RAVE-I x86 CPU

Members

Rohan Jain - rj23558

Asher Nederveld - jan3476

Varun Arumugam - va6998

Edgar Turcotte - eat2448

The University of Texas at Austin

Department of Electrical and Computer Engineering

Microarchitecture - ECE 382N.19

Supervisor

Dr. Yale Patt

ABSTRACT

For our final project for ECE 382N.19 Microarchitecture at The University of Texas at Austin, the project goal was to design and implement a CPU (central processing unit) that handled a subset of the x86 ISA using only structural Verilog and simulating it with VCS and DVE. Besides simple functionality, the other goal was to produce a high performance processor as well. To meet this goal, a seven stage pipeline was created with multiple performance features including full data forwarding, a G-Share branch predictor, and non-blocking cache. Discussions and the results of the processor will be shown later on within the report.

Table of Contents

Introduction

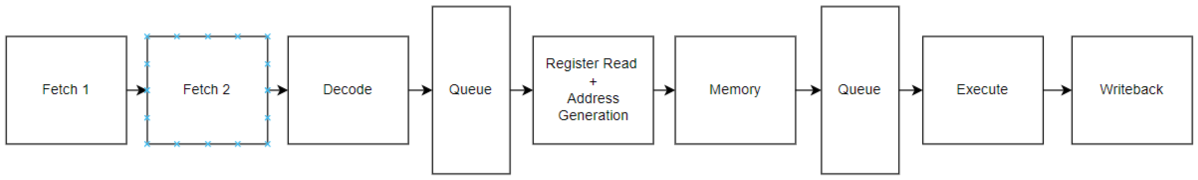
The project goals were to implement a subset of the x86 ISA using structural Verilog. On top of the general instructions that had to be implemented, a subset of prefixes also had to be supported including operand size override, segment register override, and the repeat string operation. Likewise, a total of four addressing modes were supported which were using immediates, registers, Mod R/M, and SIB for generating the desired addresses. While on the subject of addressing and memory, a functioning one level virtual memory system was implemented using a TLB with protections for segment size, page faults, and protection exceptions. The processor also successfully implements both interrupt and exception handling as well as the IRET instruction.

The high level design approach of the project was not on drastically reducing clock frequency, but instead removing as many possible dependencies and stalls within the pipeline as possible. This led to many added features that all helped reduce stalls. Some of the major ones included a G-Share branch predictor, full data forwarding, queued latches, non-blocking cache, etc. In general, a branch predictor removes a large stall whenever a branch occurs by continuing to execute instructions. A G-Share was chosen due to its higher accuracy when compared to a two bit saturating counter and could better handle the case of nested loops. Another large feature implemented was full data forwarding. This removed any stall from data dependencies so long as the value being waited on wasn't being used to generate an address. Another large feature that was added was queued latches between the DECODE stage and RRAG (Register Read, Address Generation) stage as well as a queued latch between the MEMORY stage and EXECUTE stage. These held instructions in case a stall ever occurred, whether it was a d-cache miss or a dependency stall in RRAG. However, the big help was allowing i-cache misses to occur earlier. If the d-cache ever missed and stalled, the FETCH and DECODE stages could still run and store instructions into the queued latches. This caused the i-cache to run ahead and have the misses occur at similar times as when the d-cache was stalled waiting for data.

Another large performance feature included was an non-blocking d-cache with a banked physical memory. This system allowed d-cache misses to be caught quickly without preventing the next instruction from being processed. With the banked PMEM, if two cache-lines missed in a row and a bank conflict in PMEM did not occur, both misses could be missed at the same time.

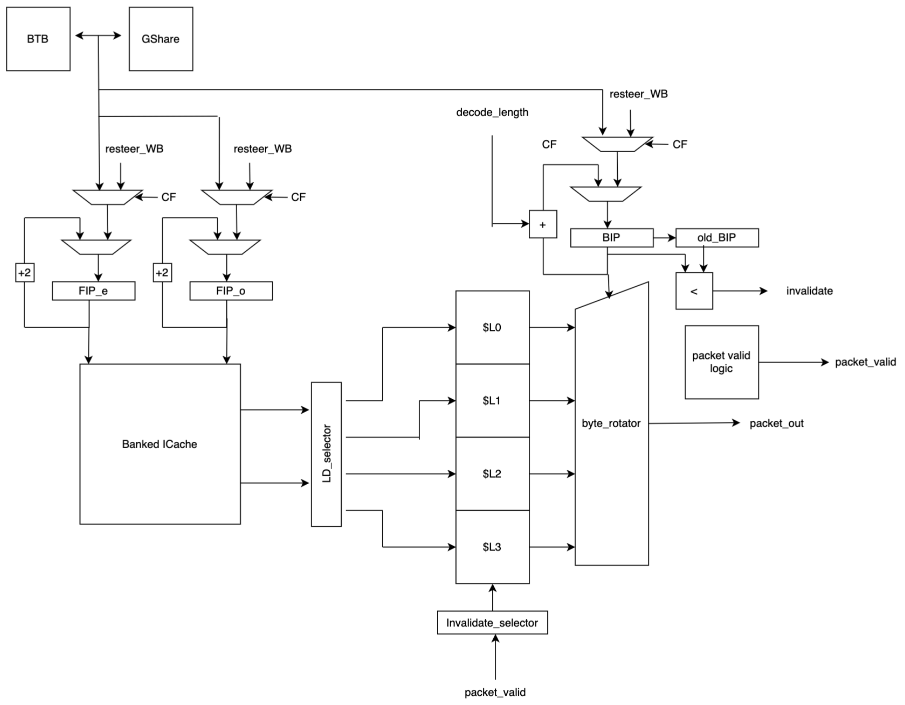
Design Overview

Our processor features a seven stage pipeline as follows:



We use latches between each stage to hold information between stages. In order to mask the effects of potential stalls originating from memory bottlenecks or occupied registers, we use queued latches in two places: between DECODE and RRAG and between MEMORY and EXECUTE. The queued latches have the ability to accumulate microcode from multiple instructions and fill in data as it becomes available from WRITEBACK (through data forwarding) or from physical memory.

Fetch (2-stage)



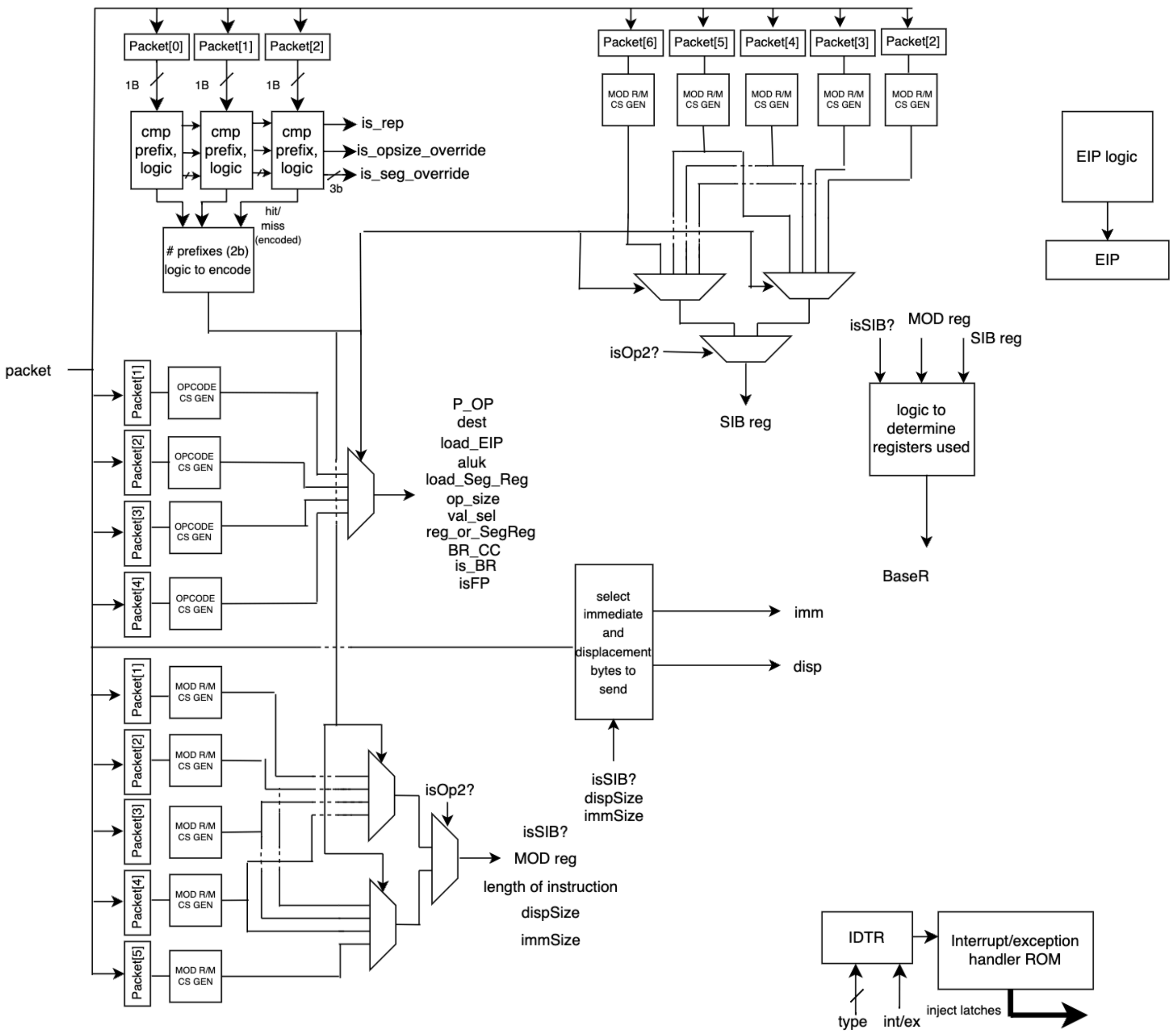
The fetch subsystem contains Fetch1, IBuffer, Fetch2, where IBuffer is the latch between the 2 stages. Fetch1 constantly sends the current FIP_o and FIP_e addresses to the ICache every cycle. However, ICache does invalidate multiple accesses to memory if the request has already been issued to reduce BUS hogging. FIP_o and FIP_e are their own registers both loaded by the same logic. There’s a mux that determines if there’s been a control flow change from a BP prediction or resteer from WB (done in priority of resteer then BP). This is then muxed between the FIP_x+2 or the CF value. This is then loaded into the register. Our ICache is banked by even and odd.

Once the packet is out of the ICache, the load selector determines which line of the IBuffer that is able to be loaded. The invalidate selector also determines which lines to invalidate this cycle. This invalidate logic is determined by a comparison between the bottom 4 bits old BIP and the current BIP. If the new is less than the old, then you invalidate.

Out of the IBuffer, you then rotate the entire buffer by all 6 bits of the BIP and pass that as the packet out. To determine if the packet is valid we check all the valid bits in the IBuffer, and if the current line we’re in (determined by current BIP) and the next line are both valid we can consider this rotation valid. This is to fend off against reading stale data.

Our BP was a GShare predictor, and our BTB was a 8 entry fully associative BTB.

Decode



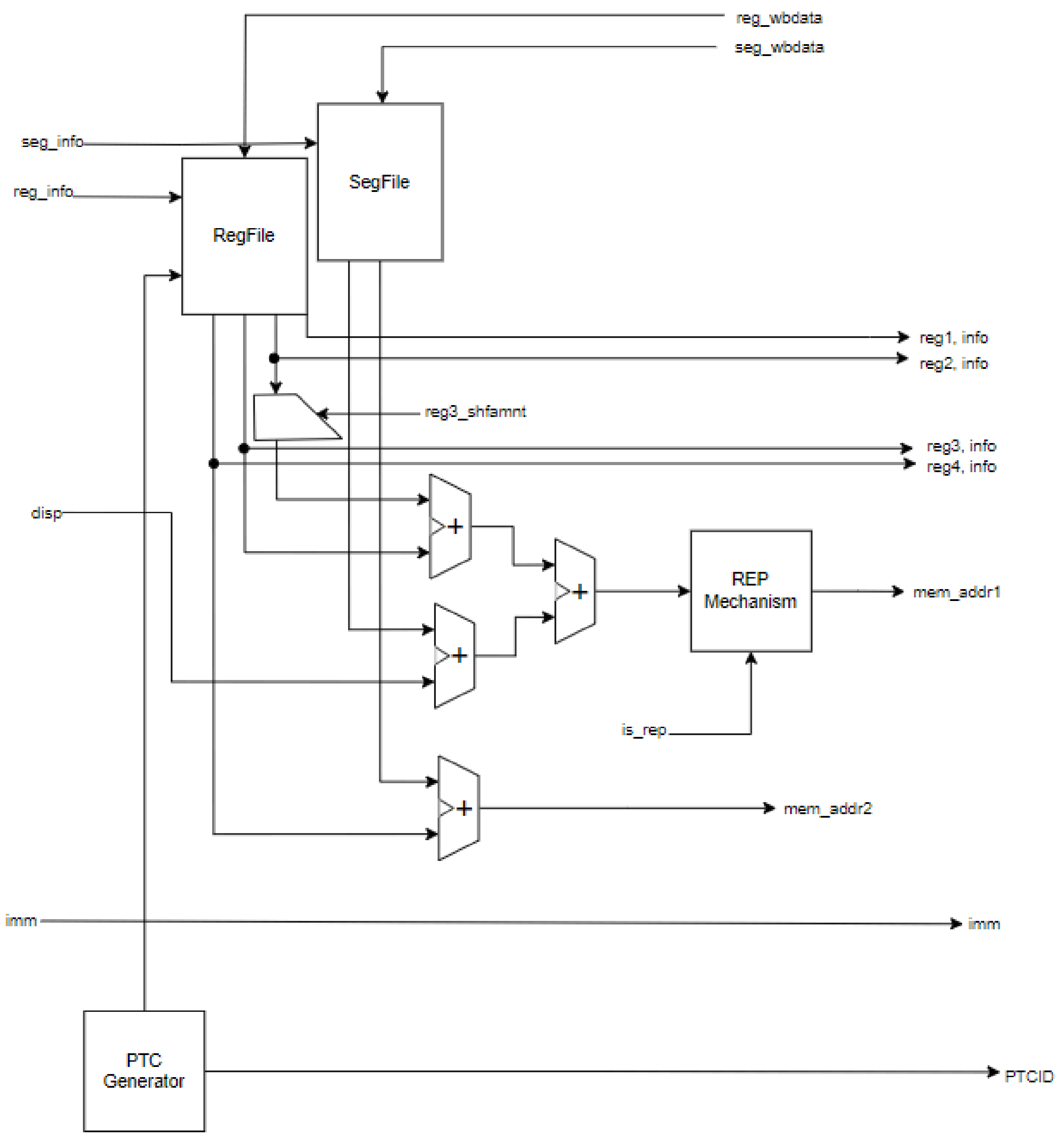
The decode stage takes in the packet from fetch and does a massively parallel decode on all the different locations the prefixes, opcode, modrm, and sib can start, and is muxed out whenever the more precedent byte is actually fully decoded. This was to cut back on time to get the length out of decode, so that we can pass it back to Fetch2 within the same cycle. The EIP lives in decode and is set through any control flow change and only loaded on valid packets and incremented by the decoded length.

