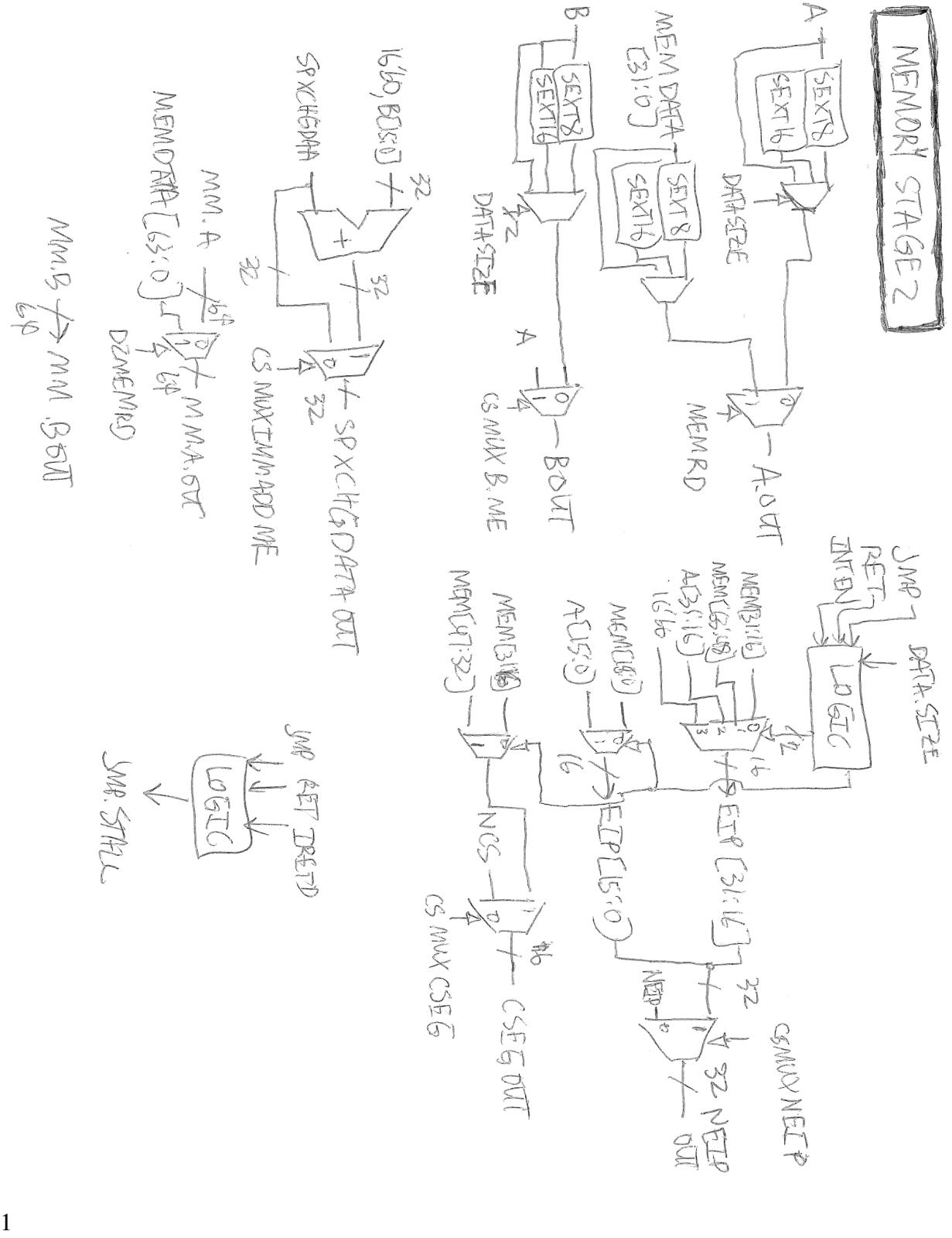


Memory Stage 1



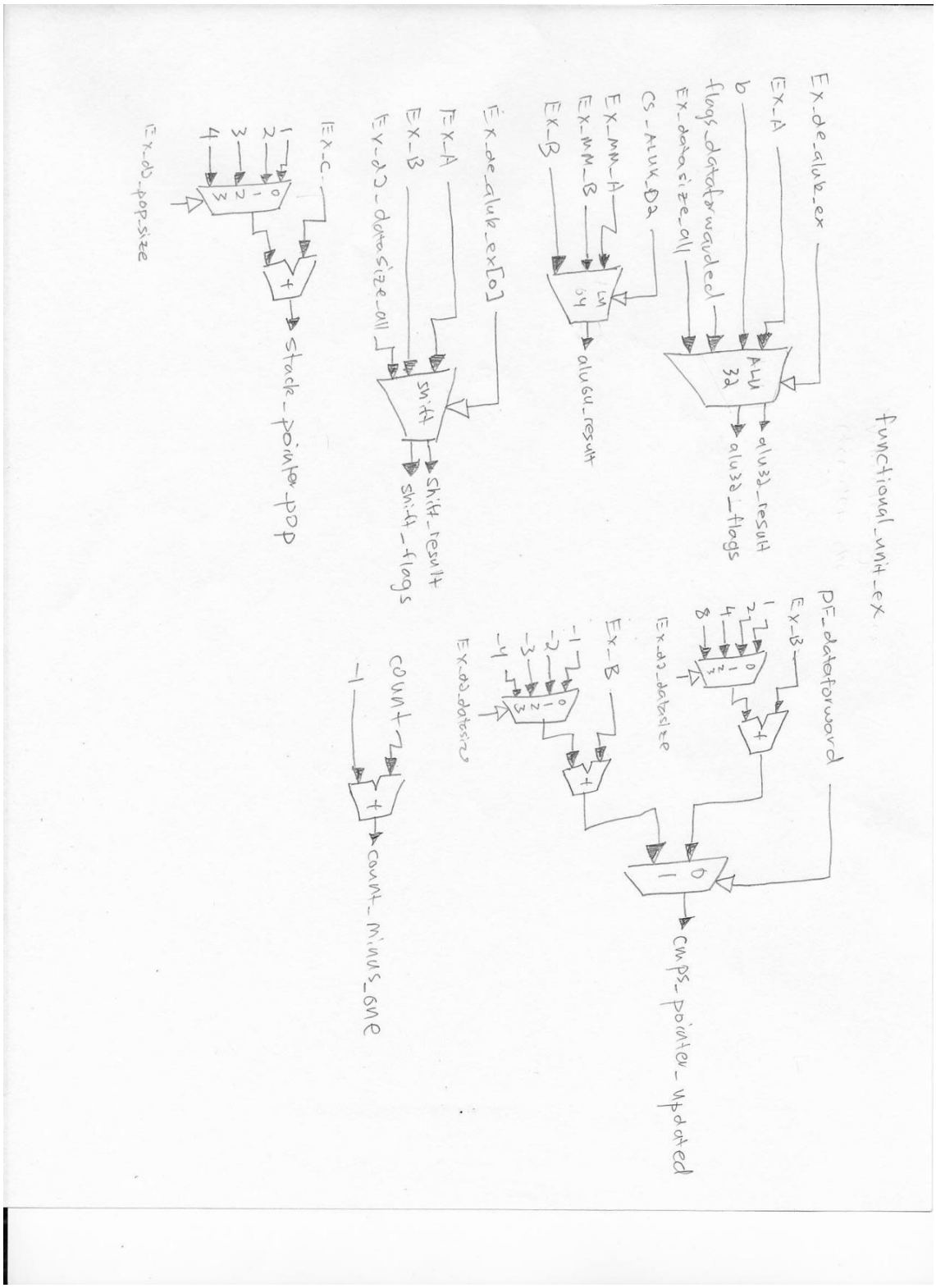
Memory Stage 1: Memory Dependency Check

Memory Stage 2

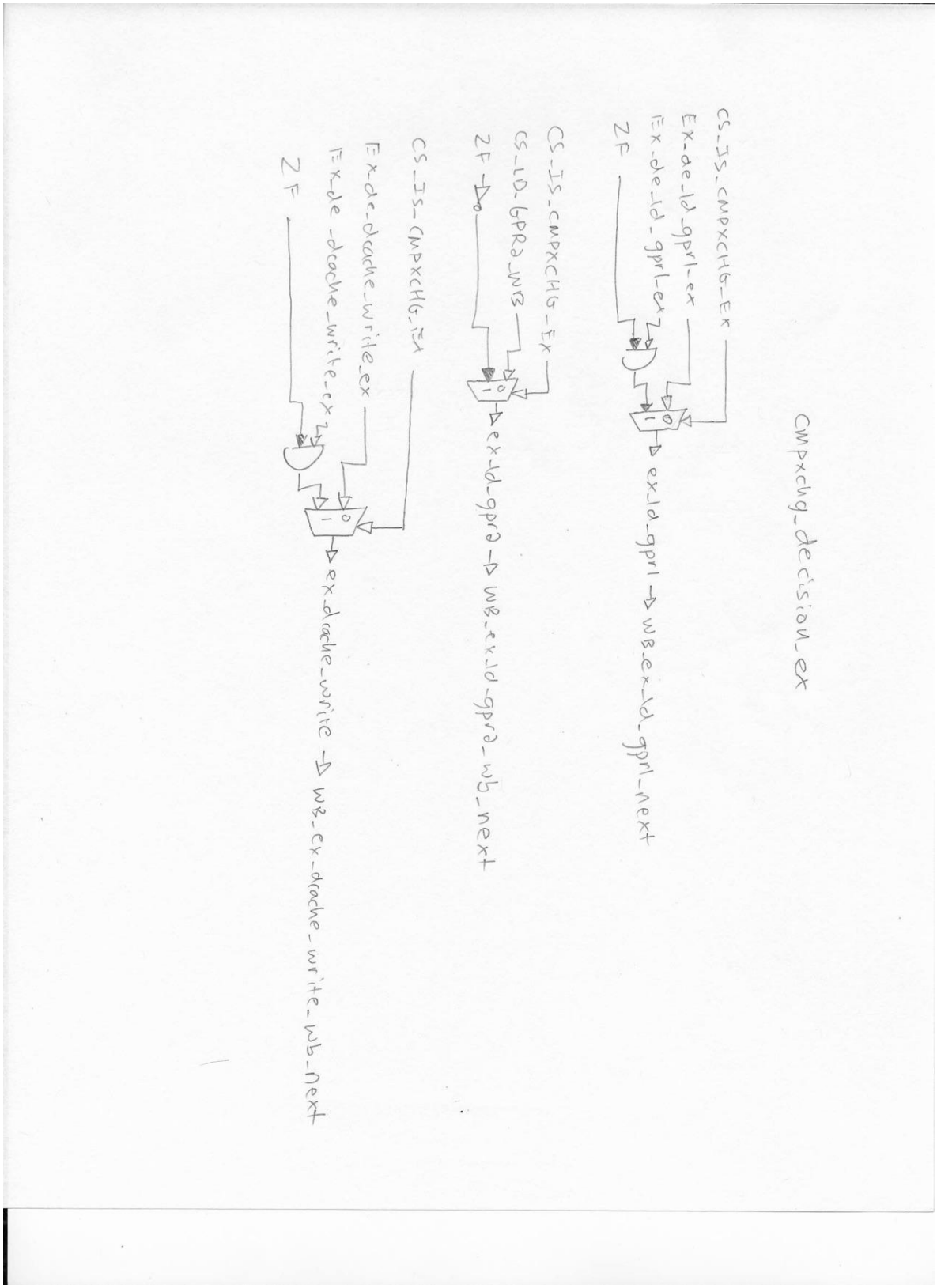


Execute: operand_select_ex

Execute: functional_units_ex

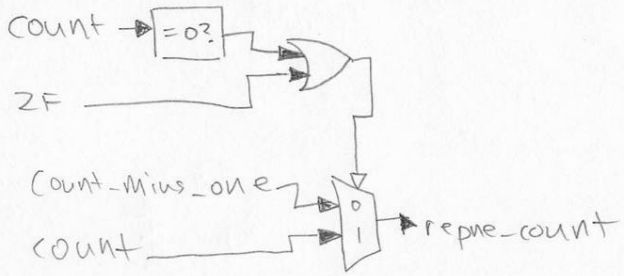


Execute: cmpxchg_decision_ex

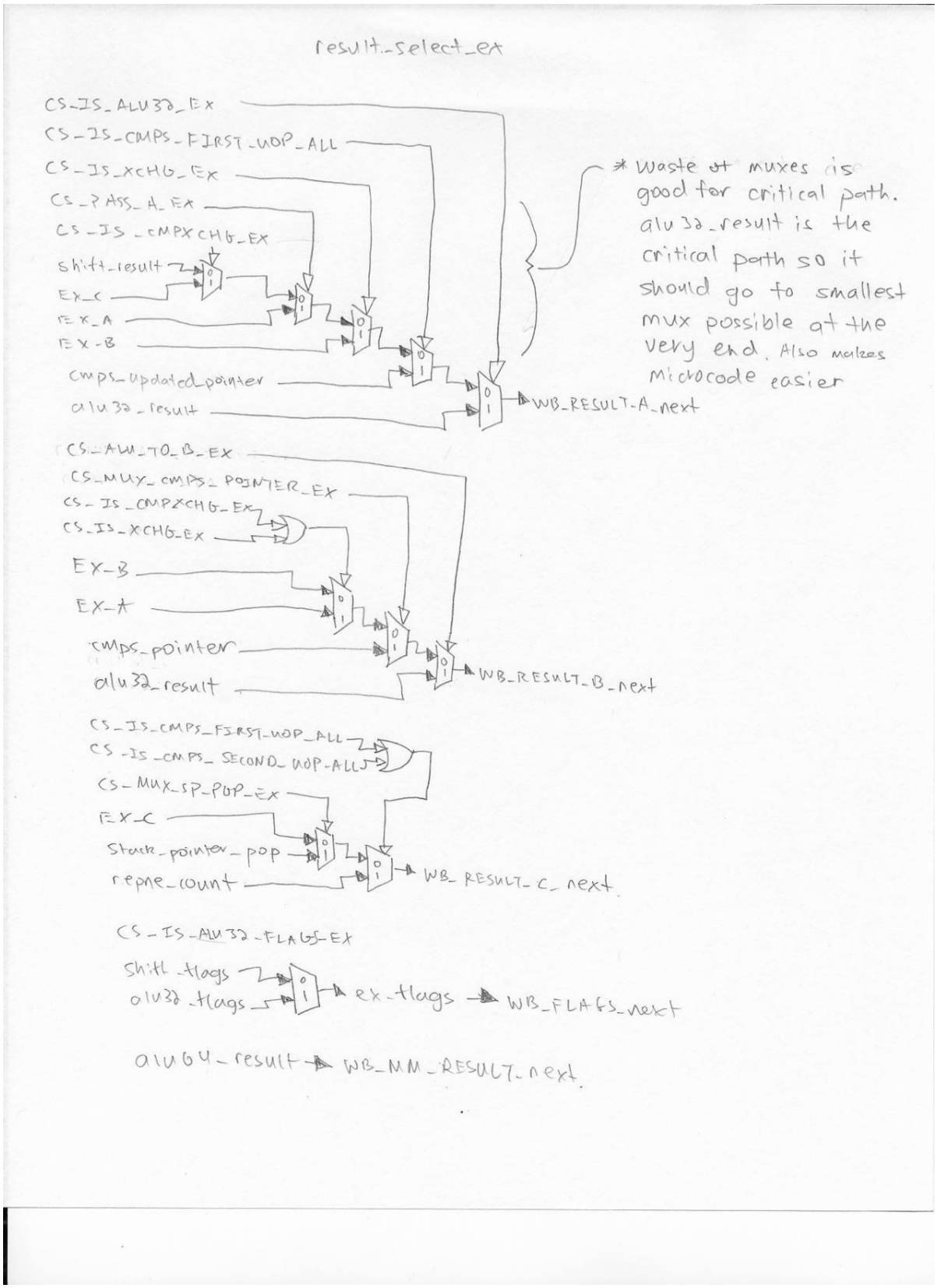


Execute: repne_support_ex

repnf_support_ex



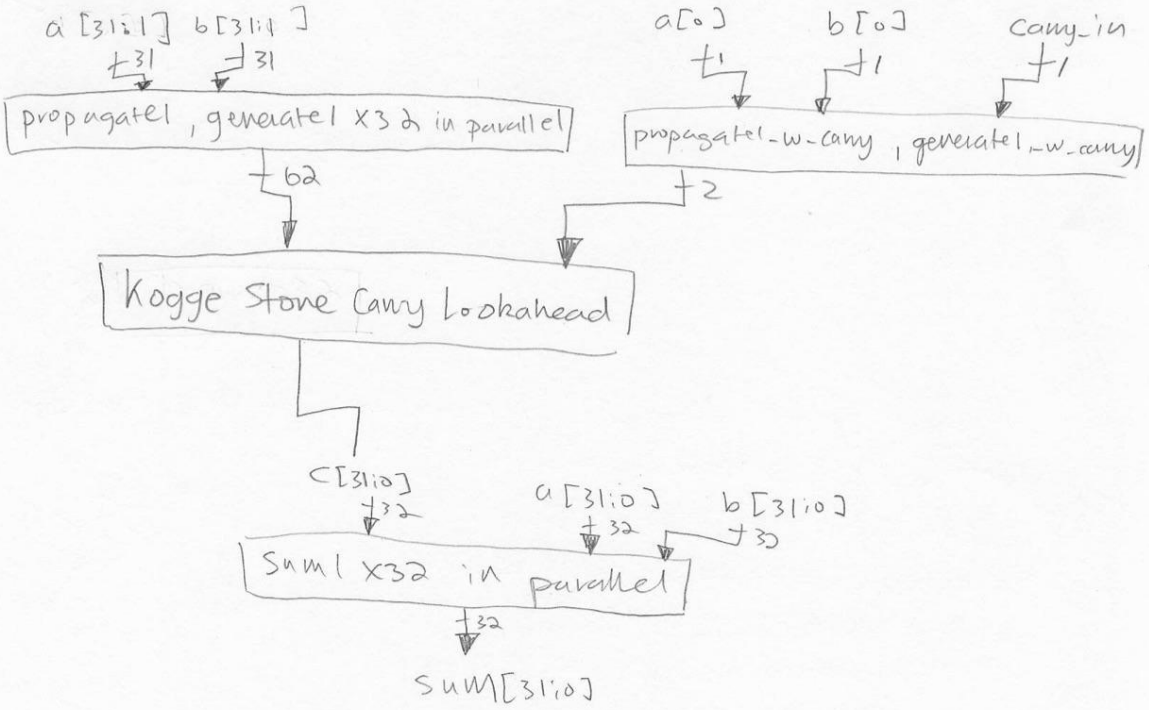
Execute: result_select_ex


result_select_ex
CS_IS_ALU32_EX
CS_IS_CMPS_FIRST_UOP_ALL
CS_IS_XCHG_EX
CS_PASS_A_EX
CS_IS_CMPXCHG_EX
shift_result
EX_C
EX_A
EX_B
cmps_updated_pointer
alu32_result
WB_RESULT_A_next
* Waste of muxes is good for critical path. alu32_result is the critical path so it should go to smallest mux possible at the very end. Also makes microcode easier
CS_ALU_TO_B_EX
CS_MUX_CMPS_POINTER_EX
CS_IS_CMPXCHG_EX
CS_IS_XCHG_EX
EX_B
EX_A
cmps_pointer
alu32_result
WB_RESULT_B_next
CS_IS_CMPS_FIRST_UOP_ALL
CS_IS_CMPS_SECOND_UOP_ALL
CS_MUX_SP_POP_EX
EX_C
stack_pointer_pop
repne_count
WB_RESULT_C_next
CS_IS_ALU32_FLAGS_EX
shift_flags
alu32_flags
ex_flags
WB_FLAGS_next
alu64_result
WB_MM_RESULT_next

