Indian Institute of Technology Kanpur
Department of Computer Science and Engineering

CovertCraft: FPGA Synthesis of a Covert Channel in Fully Associative Cache with Random Eviction

Dhruv Gupta - BT/CSE/220361

Pragati Agrawal - BT/CSE/220779

Supervisors: Prof. Mainak Chaudhuri & Prof. Debadatta Mishra

Mentor: Ms. Yashika Verma

November 12, 2024

Acknowledgements

We extend our heartfelt gratitude to *Prof. Mainak* and *Prof. Debadatta* for their invaluable guidance and unwavering support throughout this project, especially for their commitment to regular meetings despite their busy schedules.

We also wish to express our sincere thanks to our mentor, *Ms.Yashika*, for her in-depth knowledge and continuous assistance, which greatly enriched our understanding of the subject matter. CovertCraft is inspired from her research work ***LeakyRand***. Additionally, we appreciate the cooperation of the CSE department staff, whose logistical support was essential to the success of our project.

Abstract

The last-level cache (LLC) in a chip-multiprocessor is typically shared between the cores of the processor. Such a shared LLC has been shown to be vulnerable against timing channel-based security attacks that exploit the latency difference between an LLC hit and an LLC miss. Two processes (sender and receiver) co-scheduled on two cores of a chip-multiprocessor can exploit this latency difference to establish a covert channel, an illegitimate and covert way of leaking sensitive information. Recent research has proposed the fully associative cache with random replacement policy as a security solution to protect against conflict-based side channels in shared LLCs. In this project, we synthesize CovertCraft, a covert channel that exploits a fully associative cache with random replacement, on a Spartan-3E FPGA board. This synthesis effort serves as a proof of concept to demonstrate that a fully associative cache exercising the random replacement policy is still vulnerable to timing-based covert channel attacks. Our design includes a fully associative cache with random replacement and two finite state machines to act as the sender and receiver processes. The synthesized channel achieves a communication bandwidth as high as **1 Mbps** on a small 16-entry fully associative cache while maintaining a near-zero expected bit error rate.

Contents

1. Introduction

A covert channel is an illegitimate medium of communication between two entities, where information is transferred through a method not originally intended or designed for communication. Researches have shown that such covert channels can be created in a multi-core system by exploiting the resources which are shared across the cores. In multi-core systems, the last level cache (LLC) is typically shared across the cores. Two processes can exploit the latency difference between a LLC hit and a LLC miss to set up a covert channel and use it to leak sensitive information. Researches have discovered multiple strategies to establish such covert channels. Prime+Probe is one such strategy[1]. However, in this strategy it is necessary for the receiver to get a near 100% occupancy in the cache efficiently. Efficiently achieving this in a cache with LRU replacement policy is possible since it can be ensured that receiver does not self evict its blocks while filling up the cache with its addresses in the "Prime" step. Also, if the cache size is large, filling up all the cache blocks may become inefficient. Thus, typically the sender and receiver isolate a *set* of the cache for communication instead of filling up the complete cache. Due to these conditions, researches have suggested a fully associative cache with random replacement policy as a security solution to protect against such channels[3].
In this project, **our aim is to synthesize an efficient covert channel - CovertCraft which exploits a fully associative cache with random replacement policy.**

In this report we first discuss the protocol of communication that we have used (section 2). Then we discuss our implementation of this protocol, including the design and detailed description of every module (section 3). Then we discuss the assumptions we have made and limitations in our current design (section 4). Then we explain the input protocol, i.e., how input has to be provided using the FPGA board for demonstration (section 5). At last we evaluate our channel's efficiency, mentioning all the observation and results (section 6). We have also provided a glossary at the end for various terms that we have used in the report. The code for the implementation can be found at - .

2. Communication Protocol

The communication protocol used by the sender and receiver is as follows:

1. **Cache Region Fill step (CRFill):** In this step, the receiver accesses non-overlapping[1] addresses defined by the occupancy sequence and tries to occupy the entire LLC. This step happens only once during the entire communication and it takes crf-total-cycles time. During this time sender waits.