To spell it out, there are 3 different prefix decodes (byte 0, byte 1, byte 2), 5 different opcode decodes (bytes 0/1, bytes 1/2, bytes 2/3, bytes 3/4), 5 different modrm decodes (byte 1, byte 2, byte 3, byte 4, byte 5), 5 different SIB decodes (byte 2, byte 3, byte 4, byte 5, byte 6). We then have immediate rotation for little endian. By the end of immediate decode we have the complete length of the instruction. Estimated 3.5ns to get the length.

The exception FSM kicks in when an interrupt or exception signal is issued out of WRITEBACK. The signal is a bit-vector with a bit set to indicate the type of interrupt or exception. The FSM will turn off the fetching mechanism of the I-Cache and instead begin pushing customized instructions to facilitate the switching process to go from normal execution to the interrupt service routine (ISR). Once the switch has occurred, fetch will be turned back on and begin executing the ISR. When the IRETD instruction at the end of the ISR is detected, the FSM kicks back in to facilitate the switch back to the normal execution of the program.

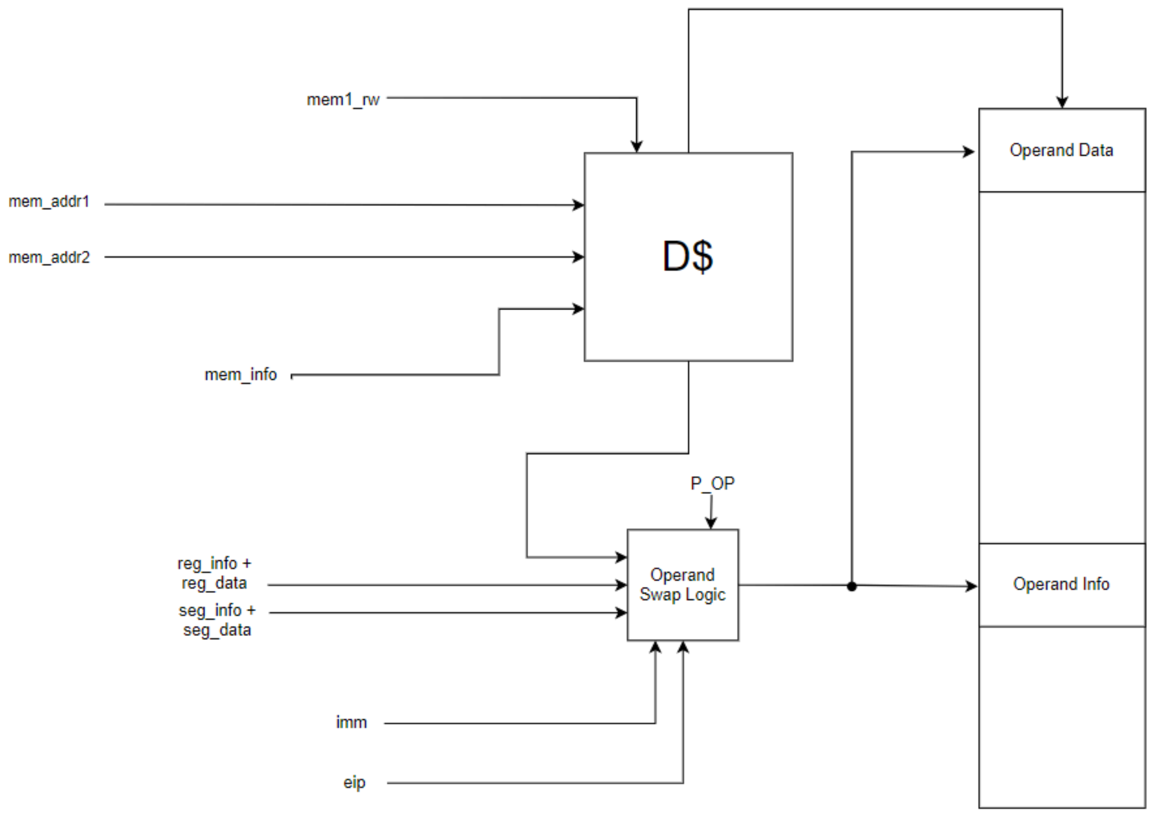
Register read & Address generation (RrAg)

The Register Read and Address Generation phase (RrAg) serves mainly to construct and pass forward any value that the rest of the pipeline will need for execution. There are five structures of interest in RrAg: the Regfile, the Segfile, address construction, the REP Mechanism, and PTC Generation. The Regfile and Segfile are 4-read port and 4-write port register files. The Segfile is simply composed of 6 16-bit registers. The Regfile must accommodate a larger number of operand sizes, so it has a more complex composition. The Regfile is capable of handling 8, 16, 32, and 64 bit values. The 64 bit registers are separated from the rest of the registers; the two parts of the Regfile are known as the GPRfile (for 8, 16, and 32 bit values) and the MMXfile (for 64 bit values). The MMX file is simply composed of 8 64 bit registers. The GPR file takes a size input to determine the size of values to use. The GPRfile is composed of 8 32-bit registers, and, if the size is 32 bits, it functions as a normal 32 bit register file. If the size is 16, then the GPRfile will output the lower 16 bits of the selected register, register selection works normally. If the size is 8, then the GPRfile will only look at the first 4 registers, so registers 0-3 will select the bottom 8 bits of registers 0-3 and registers 4-7 will select the second-from-the-bottom 8 bits of registers 0-3. Address construction is the network of adders, shifters, and muxes that build the mem1 and mem2 addresses for ModR/M-SIB and Stack addresses specifically. The REP Mechanism is our way of handling the REP prefix. The goal for REP was that, for the back-end, REP take the form of many instructions. The REP Mechanism will receive an input which determines if the instruction contains the REP prefix. If it does, the mechanism will continually increment, or decrement, mem_addr1, and turn the single REP instruction into many memory operations. The final structure is the PTC Generation block. PTC is our system of register renaming for the bypass network. PTC Generation assigns a unique ID to every operation in the pipeline. The Regfile and Segfile (and the Cache later on), will use these PTCIDs to determine if the data in the structure is up to date.


reg_wbdata
seg_wbdata
seg_info
reg_info
SegFile
RegFile
reg1, info
reg2, info
reg3_shfamnt
reg3, info
reg4, info
disp
+
+
+
REP Mechanism
mem_addr1
is_rep
+
mem_addr2
imm
imm
PTC Generator
PTCID

Memory

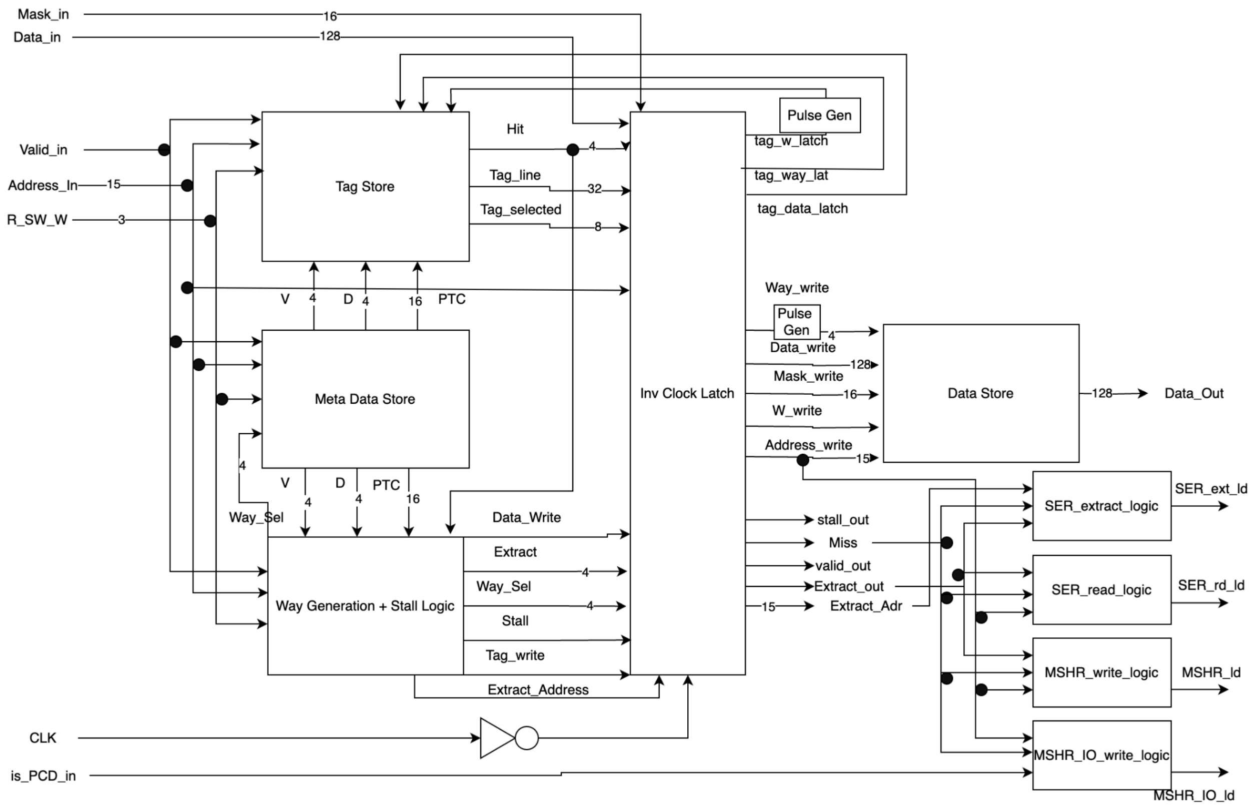
The Memory phase (Mem) has two purposes: retrieve data from Memory and to reduce the amount of data being sent to Ex. Data gets retrieved from memory using a Non-Blocking D$. The Queued Latch at the end of Mem serves as the Load-Store Queue. Additionally, the latch keeps track of whether the instruction in an entry of the latch is waiting on data to be retrieved from memory. If the instruction that is to be sent to Ex next is still waiting on data, the Q'd latch will push a bubble. Reducing the amount of data being sent to Ex is performed by the Operand Swap Logic block. Exiting RrAg, operand is stored in all possible forms that may be required at the end of the pipeline, but the maximum any one instruction may use is four. The Operand Swap Logic block serves to condense these many options into four that the instruction actually needs.