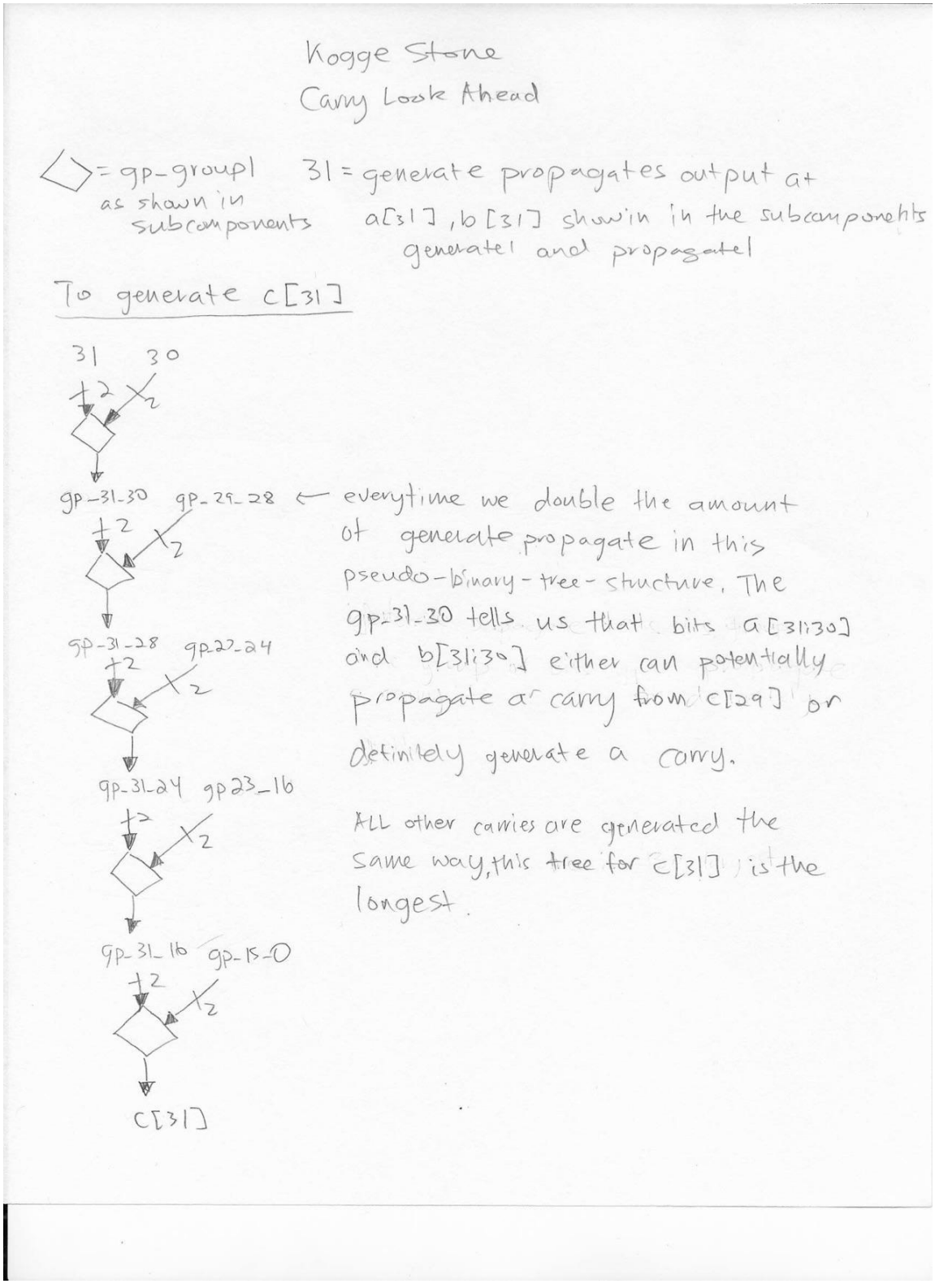
Kogge Stone Adder

Kogge Stone Adder

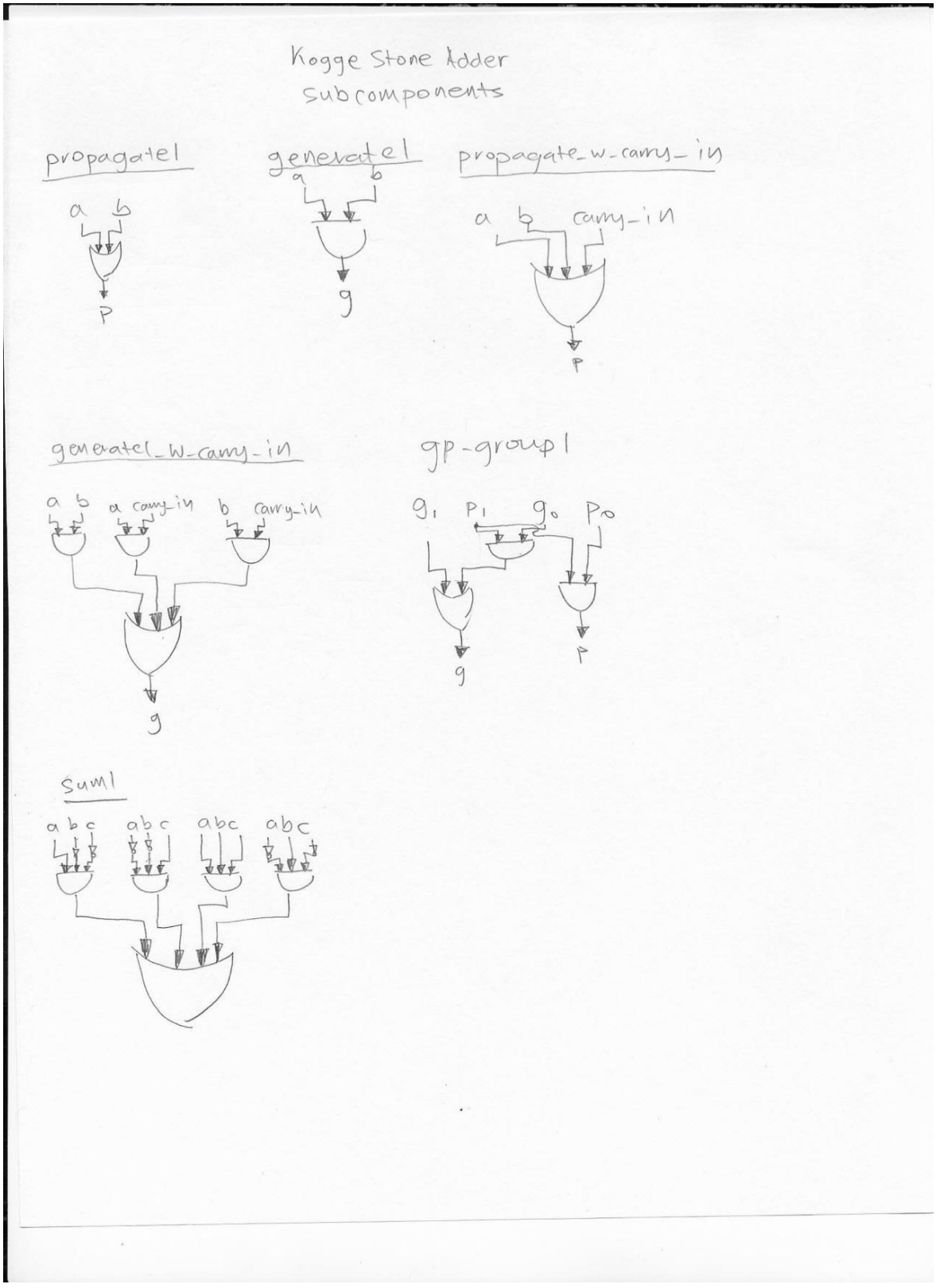
components are shown in the diagrams which follow this page