2. **Cache Region Probe step (CRProbe):** In this step, the receiver finds out how many blocks of the cache it has been able to occupy in the CRFill step. For this, it performs consecutive lookup and then flush of all the addresses from the occupancy sequence. But here the addresses are accessed in the reverse order, i.e. it starts from the last address it had accessed during CRFill and goes up to the first address it had accessed. This is done to ensure that the first CRProbe lookup certainly hits the cache and we know that the last address we had accessed must be present in the cache. So the first CRProbe lookup sets occupancy to 1. If any further lookup hits the cache, we increment the occupancy and a new hole is created due to the flush we perform after every lookup. Otherwise, if it is a cache miss, then it must go in the hole created by its preceding address, and here we do not increment the occupancy. This step takes crp-cycles0 time, which is equal to *crp_blocks * (MISS_LATENCY + HIT_LATENCY)*, since for each block we perform a lookup, which takes *MISS_LATENCY* in the worst case, and a flush which takes *HIT_LATENCY*. So at the end of all these accesses, we will have the same number of holes in the cache as its occupancy by the receiver.
   Now, the receiver fills these blocks with its addresses from the receiver probe set. This step takes crp-cycles1 time = *NUM_BLOCKS * MISS_LATENCY*, since all these lookups will surely be a cache miss. The maximum occupancy the receiver can achieve is num-blocks, so the worst-case time is assumed. The CRProbe step overall takes crp-total-cycles time and during this entire time, the sender again waits.

3. **Communication:** Now the sender starts sending communication bits as follows: If the bit to be sent is 0, the sender does not create any disturbance in the LLC. Otherwise, to send a bit 1, the sender performs a series of accesses of addresses from the disturbance set. Then it flushes all the addresses it had accessed. This step would create atleast 1 hole and atmost sndr-probe-blocks number of holes in the cache. This step would take recv-wait-cycles = sndr-probe-blocks*(MISS_LATENCY + HIT_LATENCY), because each access would take *MISS_LATENCY* and each flush would take *HIT_LATENCY* time. During this the receiver waits.
   Now the receiver accesses all the addresses from the receiver probe set. If atleast 1 address is not found in the cache, it means that the sender has created some

[1]Throughout the communication protocol, we assume that the receiver and sender address sets are disjoint. These are also disjoint from the blocks already present in the cache before communication begins.

disturbance, and hence sent a 1. Note that accessing all the addresses from the receiver probe set brings back the cache to the same state as it was before communication of this bit. This is essential for communication of the next bit because occupancy is restored. We know that atmost sndr-probe-blocks number of blocks can be evicted of the receiver, i.e. they can experience a cache miss on lookup. Rest must hit the cache. So the time receiver takes is *sndr-probe-blocks * MISS_LATENCY + (num-blocks - sndr-probe-blocks) * HIT_LATENCY*, which is defined as sndr-wait-cycles.
So for each bit to be communicated, the total time taken is total-cycles.

4. **Error-Detection Step:** There is a possibility of error when a 1 sent by the sender may be read as a 0 by the receiver if its cache occupancy is not 100%. This might happen when the sender ends up evicting only non-receiver blocks in the cache while creating the disturbance, and hence the receiver finds all its addresses hit the cache. Whenever sender evicts a non-receiver block in the cache, a hole is created which even increases the possibility of future errors, because the next miss will surely occupy the hole. So after eci many bits of communication, the sender sends a marker-bit (which is always 1) and receiver performs an error detection and correction step to increase its cache occupancy by filling hole(s) if any.
First, the receiver accesses all its addresses from the receiver probe set. But here it also counts the number of misses num-misses it detects. If num-misses = sndr-probe-blocks, then there cannot be any hole in the cache. This is because sender can evict atmost sndr-probe-blocks number of blocks in the cache and all of these are receiver's, which means there is no hole in the cache. Hence it does not perform any error correction. Otherwise, there is a chance that a hole is present in the cache and it performs the error-correction step. The error-detection step also takes total-cycles time and sender and receiver wait during the same intervals as in the communication step.