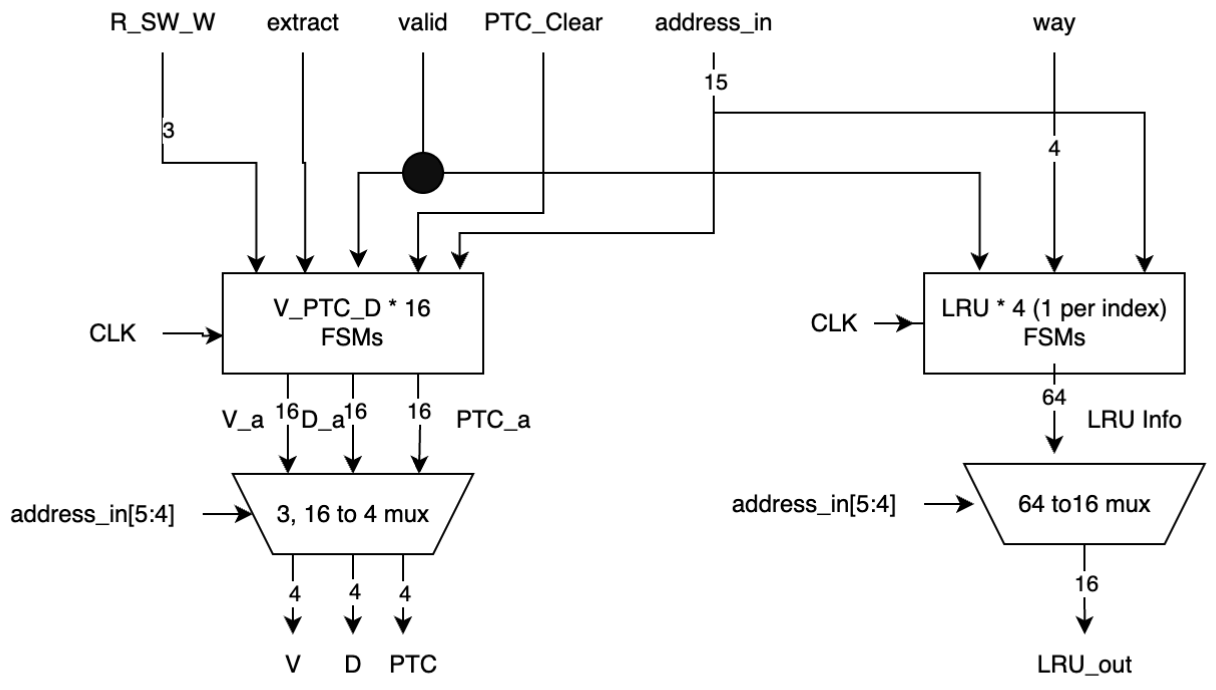
Cache

Cache Bank

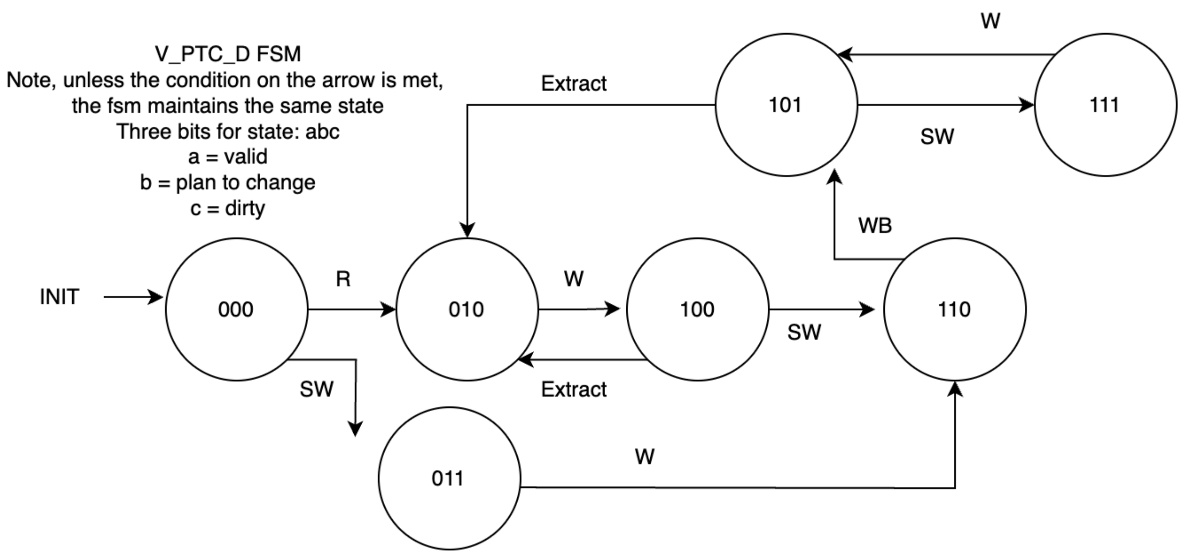
To start off, the cache bank was broken into 4 main parts: Tag Store, Meta Store, Data Store, and Way Generation. A brief schematic is shown below.



The Meta Store holds all the FSMs for controlling the cache bank. Each bank is composed of a 4 index, 4 way cache. Therefore, there are 32 total FSMs, 16 LRU FSMs and 16 V_PTC_D FSMs. The LRU FSMs were full LRU with a slight modification to handle PTC (Plan to Change) locations. If a location is PTC, the way is skipped and the next up in LRU is chosen. The PTC bit gets set whenever a soft-write goes through the cache. A soft-write occurs when you are going to write to a location after EXECUTE, but are reading from it in MEM first. The three state bits for the non-LRU FSM are the V bit, the PTC bit, and D bit. The V bit mark a cache location as holding valid data. The D bit is the dirty bit and gets set whenever a cache location gets written to. The PTC bit is set whenever a SW (soft-write) operation passes through the bank. A brief drawing of the meta store generation and the corresponding V_PTC_D FSM are shown below



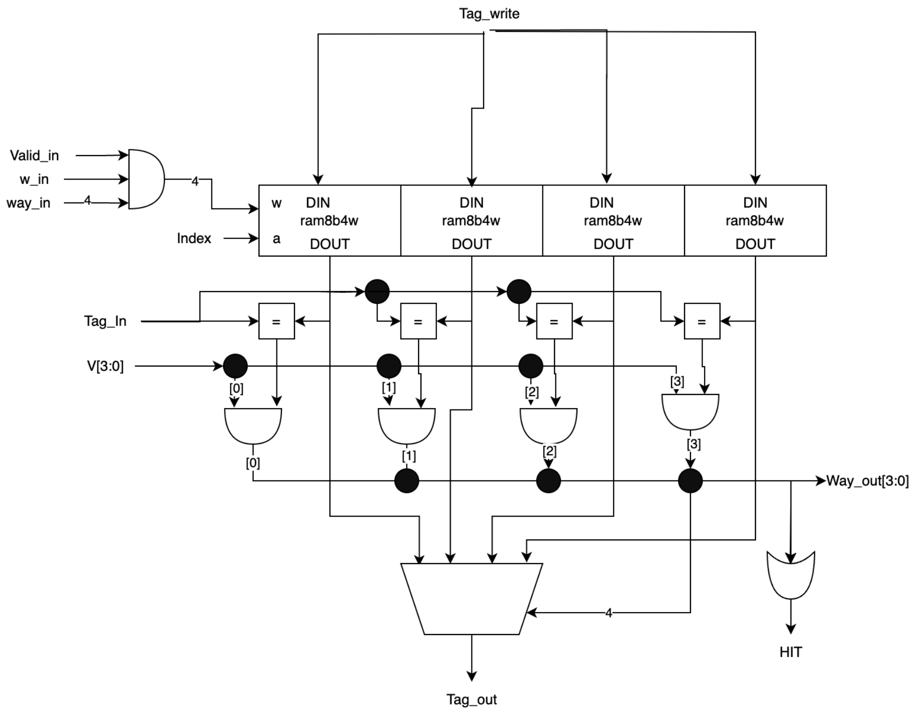
Meta Store Diagram



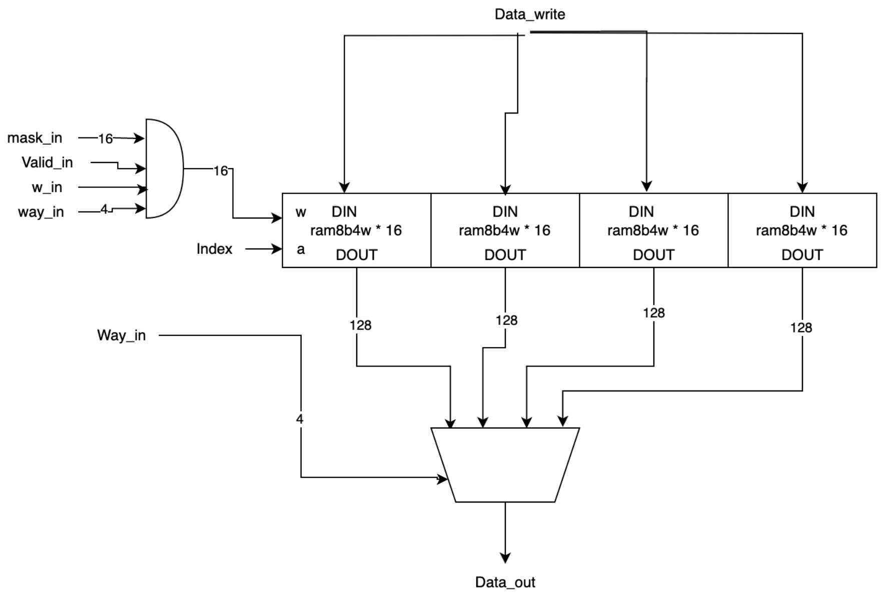
V_PTC_D FSM
Note, unless the condition on the arrow is met, the fsm maintains the same state
Three bits for state: abc
a = valid
b = plan to change
c = dirty
INIT
000
010
100
110
101
111
011
R
W
SW
Extract
WB

V_PTC_D FSM

The Tag Store acts as a regular Tag Store. The hit logic does need to be ANDed with the V bit of each way to guarantee proper operation on startup. A brief drawing of the tag store is below:

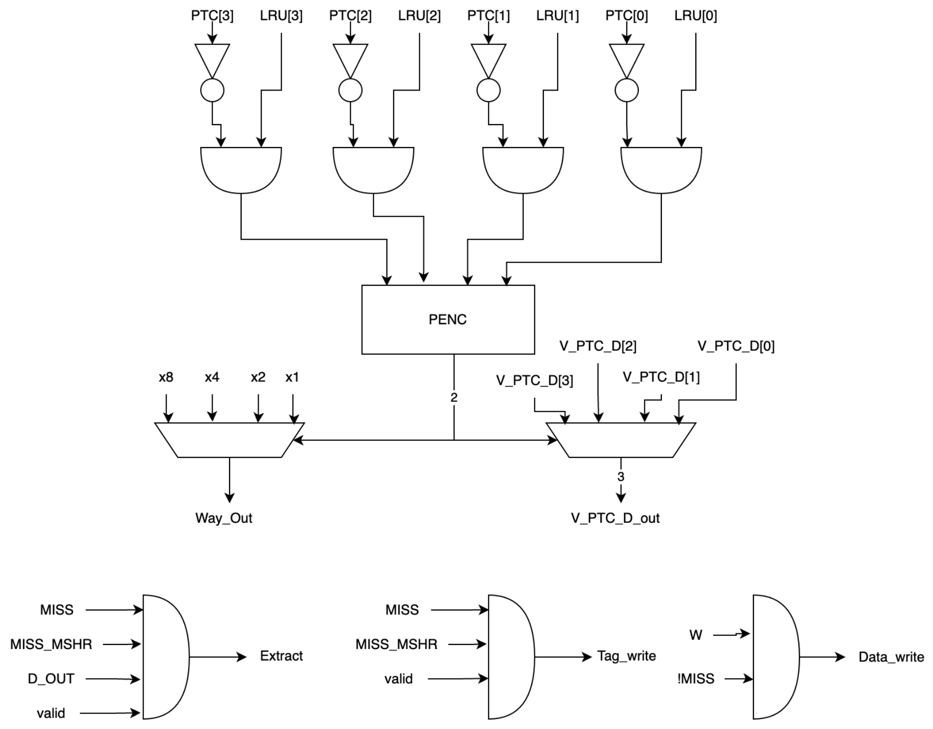


The Data Store also acts as a regular 4-way associative data store. The way must be passed in for both writing and reading, but for writing to occur, a MASK of valid bytes to write to must be written to the location.



Simplified Data Store Diagram

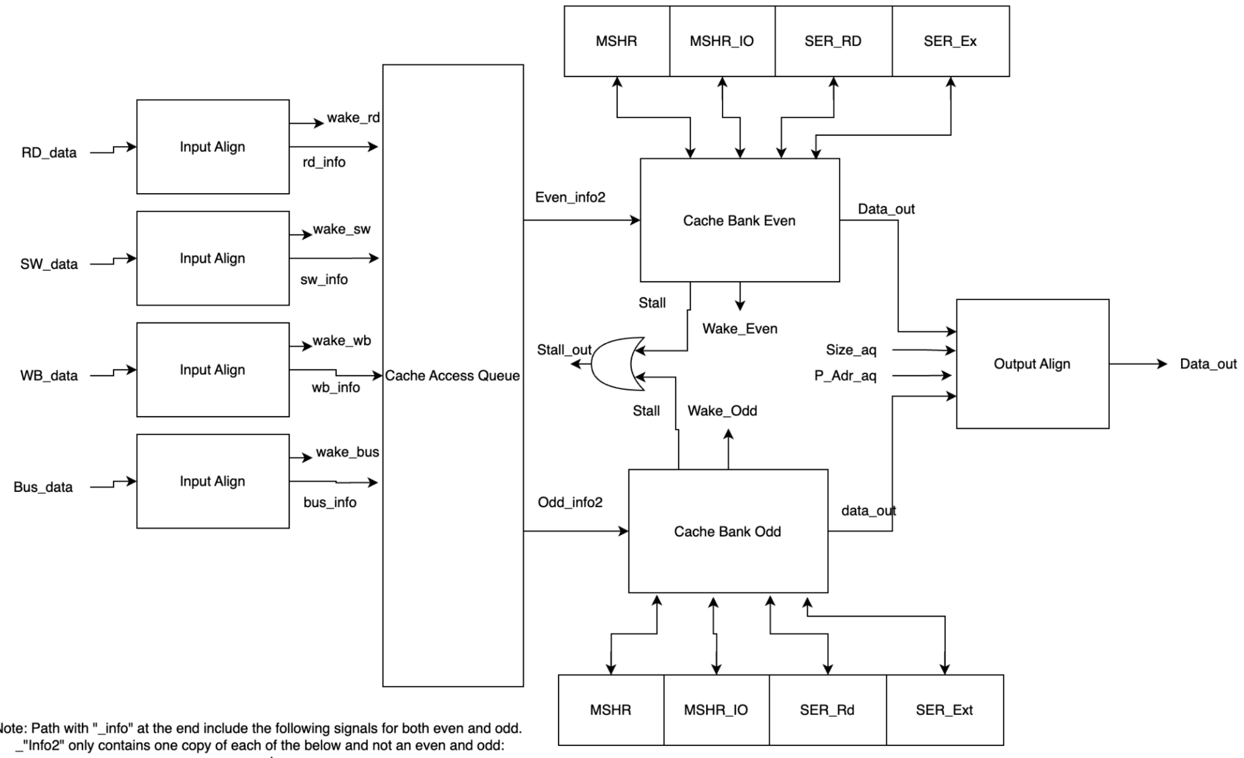
The final part of the cache bank is the way generation. This module contains the logic used to select the way used for both writing to the tag store and the data store. It relies on using the PTC and LRU bits to determine the optimal way to extract and write the data to. The logic is shown below