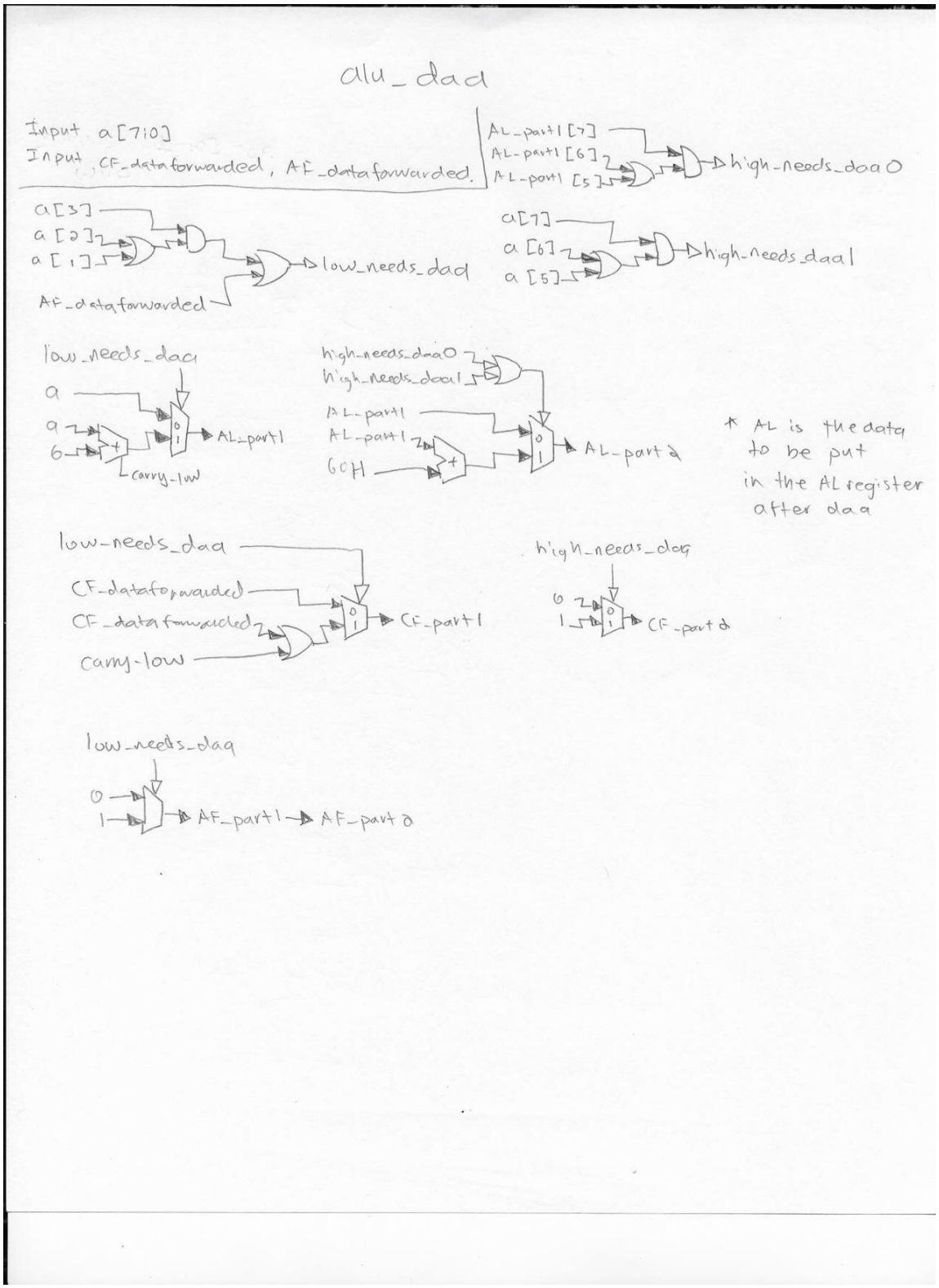
Kogge Stone Carry Look Ahead



Kogge Stone Adder subcomponents



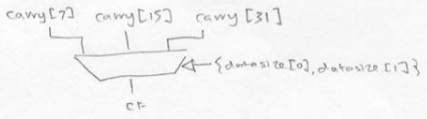
Execute: alu_daa



Execute: Flags

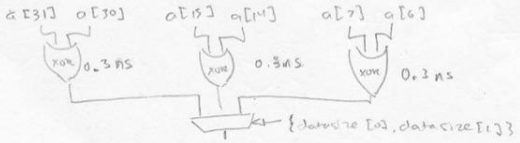
Flags

CF_logic



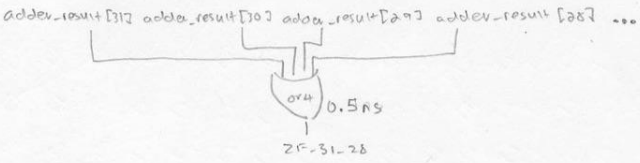
Critical path: 0.22 ns data
0.5 ns select