5. **Error-Correction Step:** In this step the receiver fills a new address in the cache which is not in the receiver probe set. Now if there was a hole, this new address must go into the hole. Then it flushes this new address. After this, the receiver accesses all its addresses from the receiver probe set. If no miss is detected, this means that there was indeed a hole in the cache so the receiver fills the hole with a new address and hence occupancy increases by 1. This new address is also included in the receiver probe set. **From now on, the receiver will use this new receiver probe set during communication and error-detection-correction steps.** Otherwise, if it finds a miss in these addresses, this means that the new address had replaced one of the receiver's own blocks and hence occupancy has not increased. In this case we stop the current error-correction step going on, because there cannot be any hole present in the cache. Since the protocol is pre-determined between the sender and the receiver, so the sender will wait for the worst-case time bound of error-correction steps in every error-correction. Therefore, the receiver will have to waste the remaining cycles of this current error-correction, as there is no hole to be filled in the cache.
Now, in the worst case, all the sndr-probe-blocks could have created holes that were not of receiver's cache blocks. So we perform the error-correction step

for sndr-probe-blocks iterations. In each iteration, we perform a lookup which will be a miss, then a flush, then we access the receiver probe set out of which atmost 1 can be a miss, all others will be a hit, and then again we may access a block if we have to increase the occupancy. So the total worst-case time for one iteration of error-correction in any eci will be *MISS_LATENCY + HIT_LATENCY + (num-blocks-1) * HIT_LATENCY + MISS_LATENCY + MISS_LATENCY = 3*MISS_LATENCY + num-blocks * HIT_LATENCY*. For sndr-probe-blocks iterations, it will take
sndr-probe-blocks * (3 * MISS_LATENCY + num-blocks * HIT_LATENCY) time for each eci.

3. Implementation

We have created multiple modules, each of which performs a specific function (discussed in section 2.2). All the code is written in Verilog on Xilinx ISE 14.7 interface. We have used Xilinx Spartan 3E Starter Board for synthesis.

3.1 Design

In this section, we will discuss the structural hierarchy of the modules. The *environment* is the top-level module that instantiates all other modules.

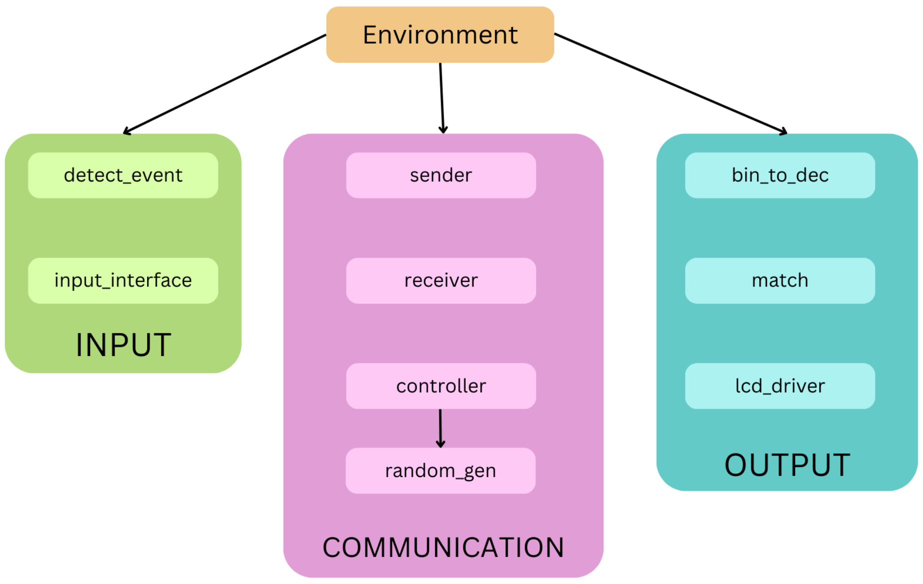


Figure 3.1: Classification of the modules

The **input block** modules take the parameters related to the communication protocol and input string (which is to be sent by the *sender* module) from the user and pass them to the *sender* and *receiver* modules. The **communication block** contains all those modules which are used during the communication. The **output block** modules then check for errors in the string produced by receiver by comparing it with the original string sent by the sender. Also it prints the number of errors and the total number of clock cycles used for communication on the lcd display. Figure 3.1 summarizes the layout of our code.