Way Generation Logic

D-Cache Subsystem

The D-Cache subsystem is broken into two cycles, separated by the cache access queue. One cycle to generate the physical address and check for exceptions, and one cycle to access into the actual cache bank. The most notable features of the D-Cache include its non-blocking feature, even/odd banked caches each with 256B of memory, a stride prefetcher, and single cycle pipelined unaligned memory access. A brief overview of the D-Cache is shown below:

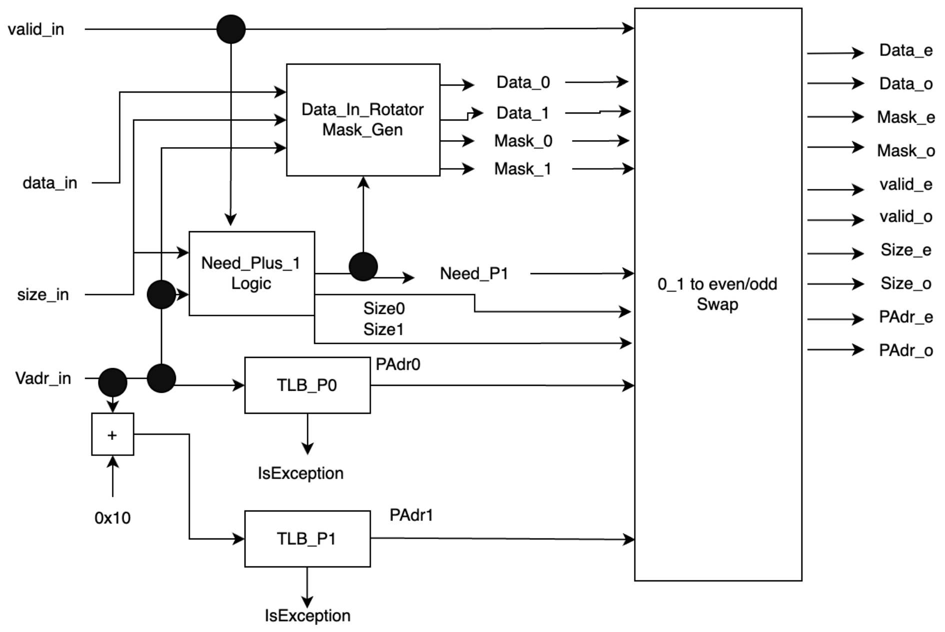


Note: Path with "_info" at the end include the following signals for both even and odd.
_"Info2" only contains one copy of each of the below and not an even and odd:
p_adr
data
size
r_w_sw
valid
fromBus
mask
ptcid
pcd

Note: Path with "_data" at the end include the following signals:
Data_in (128 bits)
V_address_in (32 bits)
size_in (2 bits)
r_w_sw (3 bits)
valid_in (1 bit)
fromBUS (1 bit)

In order to handle the non-blocking cache, each entry into the access queue is assigned a corresponding queued latch location with four wake bits, SW_even, SW_odd, RD_even, RD_odd. Until all 4 of these bits are set high, the instruction can't continue down the pipeline.

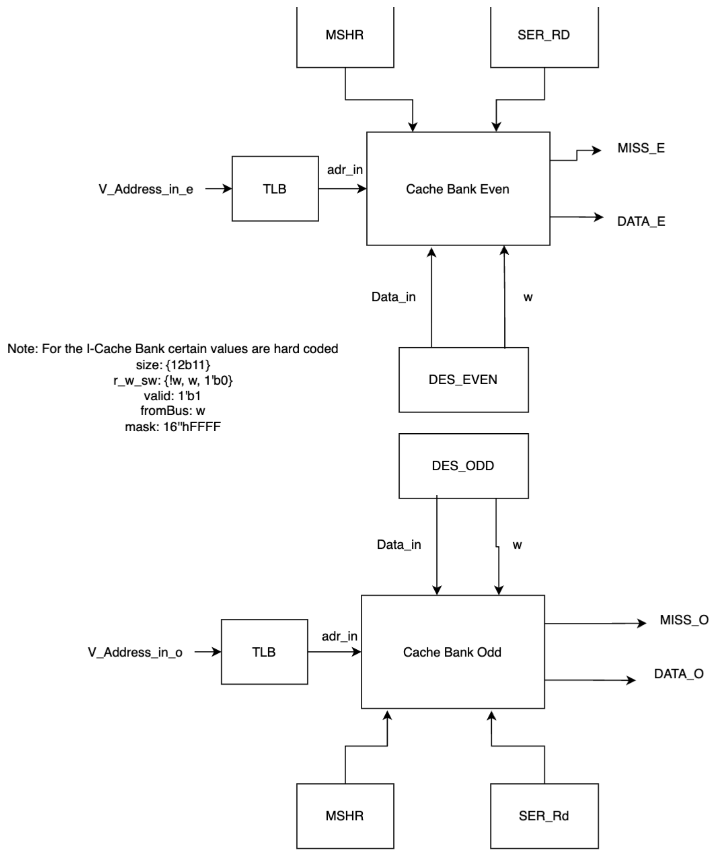
The first cycle of D-Cache involves the four possible input streams to the D-cache, Read requests, soft-write requests, writeback requests, and bus requests. All four paths get fed into the Input Align module. The input align module takes the inputted data (data_in, V_address, size, r_sw_w, valid_in, and fromBUS), and calculates V_address + 0x10, runs both addresses through the TLB, and uses the offset and size values to compute whether both even and odd accesses are needed. The wake bits also get set in the input align to account for whether or not a location is needed. If the second one is not needed, the corresponding wake bit is set high. In order to have a stride prefetcher, the valid bit is set even if the other bank is needed. However, since the wake bit gets set for the unused bank, it will not cause a stall should a miss occur. After this, the proper mask is generated. If an instruction only uses RD and not SW, both SW wake bits are set high and vice versa. A brief diagram of the input align is shown below:



All four input aligns get fed into the cache access queue, with the BUS IA having highest priority, followed by WB, SW, and then RD. From the access queue, the even values get fed to the even bank while the odd values get fed to the odd bank. These function as normal banks as shown above, and generate wake bits based on whether or not a hit occurs in the cache bank. After the bank, there is an output align module which shifts the two resulting cache lines and merges them together to output the proper 8-64 bit value requested.

I-Cache Subsystem

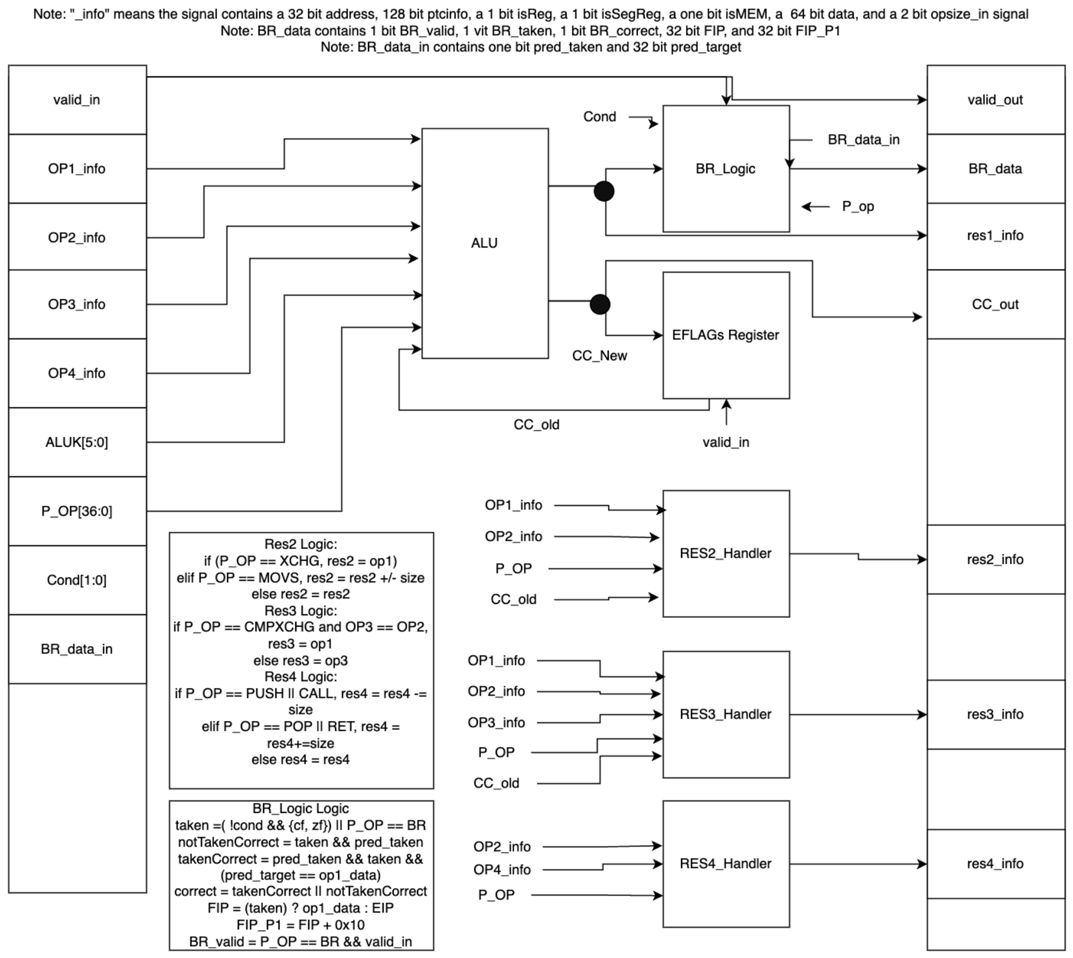
The I-Cache is a blocking cache with an even and odd banking. It uses the same banks as the D-Cache, but is made blocking by setting the MSHR to a size of one, so only one location can be missed at a time. Each bank has a total of 256B of memory.



An even and odd virtual address gets fed into the I-Cache which gets loaded into the TLB to create two physical addresses. The only outputs that get fed to Fetch are the MISS signal and DATA out signal. The write bit is tied directly to the DES full signal, and is otherwise a read. All operations can be performed in a single cycle.

Execute

The Execute stage's major components include the ALU, RES2-4 handler, BR_Logic, and the EFLAGs register file. A brief diagram is shown below:


Note: "_info" means the signal contains a 32 bit address, 128 bit ptcinfo, a 1 bit isReg, a 1 bit isSegReg, a one bit isMEM, a 64 bit data, and a 2 bit opsize_in signal
Note: BR_data contains 1 bit BR_valid, 1 vit BR_taken, 1 bit BR_correct, 32 bit FIP, and 32 bit FIP_P1
Note: BR_data_in contains one bit pred_taken and 32 bit pred_target
valid_in
OP1_info
OP2_info
OP3_info
OP4_info
ALUK[5:0]
P_OP[36:0]
Cond[1:0]
BR_data_in
ALU
Cond
BR_Logic
BR_data_in
P_op
EFLAGs Register
CC_New
CC_old
valid_in
RES2_Handler
RES3_Handler
RES4_Handler
OP1_info
OP2_info
OP3_info
OP4_info
P_OP
CC_old
valid_out
BR_data
res1_info
CC_out
res2_info
res3_info
res4_info

```
Res2 Logic:
if (P_OP == XCHG, res2 = op1)
elif P_OP == MOVS, res2 = res2 +/- size
else res2 = res2
Res3 Logic:
if P_OP == CMPXCHG and OP3 == OP2,
res3 = op1
else res3 = op3
Res4 Logic:
if P_OP == PUSH || CALL, res4 = res4 -=
size
elif P_OP == POP || RET, res4 =
res4+=size
else res4 = res4
```

```
BR_Logic Logic
taken =( !cond && {cf, zf}) || P_OP == BR
notTakenCorrect = taken && pred_taken
takenCorrect = pred_taken && taken &&
(pred_target == op1_data)
correct = takenCorrect || notTakenCorrect
FIP = (taken) ? op1_data : EIP
FIP_P1 = FIP + 0x10
BR_valid = P_OP == BR && valid_in
```

The ALU operates by having all 20 possible operations computed and their corresponding condition codes that each one computes. The results of all 20 are muxed together and chosen by the ALUK signal that is passed from the control store. The operations are shown below:

- AND, ADD, OR, 32-bit PENC, PASSA, PASSB, CLD, STD, CMPXCHG, DAA, NOT, PADDD, PADDW, PACKSSWB, PACKSSDW, PUNPCKHBW, PUNPCKW, SAR, SAL, CMOVC

The EFLAG register holds the current value of the EFLAGs as well as the old value of EFLAGs so that if an exception occurs, the bad operation can be rolled back. Aside from that, it is a regular register file that uses a MASK provided from the control store to determine which flags to update.