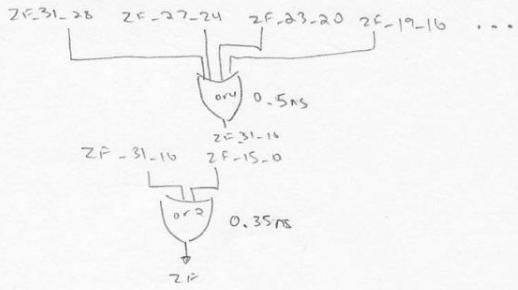
OF_logic



Critical path: 0.52 ns data
0.5 ns select

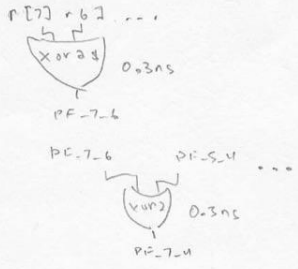
ZF_logic





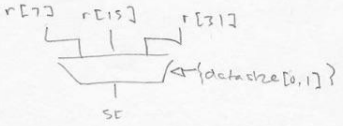
Critical path: 1.35 ns

PF_logic



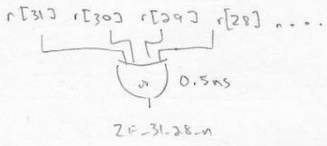
Critical path 0.9 ns

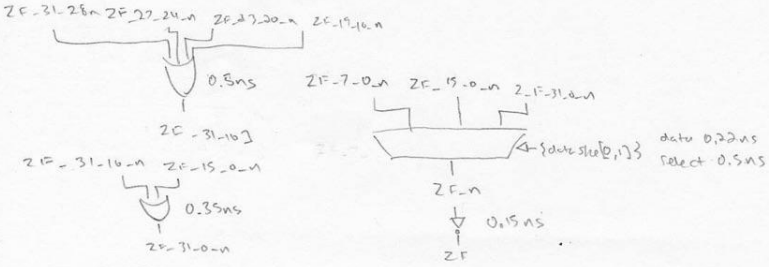
SF_logic



Critical path 0.22 ns data
0.5 ns select

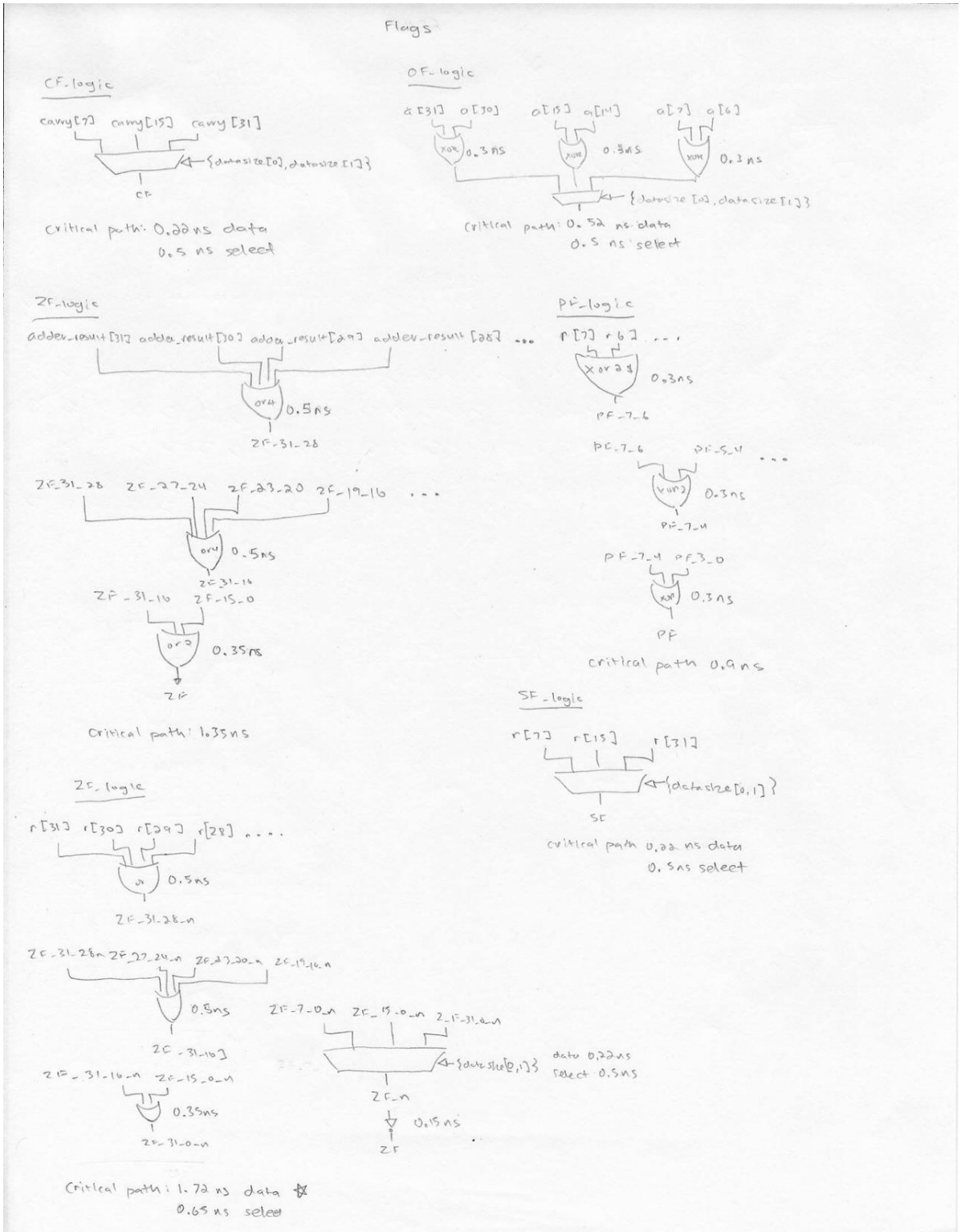
ZF_logic





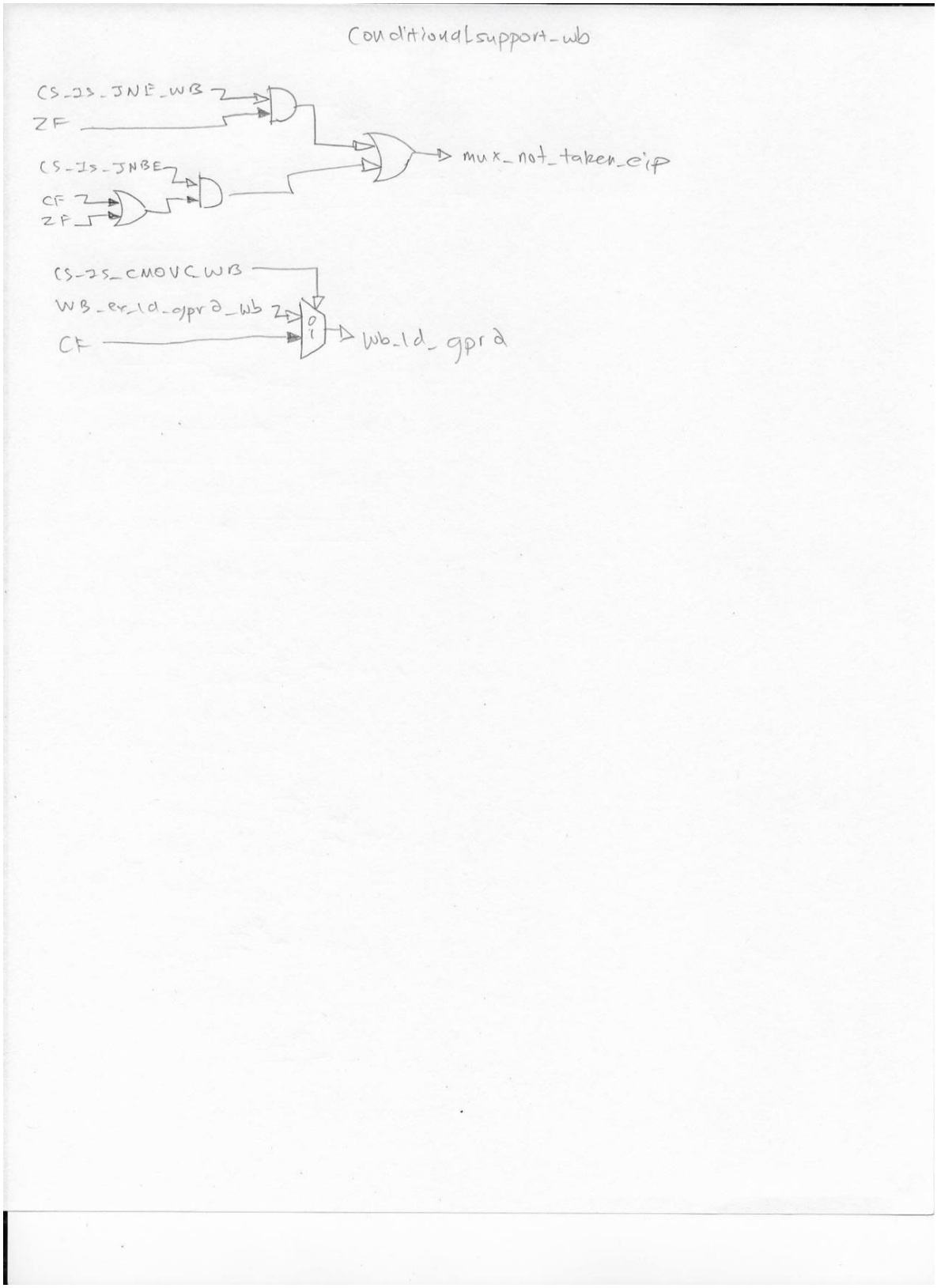
Critical path: 1.72 ns data ☆
0.65 ns select