The RES2-4 handlers are for handling instructions that require incrementing the stack pointer through PUSH or POP or by using MOVS. The logic of each is shown in the diagram above, but it effectively uses the current stack pointer and increments/decrements based on the size of the memory operation.

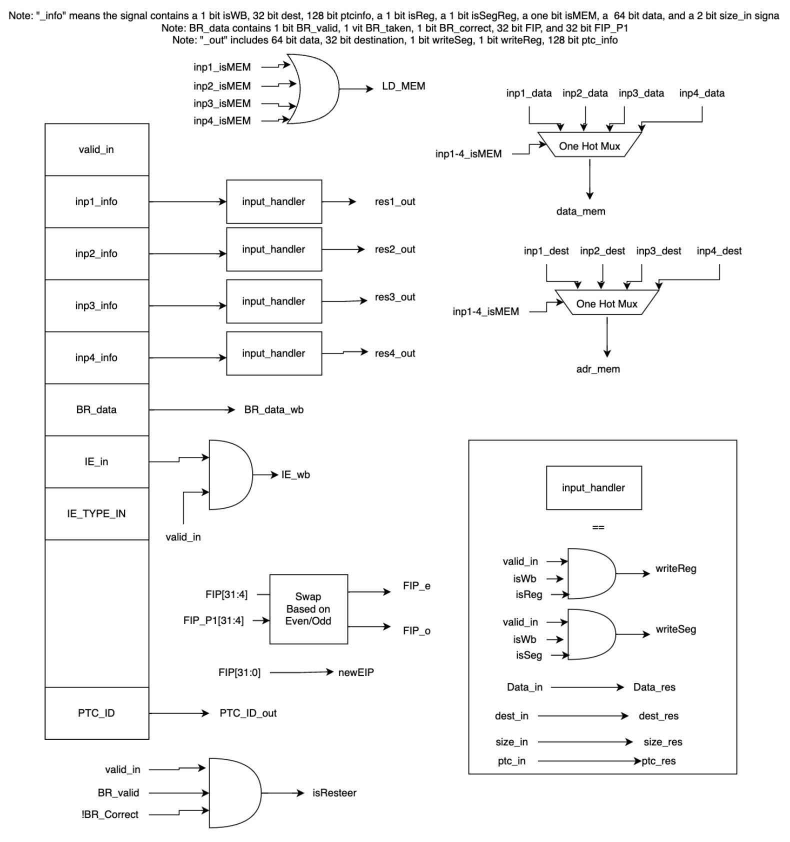
BR_Logic handles the computation for whether or not a branch occurs and whether the prediction was correct. If the prediction was correct, it passes through the predicted value. If it isn't, it passes the value computed by the output of the ALU. It passes bits forward that indicate whether the current instruction is a valid branch, the prediction was correct, and whether the branch was taken.

Writeback

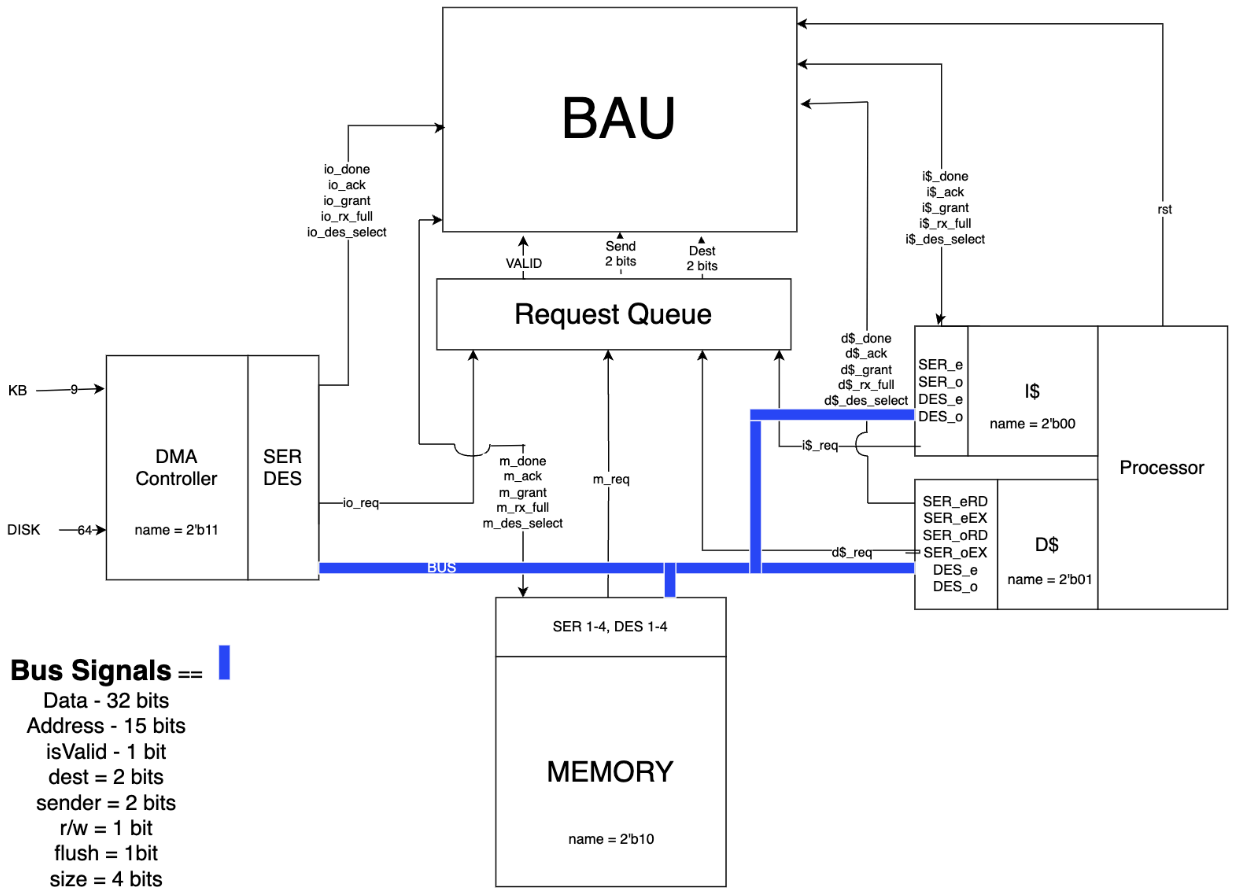
Overall, the writeback stage is relatively simple. Since the register file has 4 write ports, each inputted OP1-4 is tied directly to one write port. From there, each one computes a separate write signal by ANDing valid_in, isWB, and isReg. A similar process is done for writing to segment registers, save that instead of using isReg, isSeg is used. However, for writing back to the cache, only one write can occur to the writeback access queue. Since the writeback can occur at multiple locations, a one-hot mux using the isMEM signal of each OP1-4 to select is used to select the data written back as well as the address. Whether or not it is a valid write to memory is done by ORing all 4 isMem signals.

Resteer logic also occurs in writeback by ANDing the valid_in, BR_valid, and !BR_correct signal calculated in EXECUTE. If this is true, resteer occurs and the FIP's computed in EXECUTE get swapped to be even and odd oriented instead of FIP and FIP+1, as that is how the I-cache and Fetch handle it.

Other than that, exceptions are responded to as well. If IE_IN goes high when valid_in is high, an interrupt or exception is triggered depending on IE_TYPE_IN. From there, the FSM for handling IE takes over and all the writebacks for that instruction are canceled. A writeback diagram is shown below



BUS

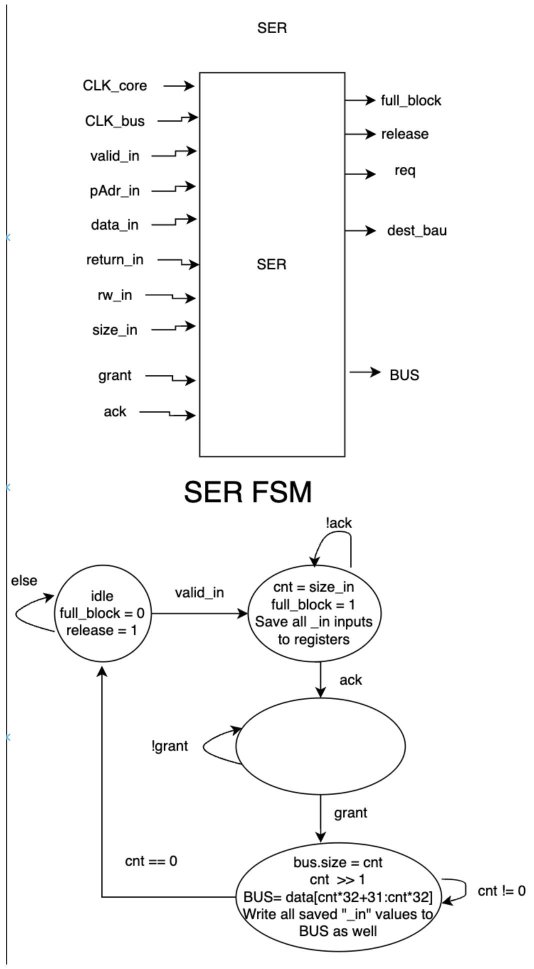


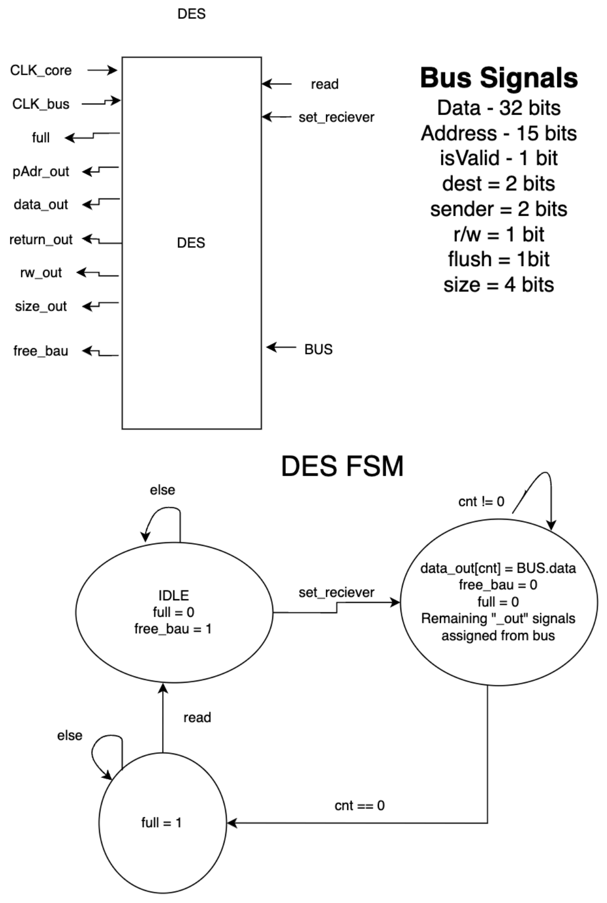
Overall Bus Architecture

The bus system is composed of three main components, the Bus Arbitration Unit (BAU), the Bus Request Queue (BRQ) and Serializer/Deserializer (SERDES). These three things work together to ensure that no one item can hog the bus and that selection occurs as quickly as possible. The main performance feature is the bus clock is clocked twice as fast as the core clock. This reduces the amount of time waiting for data to come back from PMEM and I/O.

SERDES

For bus communication, the SERIALIZER (SER) and DESERIALIZER (DES) work together to communicate with the SERs sending data to the DES of the corresponding location. The bits passed along the bus are 32 bits of data per cycle, a 15 bit address, a valid bit, a 2 bit destination location, a return destination, whether the operation is a read or write, and 4 size bits. In order to handle up to 16 bytes, the SER can hold the bus for up to four cycles. The input/output and FSM of the SER and DES is shown below.





Bus Signals

Data - 32 bits	
Address - 15 bits	
isValid - 1 bit	
dest = 2 bits	
sender = 2 bits	
r/w = 1 bit	
flush = 1bit	
size = 4 bits	

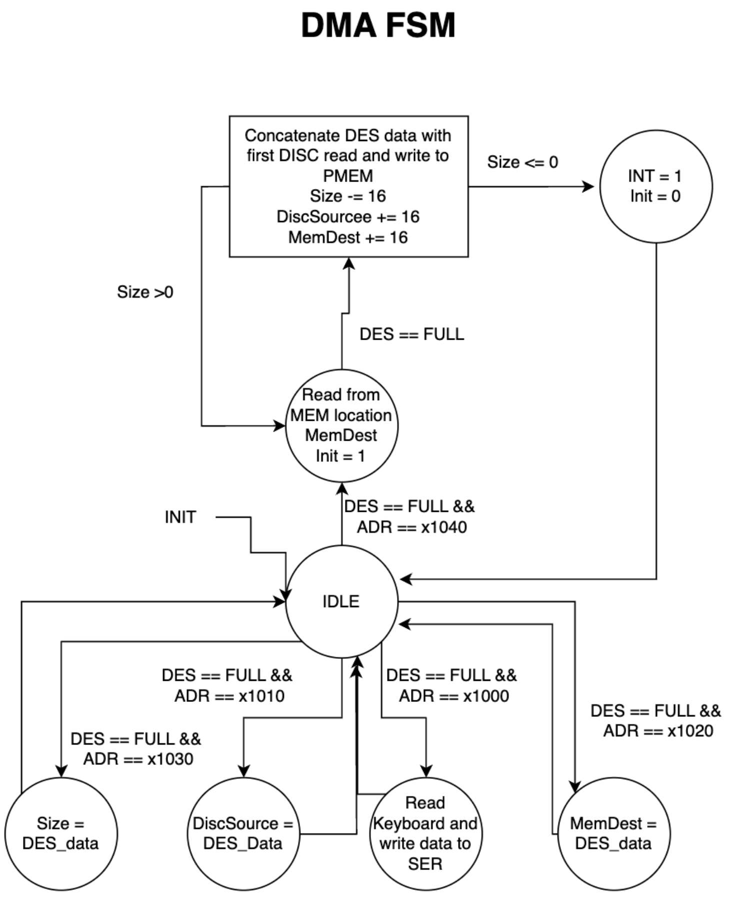
In order to have proper interaction with the core which is clocked at half the speed of the bus, the full signal of both the DES and SER can only be updated at the positive edge of the core clock. This ensures proper boundary crossing between the two clock domains.