Execute: PADDSW

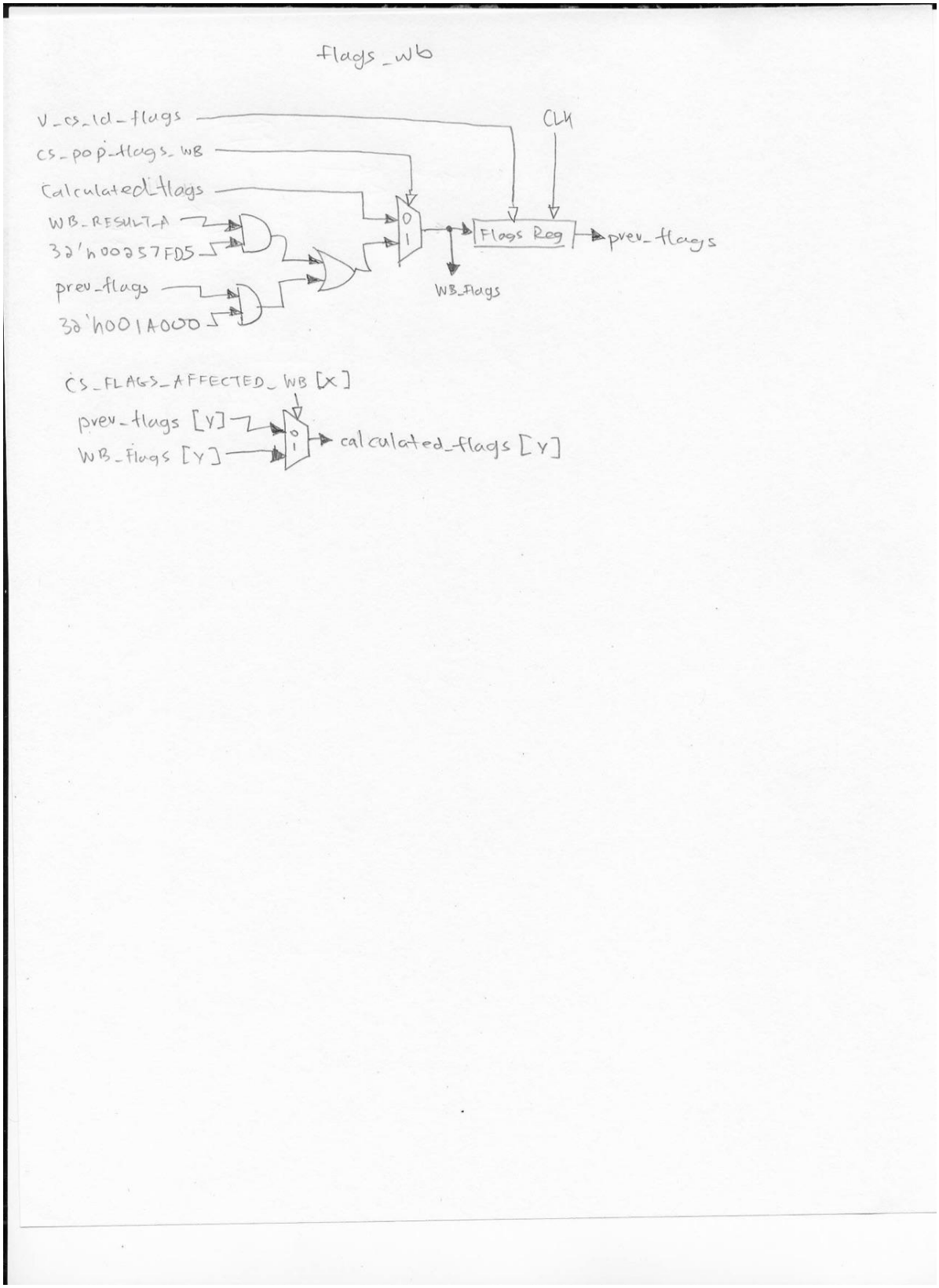


Writeback: operand_select_wb

Writeback: conditional_support_wb

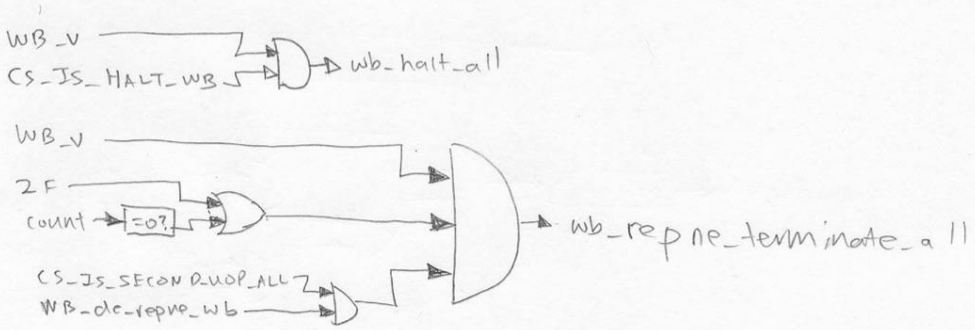


Writeback: flags_wb

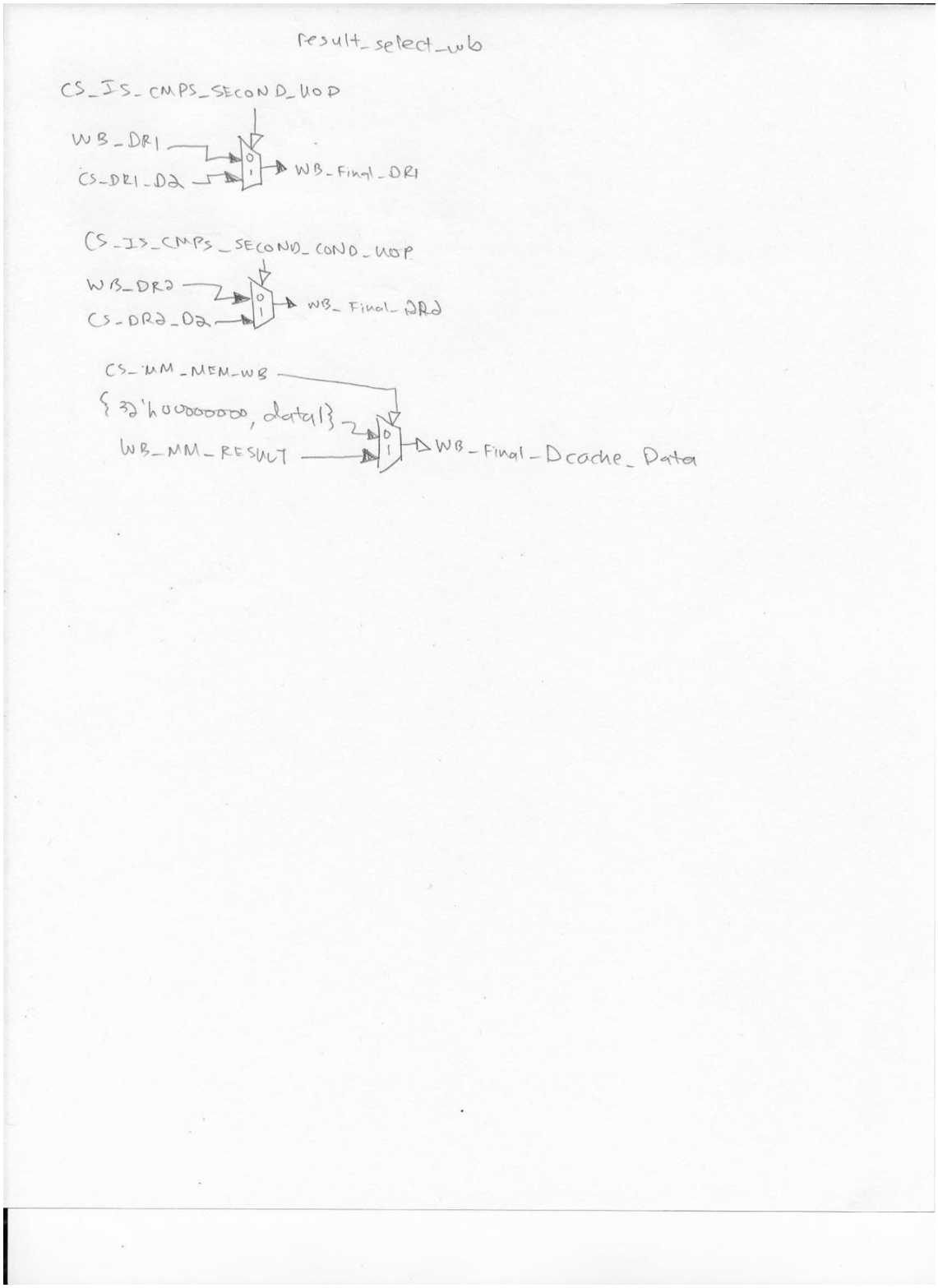


Writeback: halt_repne_support_wb

halt_repne_support_wb

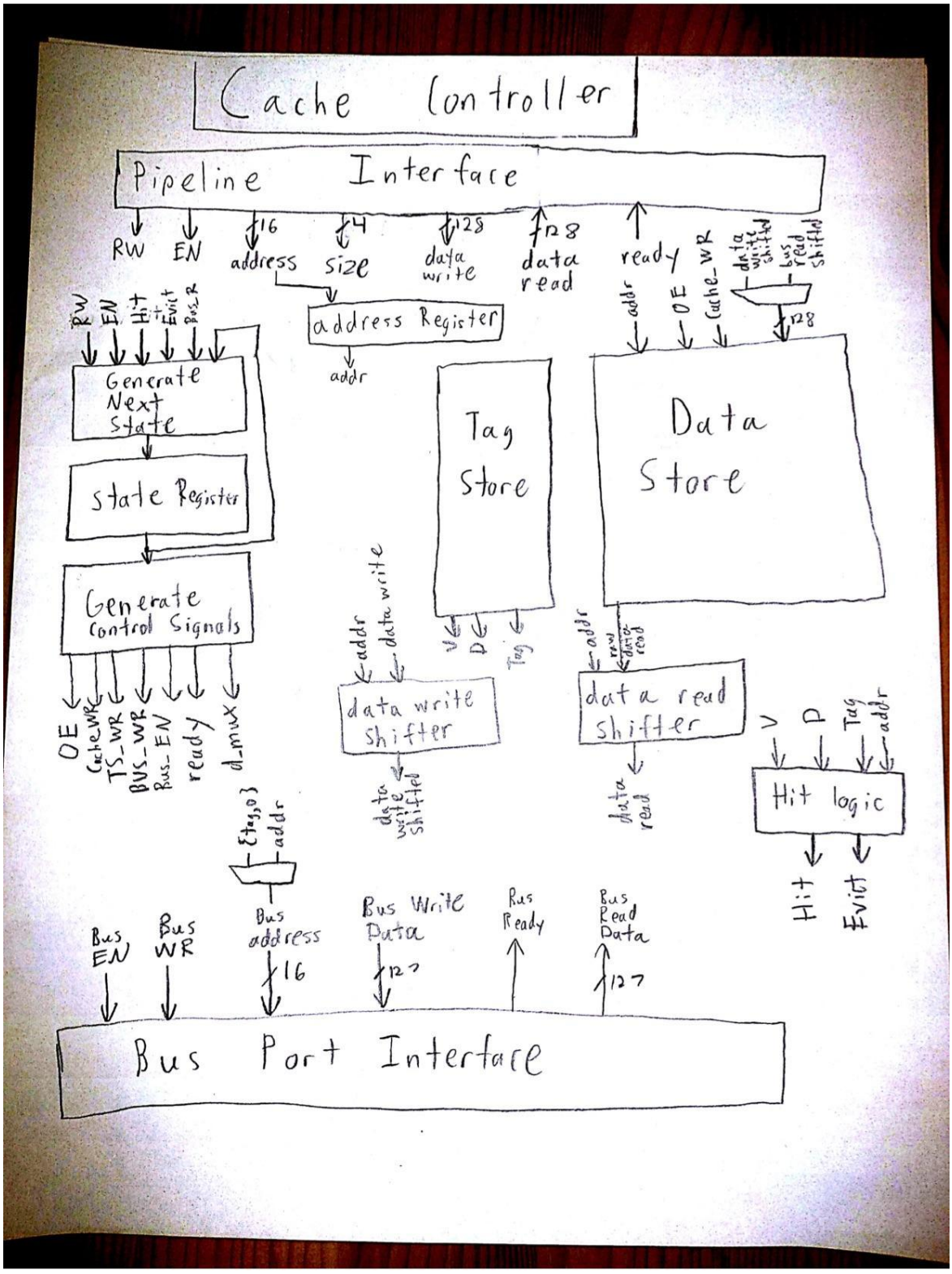


Writeback: result_select_wb

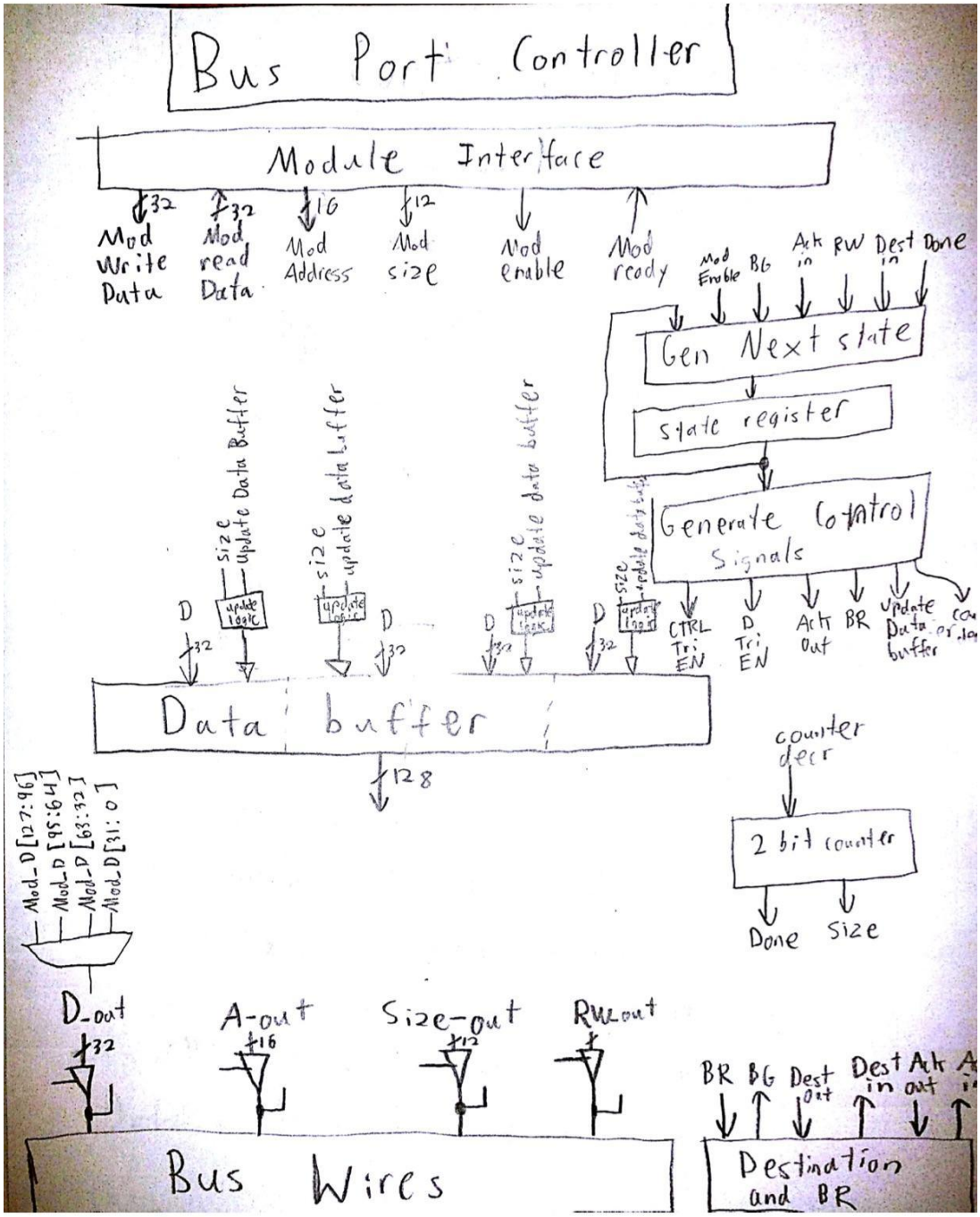


Appendix D: Memory Hierarchy and Bus Schematics

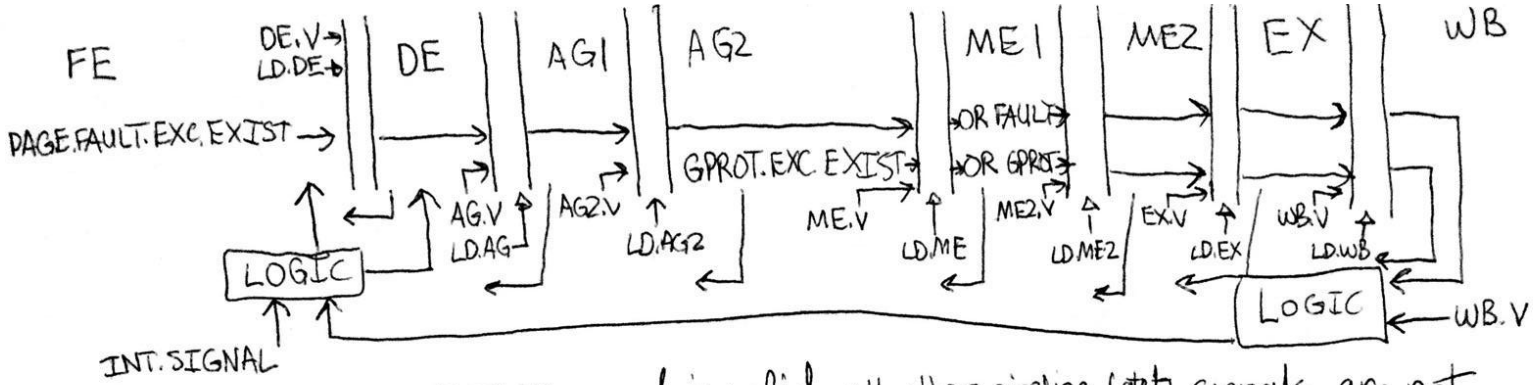
Cache Controller



Bus Port Controller



Appendix E: Interrupt and Exceptions



IF EXC.EXISTS, EXC.EXISTS signal is valid, all other pipeline latch signals are not