I/O

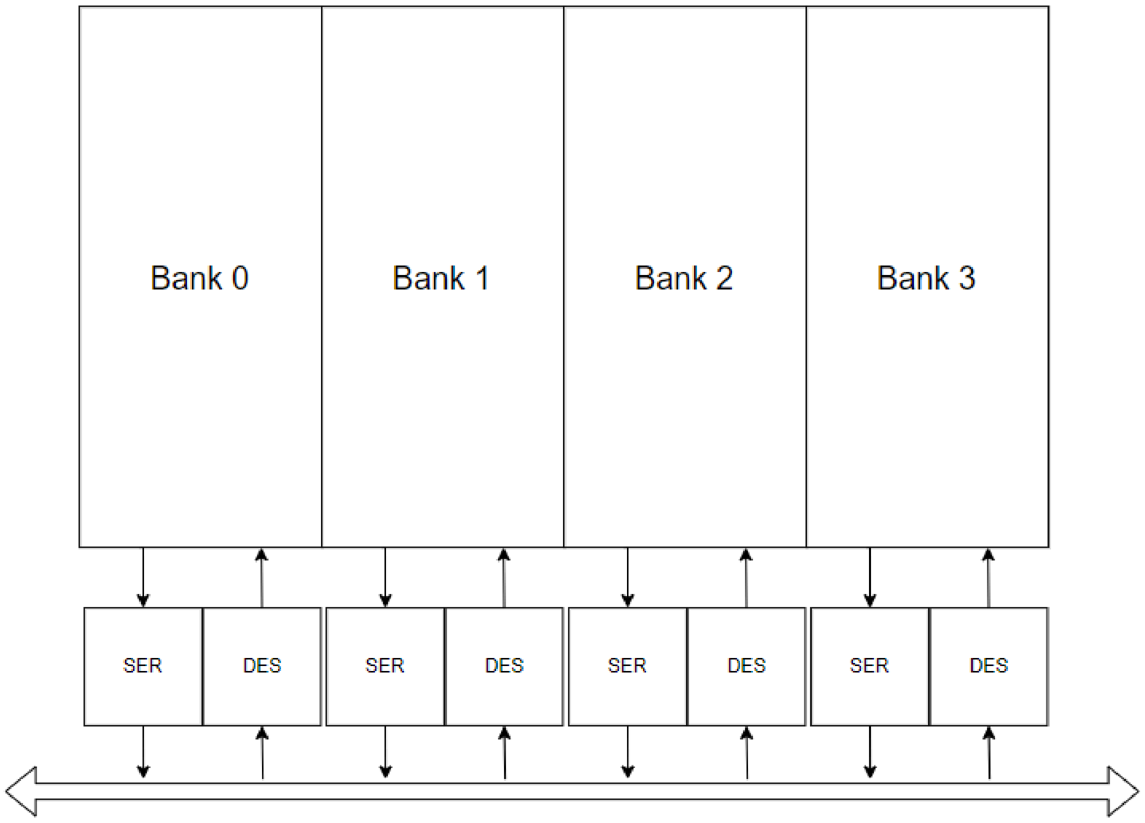
For the CPU, two I/O devices are supported: A keyboard and a disc. These are both controlled by a DMA controller that is able to interact with the core, and PMEM. The DMA controller is connected to the main bus using SERDES. In order to access the I/O, the PCD bit in the TLB must be set for the page that is trying to be accessed. The DMA controller contains five registers:

- Read Keyboard: x1000
- Disc Source Location: x1010
- Memory Destination: x1020
- Memory Read Size: x1030
- Initialize: x1040

By writing to these registers, the core can access all I/O devices. On the core side, a custom MSHR is added to handle I/O requests since the I/O locations can't be stored into the cache. Therefore, the MSHR_IO does not check for hits in itself before allocating a new entry. The FSM that controls the I/O is shown below:

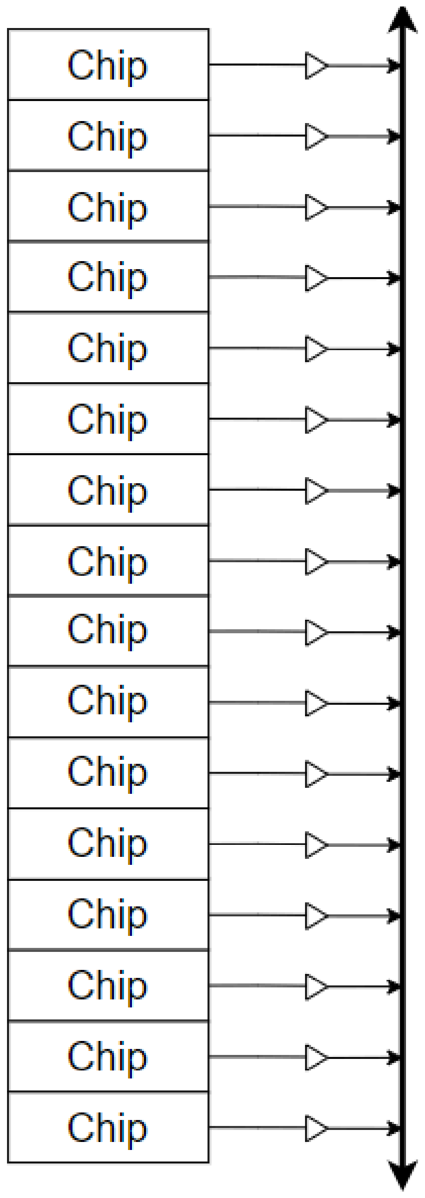

DMA FSM
Concatenate DES data with first DISC read and write to PMEM
Size -= 16
DiscSourcee += 16
MemDest += 16
Size <= 0
INT = 1
Init = 0
Size >0
DES == FULL
Read from MEM location MemDest
Init = 1
INIT
DES == FULL && ADR == x1040
IDLE
DES == FULL && ADR == x1010
DES == FULL && ADR == x1000
DES == FULL && ADR == x1020
DES == FULL && ADR == x1030
Size = DES_data
DiscSource = DES_Data
Read Keyboard and write data to SER
MemDest = DES_data

Physical Memory

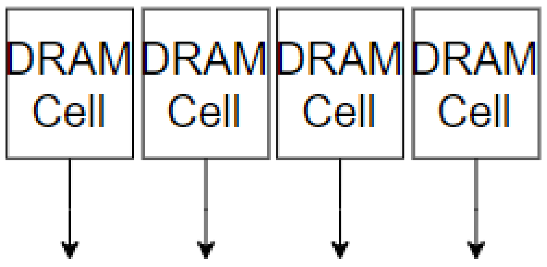


The 32kB physical memory of the processor is split into 4 banks, which are indexed by bits 4 and 5 of the physical address. Each row in each bank is 16 bytes wide, or the exact same amount as one cache access. Each access from memory will pull one full cache line back to the cache. As shown above, each bank has its own SERDES connected to it. This allows up to 4 different memory access requests to be done at the same time assuming no bank conflicts. As each SERDES uses 32B and there is a 128 byte limit, no buffering was allowed on the banks. However, to offset this, the physical memory was designed to be asynchronous and not require a clock. This allowed there to be exact timing to match what the DRAM cells required. While this improved performance, it did make adjusting the clock frequency harder as small changes on the SERDES could disrupt the timing.

Short of the banked PMEM and asynchronous memory, there weren't many other performance features. The main goal was to use fixed delays to perfectly match the timing requirements provided in the given library.



Example Bank



Interrupts and Exceptions

According to the project specifications, we support two types of exceptions: protection exceptions and page faults. Protection exceptions can originate from the TLB if an instruction attempts to write to a page that is read-only or if a memory address is past the bounds of a given segment boundary. Page faults occur when a page is not present in the TLB. To check if a memory address is past the segment boundary, we add the segment limit to the virtual segment base address and compare it with the virtual memory address. In the event of both exceptions occurring, the protection exception will take priority over the page fault. Exceptions are checked for in the EXECUTE and MEMORY stages. Interrupts are checked in WRITEBACK.

Once detected, the processor continues along as normal until the exception-causing instruction reaches WRITEBACK. The instruction is invalidated and an FSM begins injecting instructions into the pipeline to perform the switch into the ISR from the DECODE stage. The FSM pushes the EFLAGS, CS, and EIP of the exception-causing instruction first. Then it pushes two MOV instructions to move the contents of the interrupt descriptor table entry into temporary registers. A JMP instruction is pushed to load the CS and EIP with the proper contents to start the ISR. When the IRETD instruction at the end of the ISR is detected, the FSM kicks back in to facilitate the switch back to the normal execution of the program. Three POP instructions are issued to pop the EFLAGS, CS, and EIP back to their respective locations.

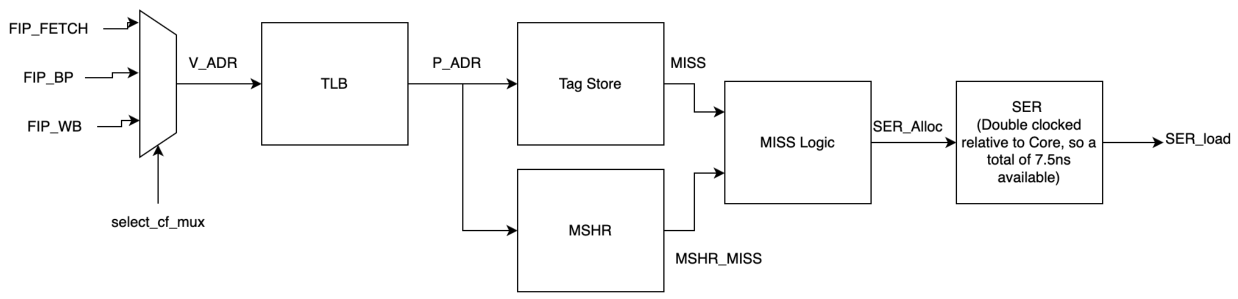
Design Considerations

Our goal for the project was to create a highly performant core within the constraints of the scope of the class. We wanted to have a fairly quick cycle time while still implementing aggressive performance features such as a g-share branch predictor and non-blocking cache. We especially wanted to reduce the impacts of stalls on the throughput of the core, with the idea that by the time an instruction reached the final WRITEBACK stage, the potential of a stall would be extremely rare. Another way we wanted to mask the effects of stalls on throughput was with the use of queued latches combined with data forwarding, which can hold microcode for multiple instructions and can update its contents as it becomes available from physical memory or through data forwarding. Combined with the non-blocking data cache, we hoped to cut down stall times significantly and improve throughput.

One tradeoff we saw potential for was with cycle time, especially with regards to cache. Because the complexity of the non-blocking cache would be significantly higher than a blocking cache, we were concerned that cycle time could increase significantly and affect performance despite the addition of the performance features. However, we felt that it would not reach a point where the added cycle time would overtake the benefits of the non-blocking cache.

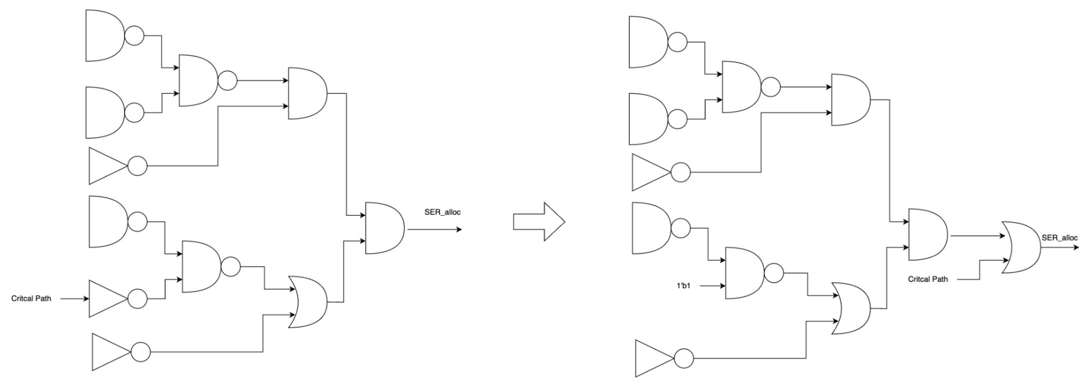
Critical Path Analysis

The critical path occurs within the fetch stage and follows the following path:



The critical path occurs in the fetch stage when going from the FIP (Fetch Instruction Pointer) selection between the BP, stored FIP, and Write back. The path occurs due to our bus being double clocked relative to the core, so the amount of time available on a cache miss is 7.5 ns. The initial mux is a 4x1 that changes at the positive edge of the clock, which has a 0.5 ns delay. The selected address is fed into the TLB and gets a physical address produced, which takes about 2.2 ns. This is followed by accessing the tag store and MSHR to determine whether or not a tag store hit or MSHR hit occurs. This also takes about 3 ns to compute, with reading the tag store taking up the lion's share of the time. From there, the MISS logic is computed to check whether or not there is a stall and the MISS is a valid operation. This takes 1.2 ns. After that, the required addresses and valid signal are passed to the SER and the SER must update the states and have all the valid inputs prior to the negative edge of the core clock. This final step takes 0.9 ns. This puts the critical path at 7.5 ns and prevents the clock from being faster.