INT/EXC LOAD/V SIGNALS

LD.AG2 = LD.ME

AG2.V = AG.PS.V AND WB.FLUSH.PIPELINE.BAR AND $\overline{\text{AG.DEP.STALL}}$

LD.ME = !(RD.STALL OR WR.STALL OR MEM DEP STALL) OR FLUSH.LD

ME.V = AG2.V.PS AND WB.FLUSH.PIPELINE.BAR

LD.ME2 = !(RD.STALL OR WR.STALL) OR FLUSH.LD

ME2.V = WB.FLUSH.PIPELINE.BAR AND $\overline{\text{MEM.DEP.STALL}}$ AND ME.PS.V

LD.EX = $\overline{\text{WR.STALL}}$ OR FLUSH.LD

EX.V = ME2.V.PS AND WB.FLUSH.PIPELINE.BAR AND $\overline{\text{RD.STALL}}$

LD.WB = $\overline{\text{WR.STALL}}$ OR [wb.repre.terminate OR wb.mispredict OR wb.halt OR INT.SIGNAL] ← FLUSH.LD

WB.V = ![(WB.PS.V AND (WB.PAGE.FAULT.EXC.EXIST OR WB.GPROT.EXC.EXIST)) OR (wb.repre.terminate.all) OR INT.SIGNAL OR wb.mispredict OR wb.halt)] AND EXV

WB.FLUSH.PIPELINE.BAR = Brackets

LD.DE = (LDAG AND $\overline{\text{DE.UDP.STALL}}$) OR INT.EXC.EXIST OR FLUSH.LD

DE.V = [!(EXC.EN.V.{D2, AG, AG2, ME, ME2, EX, WB} OR JMP.STALL.{D2, AG, AG2, ME, ME2, EX, WB})
AND !(FETCH.STALL OR wbmispredict)] OR INT.EXC.EXIST OR wb.repne.terminate

LDAG = !(AG.DEP.STALL OR MEM.DEP.STALL OR RD.STALL OR WR.STALL) OR FLUSH.LD

AG.V = DE.PS.V AND WB.FLUSH.PIPELINE.BAR

INT/EXC
LOAD/V SIGNALS

Appendix G: Complete Description of Control Logic

CS_UOP_STALL_DE

Indicates whether this micro-op of the instruction has more micro-ops following it. All regular instructions would have this signal set to 0. For an N micro-op instructions, the first N-1 micro-ops would have this signal set to 1, then the last micro-op would have this signal set to 0.

0	last/only micro-op
1	More micro-ops to come

CS_MUX_OP_SIZE_D2

Selects whether operand override is possible for this instruction.

0	possible
1	data size from control store

CS_OP_SIZE_D2

Data size if determinable from opcode. Chosen if CS_MUX_OP_SIZE_D2 is 1’b1.

00	8 bits
01	16 bits
10	32 bits
11	64 bits

CS_MUX_SR1_SIZE_D2

Force SR1 size to 32-bits for memory operations

0	use output from MUX_OP_SIZE

1	32-bits (2’b10)

CS_MUX_SR2_SIZE_D2

Select output from operand size or take from control store

0	use output from MUX_OP_SIZE
1	CS_SR2_SIZE_D2 (from control store)

CS_SR2_SIZE_D2

Data size if determinable from opcode. Chosen if CS_MUX_SR2_SIZE_D2 is 1’b1.

00	8 bits
01	16 bits
10	32 bits
11	64 bits

CS_MUX_DR1_SIZE_D2

Force DR1 size to 32-bits for CMPS address update

0	use output from MUX_OP_SIZE
1	32-bits (2’b10)

CS_MUX_DR2_SIZE_D2

Force DR2 size to 32-bits for CMPS address update

0	use output from MUX_OP_SIZE
1	32-bits (2’b10)

CS_IS_FAR_CALL_D2

Set if instruction is a far call or far return for memory read and write size. CALL pushes both EIP and CS in the same operation (64-bits). Or 32-bits if operand override.

0	take from MUX_OP_SIZE
1	take from 64-bit or 32-bit size

CS_MUX_MEM_RD_DE

Set if instruction is known to read or not read from memory

0	use mod r/m
1	take value from control store

CS_MEM_RD_DE

0	don’t read from memory
1	read from memory

CS_JMP_STALL_DE

Set for unconditional JMP and CALL instructions

0	not a JMP or CALL
1	unconditional JMP or CALL (stall fetch)

CS_MUX_SR1_D2

Select SR1 ID from R/M bits or hard coded value in control store

0	use R/M bits from opcode

1	take value from control store

CS_SR1_D2

3-bit encoding for GPRs or SEGRs or MMRs

CS_MUX_SR2_D2

Select SR2 ID from R/M bits or hard coded value in control store

0	use R/M bits from opcode
1	take value from control store

CS_SR2_D2

3-bit encoding for GPRs or SEGRs or MMRs

CS_MUX_SEG1_D2

Select segment ID from decode generated value or ES (3’b000)

0	use SEG ID generated from decode
1	use encoding for ES (for CMPS)

CS_MUX_SEG2_D2

select segment ID from SS (3’b010) or REG field of opcode (for MOV SEG)

0	use SS (for stack accesses)
1	use REG field of opcode

CS_SR1_NEEDED_AG

Set if SR1 is needed for instruction; otherwise, 0. Can be overridden when mod r/m indicates a memory operation

0	SR1 ID is not needed
1	SR1 ID is needed

CS_SR2_NEEDED_AG

Set if SR2 is needed for instruction; otherwise, 0

0	SR2 ID is not needed
1	SR2 ID is needed

CS_MUX_SEG1_NEEDED_AG

Select if SEG1 is needed for instruction. Take the value from control store or choose if mod r/m equals a memory operation

0	Choose mod r/m from decode
1	Take from control store

CS_SEG1_NEEDED_AG

Set if SEG1 is needed for instruction; otherwise, 0

0	SEG1 ID is not needed
1	SEG1 ID is needed

CS_SEG2_NEEDED_AG

Set if SEG2 is needed for instruction; otherwise, 0

0	SEG2 ID is not needed

1	SEG2 ID is needed

CS_MM1_NEEDED_AG

Set for MM instructions if MM1 is needed; otherwise, 0

0	MM1 ID is not needed
1	MM1 ID is needed

CS_MM2_NEEDED_AG

Set for MM instructions if MM2 is needed; otherwise, 0

0	MM2 ID is not needed
1	MM2 ID is needed

CS_MUX_A_AG

Select sources for operand A. Can take from SR1 ID read from register file, SEG1 ID from register file, SEG2 ID data from register file, or CS data passed from decode

00	SR1_DATA
01	SEG1_DATA
10	SEG2_DATA
11	CS_DATA (for push CS)

CS_MUX_B_AG

select if the B operand should come from SR2 ID data from register file or immediate

0	SR2_DATA

1	IMM32 from decode

CS_MUX_DRID1_AG

select if the destination operand should come from SR1 ID or SR2 ID

0	SR1 ID (3-bits)
1	SR2 ID (3-bits)

CS_MUX_EIP_JMP_REL_AG

select if relative JMP immediate should be added to EIP

0	32’b0
1	offset from JMP or CALL relative instr.

CS_MUX_NEXT_EIP_AG

select if next EIP comes directly from instruction bytes

0	output of MUX_EIP_JMP_REL
1	offset[31:0[from decode

CS_MUX_NEXT_CSEG_AG

select if next CS register value comes from instruction or latches

0	CS from pipeline latches
1	offset bytes from instruction

CS_MUX_SP_ADD_SIZE_AG

select what size to add to stack pointer

0	add negative size for push
1	add previous pop size for multi-pop ops

CS_MUX_TEMP_SP_AG

select the saved stack pointer value or the value passed from latches

0	SR2_DATA
1	value saved in internal SP register

CS_MUX_SP_PUSH_AG

set if stack pointer must be pre decremented for push or added on next pop

0	32’b0
1	current push or previous pop size

CS_MUX_MEM_RD_ADDR_AG

select final linear address for instructions needing to read from memory

00	address from r/m or SIB
01	stack address
10	interrupt table vector address
11	CMPS address

CS_MUX_MEM_WR_ADDR_AG

select final linear address for instructions needing to write to memory

00	address from r/m or SIB

01	stack address
10	unused
11	unused

CS_MUX_IMM1_AG

force an immediate of one for increment and shift by one operations

0	value from B latch
1	32’b1

CS_MUX_B_ME

swap the values in A and B for CMPS operations (when both register and memory are needed). Memory always comes through the A latch

0	B latch pass through
1	A latch value

CS_MUX_IMM_ADD_ME

select to add the immediate in B for RET imm16 instructions

0	32’b0
1	B (contains imm16)

CS_IS_NEAR_RET_M2

set for near return instructions to determine location of EIP in stack memory read

0	not a near return

1	near return instruction

CS_IS_FAR_RET_M2

set for far returns and IRETDs to determine locations of EIP and CS in stack memory reads

0	not far return or IRETD
1	instruction is far return or IRETD

CS_MUX_NEXT_EIP_M2

select to load EIP from memory read or A operand (for JMP r/m, CALL r/m), or pass through EIP value from latch

0	EIP latch
1	selection of memory or A muxes

CS_MUX_NEXT_CSEG_M2

select to load CS segment register from memory read, or pass through CS value from latch

0	CS latch (code segment register)
1	memory read operand

CS_IS_CMPS_FIRST_UOP_ALL

Selects whether the micro-op is the first uop of an cmps instruction.