However, initially, the logic didn't meet the 7.5 ns requirement. One big change was going through, allowing everything else to be computed for sending to the SER before the Tag Store MISS arrives since the MISS signal is the critical path. This saved 2 NAND gates and 1 AND gate from the critical path, cutting down the required time by 0.7ns for half a clock cycle,which in turn meant a 1.4 ns shorter clock cycle. An example is shown below.



Final Design Decisions

- Seven stage pipeline with 66MHz clock
- 4 port R/W register file
- 4 port R/W segment register file
- Register renaming for byte accurate data forwarding
- Partial Data-forwarding on register file
- Full Data-forwarding on caches
- G-share branch predictor with 8 entry BTB
- 8 entry TLB
- Two 8-entry Queued Latches
- 16 Byte Cache Line Size
- Even/Odd Banked I-Cache with 256B per bank, 4 Way Associative
- 2-Line prefetch for front-end
- Non-Blocking D-Cache
- Even/Odd Banked D-Cache with 256B per bank, 4 way Associative
- Single Cycle Cache Extraction
- D-Cache next line prefetcher
- D-Cache cache access queue to record up to 4 cache accesses per cycle
- Double bus clock frequency relative to core clock
- Bus Arbitration Unit skips busy SERDES and picks next highest priority
- MSHR can retire bus accesses out of order
- 4-way banked PMEM with 4 SERDES
- Asynchronous PMEM bank to perfectly meet timing requirements of DRAM cells

Conclusion

Overall, the core's performance was very good. On the three performance test cases, the branch predictor test case took roughly 5180 cycles, the memory latency testing took roughly 450 cycles, and the cache latency testing took roughly 600 cycles at a 15 ns clock (66.67 Mhz frequency). Our overall design had aggressive performance features at the cost of a faster clock. Overall, it was still a positive tradeoff as the non-blocking, prefetching caches (containing the critical path) more than offset the additional 2 ns it cost per clock cycle. 13 ns was the next best critical path on the decode stage.

In retrospect, the largest constraint faced was the time to complete the project. The chosen design, as seen above, was overly aggressive for completing within a single semester. While most of our changes saw a noticeable net increase in performance, the time to complete the branch predictor with proper resteering and non-blocking cache took significantly more time than expected. Since most of our features interworked with each other, like the queued latches and non-blocking caches, it was hard to accurately measure the effect of each individual performance feature. Likewise, another constraint was the clock frequency. The non-blocking cache was slow in order to handle single cycle cache access. One area we spent a lot of time on was the REP operation. This operation consumed a large amount of time in the optimization of it. There was an assumption that it would appear on a performance test case, but when it failed to show up, it cost us time that could have been spent tightening the clock frequency.

Another large constraint was the bus data width and amount of buffering on each side of the bus. Due to the data width only being ¼ of a cache line, it was impossible for us to have the cache never stall on misses because a miss could occur every cycle, but the SER could only send an eviction every 4 cycles. This was mitigated by doubling the frequency of the bus relative to the cache, which reduced this penalty to only 2 cycles to 1. Now, the stalling could have been minimized if there was enough buffering on each side of the bus. However, the 128B limit on each side of the bus of buffering removed any chance of extra buffering. This is because on the core side, there were four cache banks, each with their own DES. Each DES has to store upwards of 16 bytes, and when combined with them, it removes half of the available buffering. This is further compounded by the need for eviction on the D-cache, each SER for eviction requires 16B to hold the evicted cache line. In order to squeeze out slightly more performance, we realized for read requests towards memory, no data needs to be sent so we could add extra SERs only for reading and not eviction that don't contain any buffers so that an eviction and a read request could happen concurrently.

The main recommendation for the next design would be to optimize the cache such that the latch driven by the inverse clock isn't necessary. This would boost timing performance by roughly 15%. One change would be to add another stage in the cache that splits reading the tag store and modifying the data store. This would effectively remove any sort of critical path from the cache, at the expense of a slightly longer pipeline on cache accesses. Another change would be to move to a 3 stage FETCH, with an extra stage focused purely on address generation to send to the I-cache and could further reduce the time needed for the 13 ns second most critical path.

If the project had to be done again, one major change would be the order in which everything is done. Initially, we focused on implementing EXECUTE, WRITEBACK, the branch predictor, the register file, and decode. However, none of these could be integrated together and due to the out of sync of each, WRITEBACK had to be heavily redone. If it were to be done again, the main starting points would be the I-Cache, Fetch, the bus, and PMEM. This is because testing all other integration relies heavily on each one of these. Integration went rough because Fetch and the I-Cache were one of the last things to work and this severely limited the ability to test the remainder of the pipeline, especially the execute stage.

Appendix

Control Store:

Due to it taking 28 pages to print, a digital copy of the control store can be found here: https://drive.google.com/file/d/1_0APavNS47ipq6fVIf3bLMea7Pkmhmik/view?usp=drive_link

However the list of all signals stored in each instruction are:

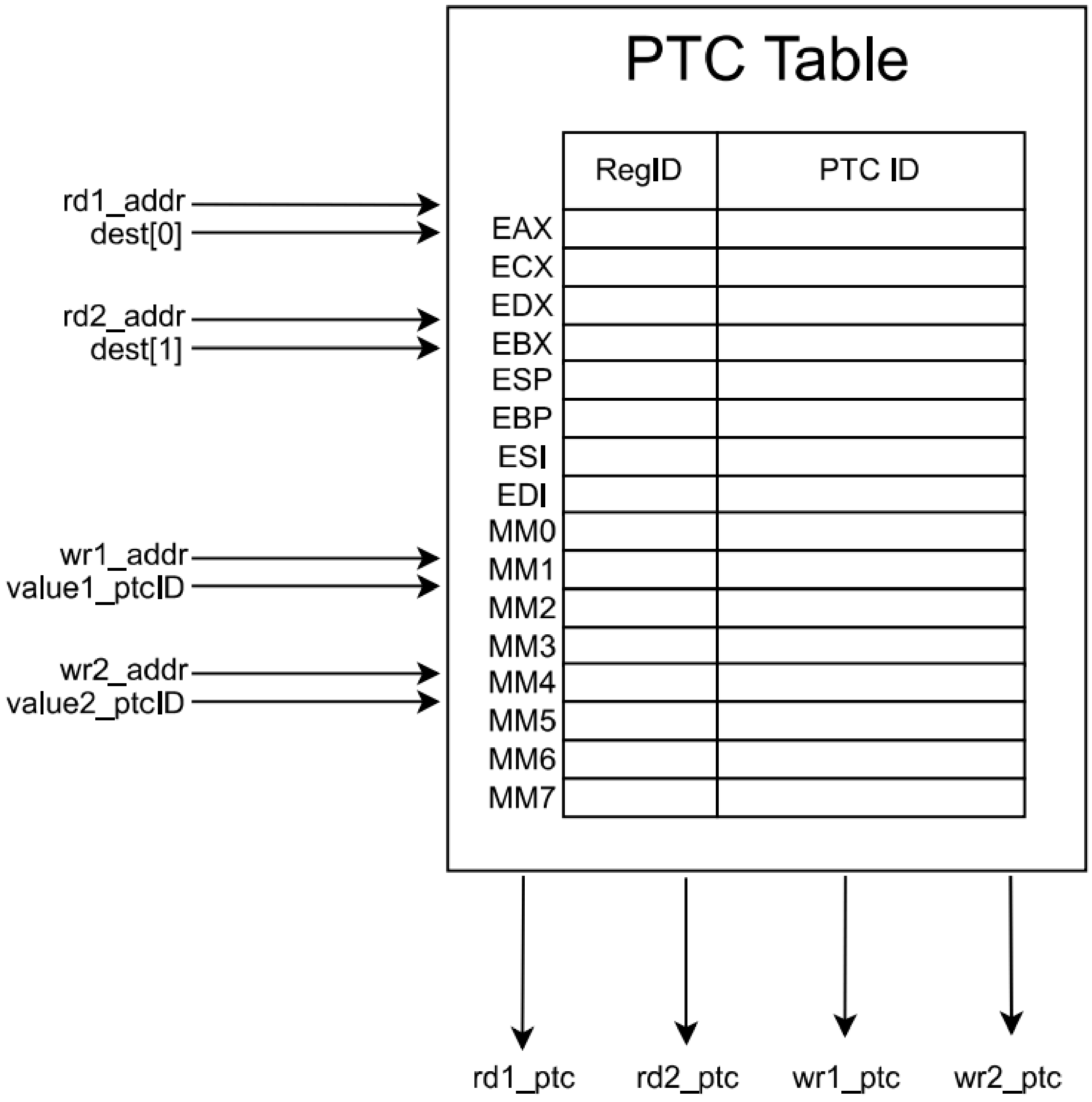
- OPCODE - 32 bits - used for determining which row of the control store to use
- isMOD - 1 bit - Tells whether the instruction has a MODR/M byte
- isDouble - 1 bit - Tells whether or not the opcode is two bytes
- OPCext - 8 bits - Second byte for double opcodes
- Aluk - 5 bits - Used to determine which operation to perform in EXECUTE
- MUX_SHIFT - 1 bit - Used to tell EXECUTE to shift by 1 for OP1
- P_OP - 37 bits - One hot signal that tells what type of operation is performed - See below for exact numbering
- FMASK - 17 bits - Flag mask for determining which EFLAGs get updated by the given operation
- Conditionals - 2 bits - For the conditional branches, these bits tell whether ZF OR CF needs to be true
- isBR - 1 bit - Tells whether an instruction causes a branch. Used in decode
- isImm - 1 bit - Tells decode that the instruction uses an immediate value
- immSize - 2 bits - Tells the size of the immediate
- Size - 2 bit - Tells the size of the registers that get accessed
- R1 - 3 bit - Specifies which register to read from for a regfile port
- R2 - 3 bit - Specifies which register to read from for a regfile port
- R3 - 3 bit - Specifies which register to read from for a regfile port
- R4 - 3 bit - Specifies which register to read from for a regfile port
- S1 - 3 bit - Specifies which register to read from for a segfile port
- S2 - 3 bit - Specifies which register to read from for a segfile port
- S3 - 3 bit - Specifies which register to read from for a segfile port
- S4 - 3 bit - Specifies which register to read from for a segfile port
- Op1_mux - 13 bit - Specifies which value to use for OP1, i.e. M1, M2, R1-R4, S1-S4, EIP, CS_EIP…
- Op2_mux - 13 bit - Specifies which value to use for OP2, i.e. M1, M2, R1-R4, S1-S4, EIP, CS_EIP…
- Op3_mux - 13 bit - Specifies which value to use for OP3, i.e. M1, M2, R1-R4, S1-S4, EIP, CS_EIP…
- Op4_mux - 13 bit - Specifies which value to use for OP1, i.e. M1, M2, R1-R4, S1-S4, EIP, CS_EIP…
- Dest1_mux - 13 bit - Specifies where to store OP1 at the end of the pipeline

- Dest2_mux - 13 bit - Specifies where to store OP2 at the end of the pipeline
- Dest3_mux - 13 bit - Specifies where to store OP3 at the end of the pipeline
- Dest4_mux - 13 bit - Specifies where to store OP4 at the end of the pipeline
- Op1_wb - 1 bit - Tells whether or not the given OP is written back at the end
- Op2_wb - 1 bit - Tells whether or not the given OP is written back at the end
- Op3_wb - 1 bit - Tells whether or not the given OP is written back at the end
- Op4_wb - 1 bit - Tells whether or not the given OP is written back at the end
- R1_MOD_OVR - 1 bit - Allows the R1 value to be overwritten
- M1_RW - 2 bit - Tells whether the first memory address reads or write from memory
- M2_RW - 2 bit - Tells whether the second memory address reads or write from memory
- OP_MOD_OVR - 2 bit - Tells which operand the MODR/M should overwrite
- S3_MOD_OVR - 1 bit - Allows the S3 value to be overridden
- memSizeOVR - 4 bit - Sets the size of the memory access if it should different than the operand size
- is_EXT - 1 bit - Tells whether its an extended opcode
- MOD_EXT - 3 bit - The 3 extra bits from MODR/M needed to determine the opcode if it is extended