0	not first uop of cmps
1	first uop of cmps

CS_IS_CMPS_SECOND_UOP_ALL

Selects whether the micro-op is the second uop of an cmps instruction.

0	not second uop of cmps
1	second uop of cmps

CS_REPNE_STEADY_STATE

Selects whether the micro-op is in the second or later iteration of repne cmps.

0	first iteration of repne cmps
1	second or later iteration of repne cmps

CS_REPNE_INTERNAL_COUNTN_WB

Selects whether the micro-op sources count from ECX or the internal count register.

0	source count from ECX
1	source count from internal register

CS_REPNE_INTERNAL_COUNT_WB

Selects whether the micro-op sources count from ECX or the internal count register.

0	source count from ECX
1	source count from internal register

CS_IS_CMPXCHG_EX

Selects whether the micro-op is the micro-op for the cmpxchg instruction or not.

0	not cmpxchg instruction
1	cmpxchg instruction

CS_LD_GPR1_D2

Selects whether the micro-op will load DR1. This may be overwritten in decode.

0	does not load GPR1
1	loads GPR1

CS_DR1_D2

Selects destination register 1. This may be overwritten in decode.

000	AL/AX/EAX
001	CL/CX/ECX
010	DL/DX/EDX
011	BL/BX/EBX
100	AH/SP/ESP
101	CH/BP/EBP

110	DH/SI/ESI
111	BH/DI/EDI

CS_MUX_DR1_D2

Selects whether the micro-op chooses DR1 from the control store or if it’s a hard coded value.

0	DR1 is hard coded
1	DR1 is from control store

CS_LD_GPR2_EX

Selects whether the opcode writes to DR2 or not.

0	does not writes DR2
1	writes to DR2

CS_ALU_TO_B_EX

Puts the results of ALU in the B latch between EX and WB.

0	don’t choose WB_B as alu result
1	choose WB_B as alu result

CS_DR1_D2

Selects destination register 2. This may be overwritten in decode.

000	AL/AX/EAX
001	CL/CX/ECX
010	DL/DX/EDX
011	BL/BX/EBX
100	AH/SP/ESP
101	CH/BP/EBP
110	DH/SI/ESI
111	BH/DI/EDI

CS_MUX_DR2_D2

Selects whether the micro-op choose DR2 from the control store or if it’s a hard coded value.

0	DR2 is hard coded
1	DR2 is from control store

CS_DCACHE_WRITE_D2

Selects whether the micro-op writes to data cache or not

0	does not write to data cache
1	write to data cache

CS_MUX_MEM_WR_D2

Selects whether the dcache write signal is determined by the control store or is hard coded.

0	dcache write signal is hard coded
1	dcache write signal is from control store

CS_LD_GPR3_WB

Selects whether the micro-op writes to DR3 or not

0	does not write to DR3
1	writes to DR3

CS_DR3_WB

Selects destination register 2. This may be overwritten in decode.

000	AL/AX/EAX
001	CL/CX/ECX

010	DL/DX/EDX
011	BL/BX/EBX
100	AH/SP/ESP
101	CH/BP/EBP
110	DH/SI/ESI
111	BH/DI/EDI

CS_ALUK_D2

Selects the operation for ALU32 or ALU64

000	ADD/PADDW
001	OR/PADDD
010	NOT/PADDSW
011	DAA/PSHUFW

100	AND/PASS_A
101	CLD/SWAP_A
110	CMP
111	STD

CS_MUX_ALUK_D2

Selects whether the ALUK comes from control store or from decode.

0	ALUK comes from decode
1	ALUK comes from control store

CS_IS_ALU32_EX

Selects whether the A latch between EX and WB is the results of ALU32

0	WB_A is not the results of ALU32
1	WB_A is the results of ALU32

CS_IS_ALU32_FLAGS_EX

Selects whether the Flags latch between EX and WB is the result of ALU32

0	WB_FLAGS does not come from the results of ALU32

1	WB_FLAGS comes from the results of ALU32

CS_IS_XCHG_EX

Selects whether EX_A and EX_B latches passes into WB_A and WB_B latches or are swapped.

0	not swap
1	swap

CS_IS_PASS_A_EX

Selects whether the EX_A latch is passed into the WB_A latch or not

0	not pass
1	pass

CS_MUX_SP_POP_EX

Selects whether the WB_C latch is the result of the increment stack pointer unit or not

0	WB_C is not incremented stack pointer
1	WB_C is incremented stack pointer

CS_MUX_CMPS_POINTER_EX

Selects whether or not the WB_B latch is the result of the updated pointer used by cmps or not

0	WB_B is not updated pointer

1	WB_B is updated pointer

CS_SAVE_NEIP_WB

Selects whether or not the writeback stage saves NEIP into the SAVE_NEIP register or not.

0	do not save NEIP
1	save NEIP

CS_SAVE_NCS_WB

Selects whether or not the writeback stage saves NCS into the SAVE_NCS register or not.

0	do not save NCS
1	save NCS

CS_PUSH_FLAGS_WB

Selects whether the micro-op pushes the flags onto the stack or not

0	does not push flags
1	push flags

CS_POP_FLAGS_WB

Selects whether the micro-op pushes the flags from the stack or not

0	does not pop flags
1	pop flags

CS_USE_TEMP_NEIP_WB

Selects whether the micro-op uses the saved NEIP or not

0	use regular NEIP
1	use saved NEIP

CS_USE_TEMP_NCS_WB

Selects whether the micro-op uses the saved NEIP or not

0	use regular NCS
1	use saved NCS

CS_IS_JNBE_WB

Selects whether the micro-op is a JNBE micro-op

0	not JNBE
1	JNBE

CS_IS_JNE_WB

Selects whether the micro-op is a JNBE micro-op

0	not JNE
1	JNE

CS_LD_EIP_WB

Selects whether the micro-op loads the EIP with NEIP or not

0	does not load NEIP into EIP
1	load NEIP into EIP

CS_LD_CS_WB

Selects whether the micro-op loads the NCS with CS or not

0	does not load NCS into CS
1	load NCS into CS

CS_IS_CMOVC_WB

Selects whether the micro-op is the micro-op for cmovc or not

0	is not CMOVC
1	is CMOVC

CS_LD_SEG_WB

Selects whether the micro-op writes to the segment register or not

0	does not write to segment register
1	writes to segment register

CS_LD_MM_WB

Selects whether the micro-op writes to the segment register or not

0	does not write to MM register

1	writes to MM register

CS_MM_MEM_WB

Selects whether the data we write to the dcache is from the WB_MM register or from the WB_A latch.

0	source dcache write data from WB_A latch
1	source dcache write data from WB_MM latch

CS_LD_FLAGS_WB

Selects whether the micro-op writes the flags or not

0	does not write the flags
1	writes the flags

CS_IS_HALT_WB

Selects whether the micro-op will halt the processor or not

0	does not halt the processor
1	halts the processor

CS_FLAGS_AFFECTED_WB

Selects whether a flag is affected by the micro-op (1 at that position) or is not affected by the micro-op (0 at that position).

bit 0	determines if CF flag be affected

bit 1	determines if PF flag is affected
bit 2	determines if AF flag is affected
bit 3	determines if ZF flag is affected
bit 4	determines if SF flag is affected
bit 5	determines if DF flag is affected
bit 6	determines if OF flag is affected

CS_MUX_MICROSEQUENCER_D2

Selects whether the next micro-address comes from the micro-sequencer or from decode.

0	next micro-address is form decode
1	next micro-address is from micro-sequencer

APPENDIX F: Parts Listing

lib1:

// nand2$ - 2 inputs nand

// nand3$ - 3 inputs nand

// nand4$ - 4 inputs nand

// and2$ - 2 inputs and

// and3$ - 3 inputs and

// and4$ - 4 inputs and

// nor2$ - 2 inputs nor

// nor3$ - 3 inputs nor

// nor4$ - 4 inputs jor

// or2$ - 2 inputs or

// or3$ - 3 inputs or

// or4$ - 4 inputs or

// xor2$ - 2 inputs xor

// xnor2$ - 2 inputs xnor

// inv1$ - 1 input inverter

// dff$ - edge-triggered D-ff with set/reset

lib2:

// dff16b$ - 16-bits D-flip flop with bit addressable

// dff32b$ - 32-bits D-flip flop with bit addressable

// reg64e$ - 64-bits registers with write enable

// reg32e$ - 32-bits register with write enable

// ioreg8$ - 8-bit IO register (D-flip flop)

// ioreg16$ - 16-bits IO register (D-flip flop)

lib3:

// regfile$ 16x8bits double-read port register files

// regfile8x8$ 8x8bits single-read port register files

// ram16b8w$ 16-bits x 8 words RAM

// ram8b8w$ 8-bits x 8 words RAM

// ram8b4w$ 8-bits x 4 words RAM

// rom4b32w$ 4-bits x 32 words ROM

// rom32b32w$ 32-bits x 32 words ROM

// rom64b32w$ 64-bits x 32 words ROM

lib4:

// mux2$ - 1-bit 2-1 multiplexer

// mux3$ - 1-bit 3-1 multiplexer

// mux4$ - 1-bit 4-1 multiplexer

//

// mux2_8$ - 8-bits 2-1 multiplexer

// mux2_16$ - 16-bits 2-1 multiplexer

// mux3_8$ - 8-bits 3-1 multiplexer

// mux3_16$ - 16-bits 3-1 multiplexer

// mux4_8$ - 8-bits 4-1 multiplexer

// mux4_16$ - 16-bits 4-1 multiplexer

lib5:

decoder2_4$ - 2 to 4 decoder

mag_comp8$ - 8-bits magnitude comparator

lib6:

sram128x8$ 128 x 8-bit SRAM