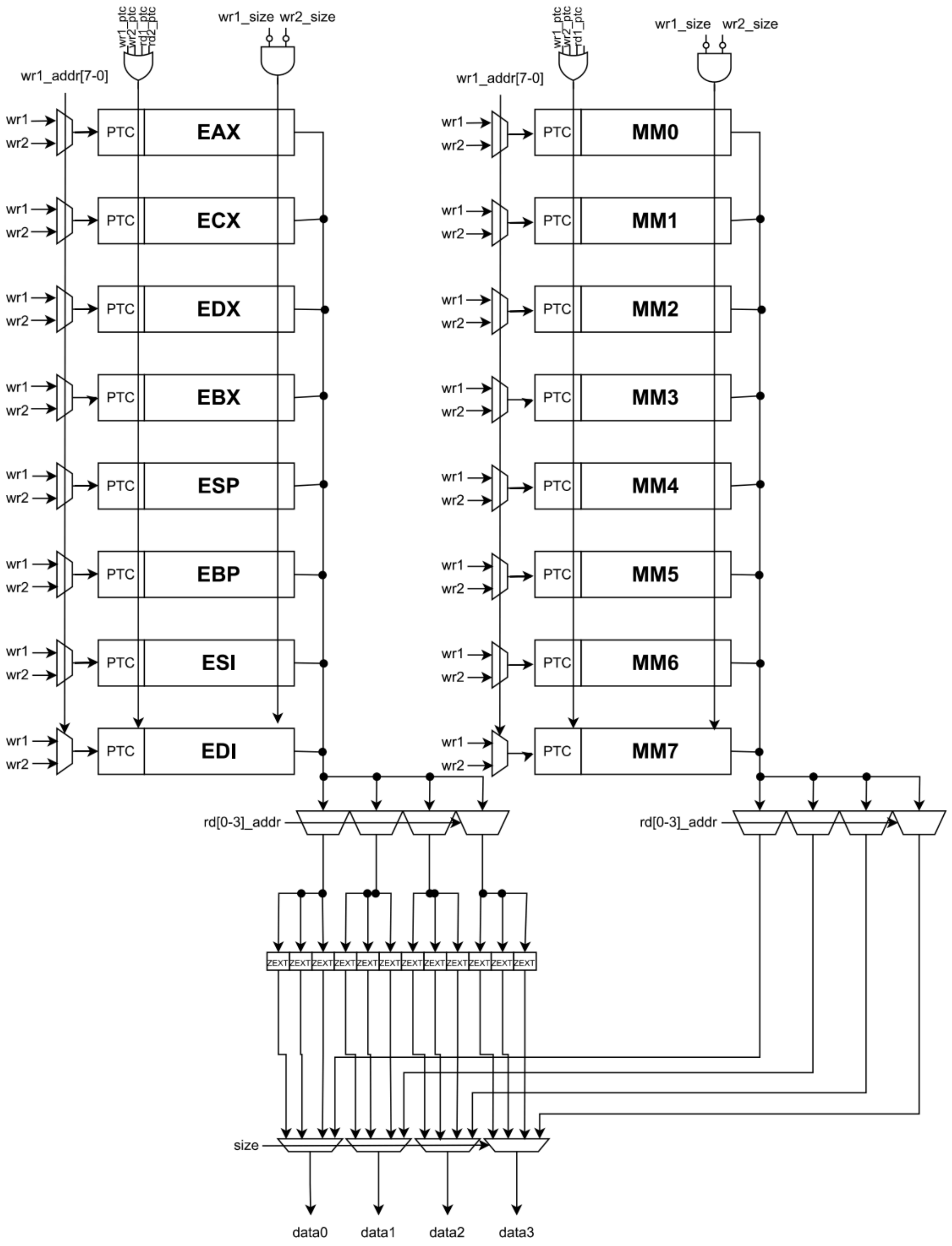
P_OP List:

0	ADD
1	AND
2	BSF
3	CALLnear
4	CLD
5	STD
6	CMOVC
7	CMPXCHG
8	DAA
9	HLT
0	IREtd
11	JMPnear
12	JMPfar
13	MOV
14	MOVQ
15	MOVS
16	NOT
17	OR
18	PADDW
19	PADDD
20	PACKSSWB
21	PACKSSDW
22	PUNPCKHBW

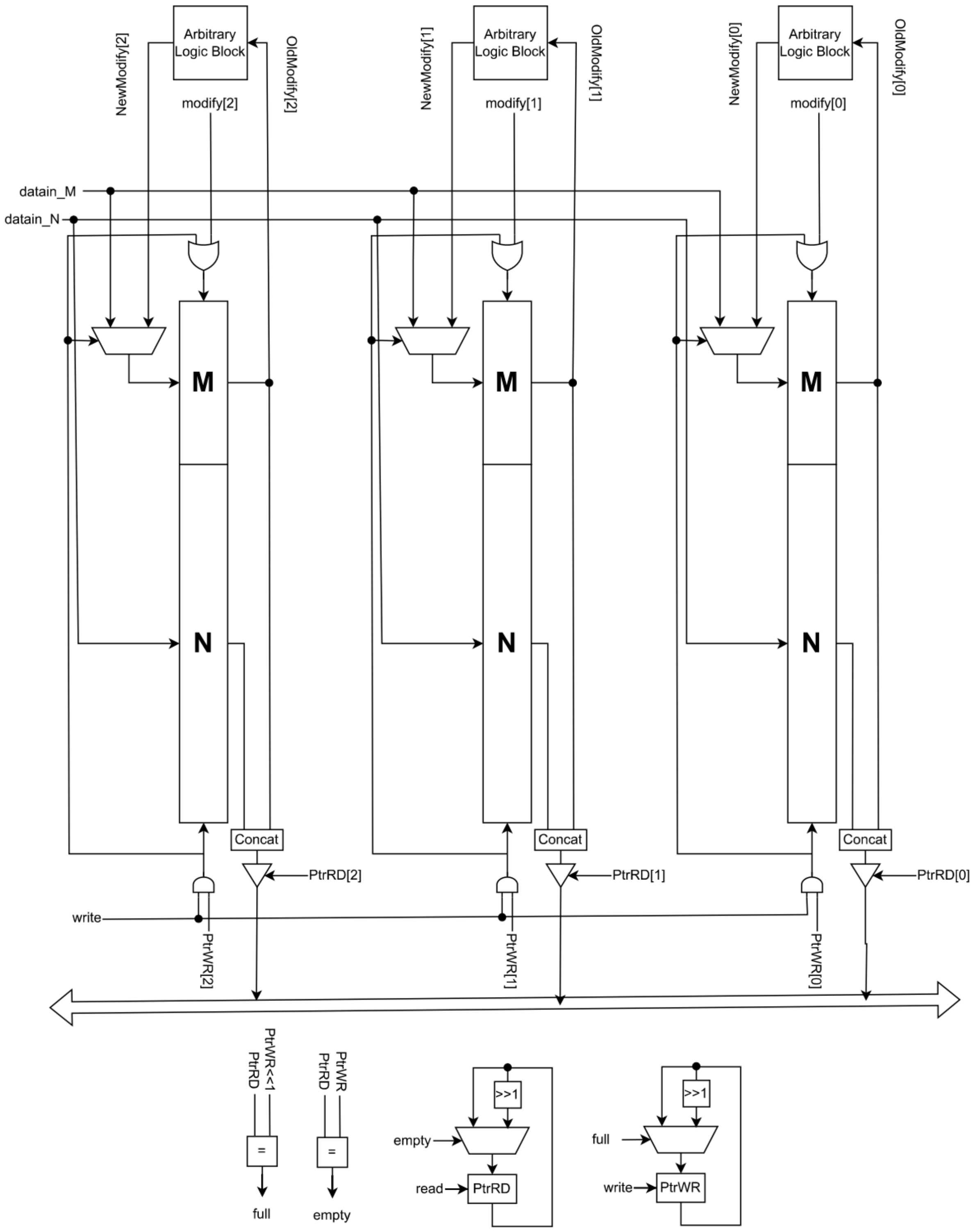
23 PUNPCKHWD
24 POP
25 POP_seg
26 PUSH
27 PUSH_seg
28 RET
29 SAL
30 SAR
31 STD
32 JMPptr
33 XCHG
34 CALLfar
35 call ptr
36 return ptr


PTC Table
RegID
PTC ID
rd1_addr
dest[0]
rd2_addr
dest[1]
wr1_addr
value1_ptcID
wr2_addr
value2_ptcID
EAX
ECX
EDX
EBX
ESP
EBP
ESI
EDI
MM0
MM1
MM2
MM3
MM4
MM5
MM6
MM7
rd1_ptc
rd2_ptc
wr1_ptc
wr2_ptc

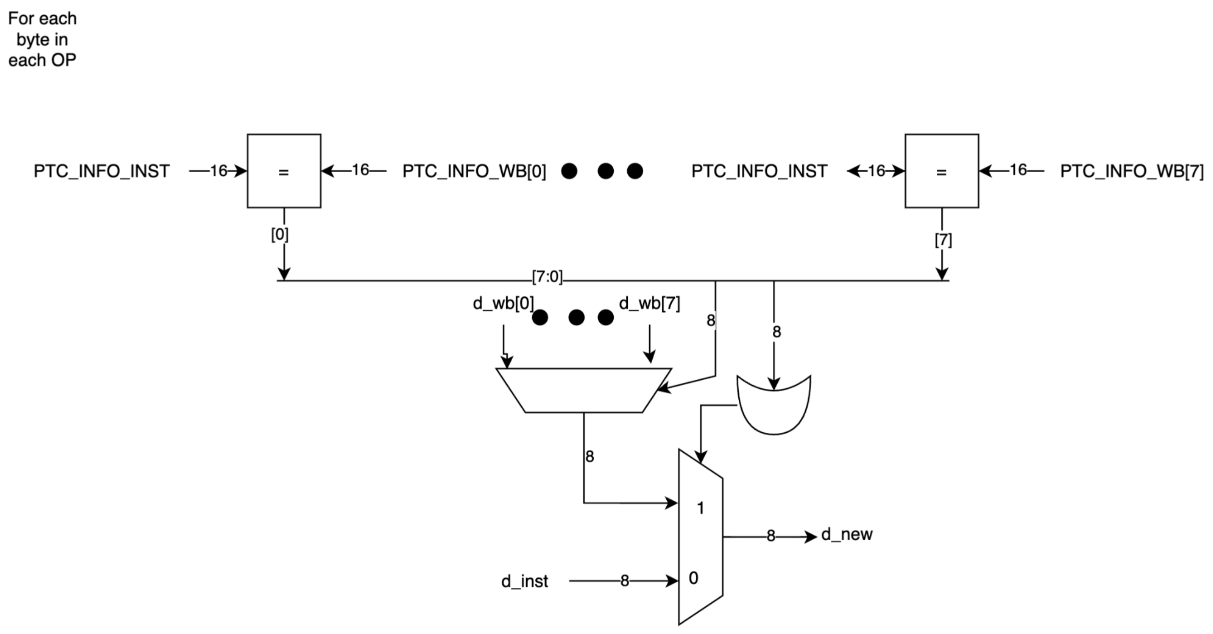
PTC Table



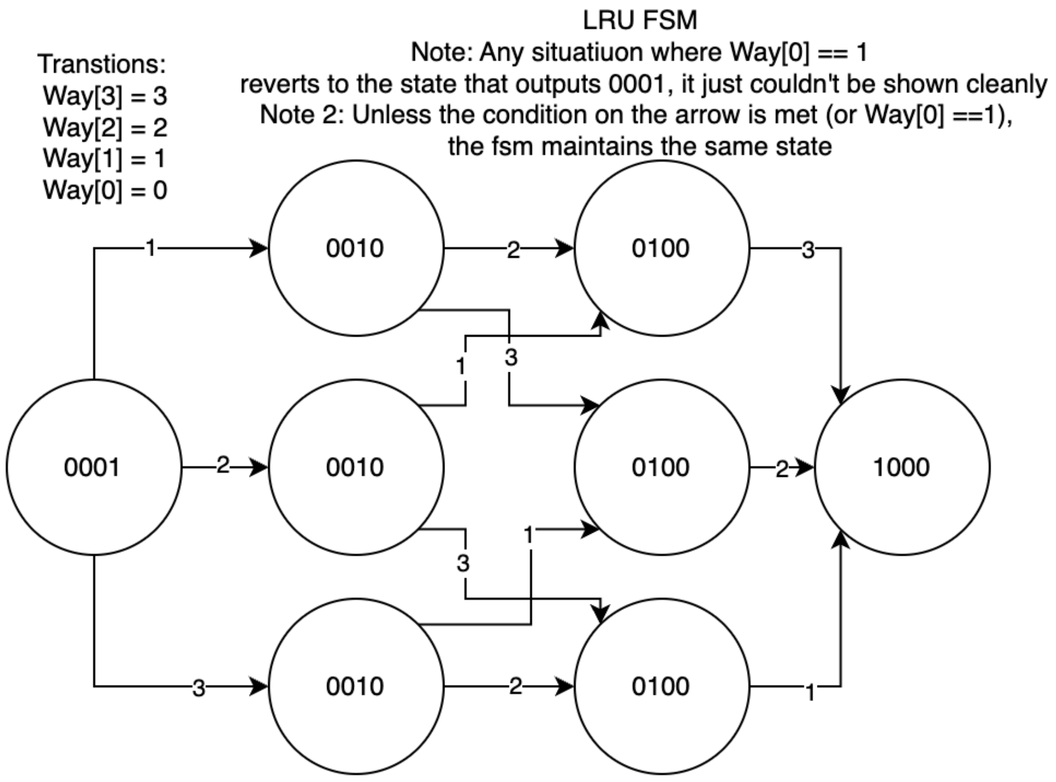
Register File



Queued Latches



Data Forwarding


LRU FSM
Note: Any situatiuon where Way[0] == 1
reverts to the state that outputs 0001, it just couldn't be shown cleanly
Note 2: Unless the condition on the arrow is met (or Way[0] ==1),
the fsm maintains the same state
Transtions:
Way[3] = 3
Way[2] = 2
Way[1] = 1
Way[0] = 0
0001
0010
0010
0010
0100
0100
0100
1000
1
2
3
2
3
1
3
1
3
1
2
2
1

LRU FSM