3.2 Environment

The ***environment*** module acts as an interface of communication among all the other modules. It sends the clock signal *clk* to all the modules and instantiates all of them

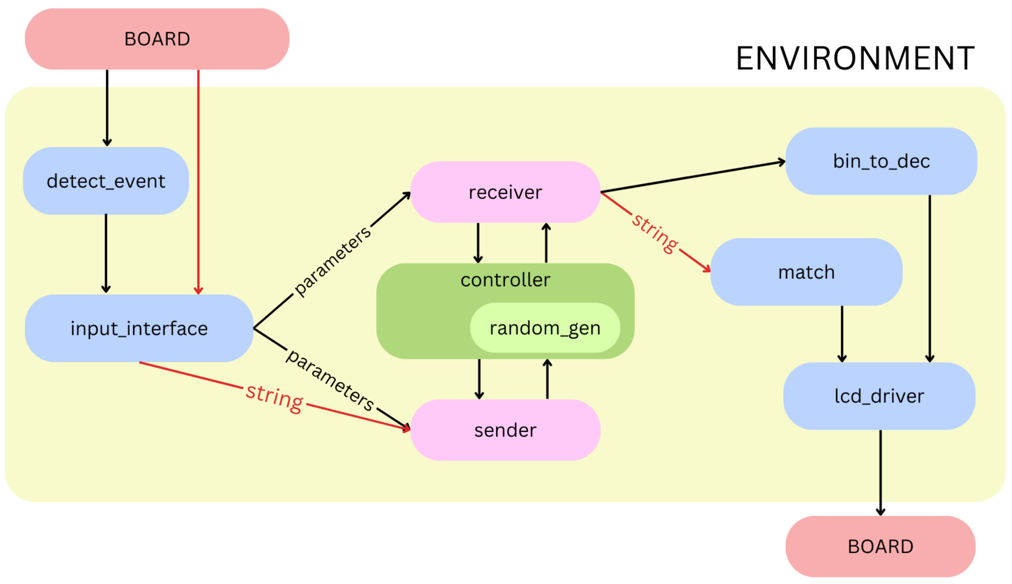


Figure 3.2: Overview of the design

(except the random_gen module which is instantiated by the controller). The occupancy sequence, sndr-probe-blocks and the eci are sent as input from the board. The environment takes these and calculates the crf-total-cycles, crp-total-cycles, sndr-wait-cycles, recv-wait-cycles, total-ed-cycles etc. and sends these to all those modules which require these. Similarly, it collects signals and outputs from *match* and accordingly sends strings to be printed on the LCD.

3.3 Input Block Modules

For giving input using the FPGA board we have used the rotary center, slide switches and push buttons. Each rotation of the rotary generates two signals *rot_a* and *rot_b* which are captured (as input) by the ***detect_event***[1] module to generate a short pulse of *rot_event* signal (as output), which is used by the *input_interface* module. The ***input_interface*** module stores 4 bits for each rotation in an *input_data* array. The input data comprises of the input string which is to be communicated by the sender to the receiver as well as the parameters used in the communication protocol - eci, sndr_probe_blocks and the occupancy sequence. The size of *input_data* (INP_DATA_LEN), is enough to store all the input parameters as well as the input string (which can have atmost STR_LEN bits).[2] To mark the end of the input a push button signal *PB1* is used. This sets the *input_ready* signal to 1. Otherwise, it automatically becomes 1 (without needing to use the push button) once the *input_data* array is completely filled (i.e., max string length is reached). This signal is given as input to the *sender* and *receiver* modules which start working as soon as the *input_ready* signal is received. Also, a variable *eos* is sent as input to the *sender* and *receiver*. It

[1] We would like to thank Prof. Mainak for allowing us to use and modify his code for detect_event.
[2] For detailed description of *input_data*, please refer to section 5.

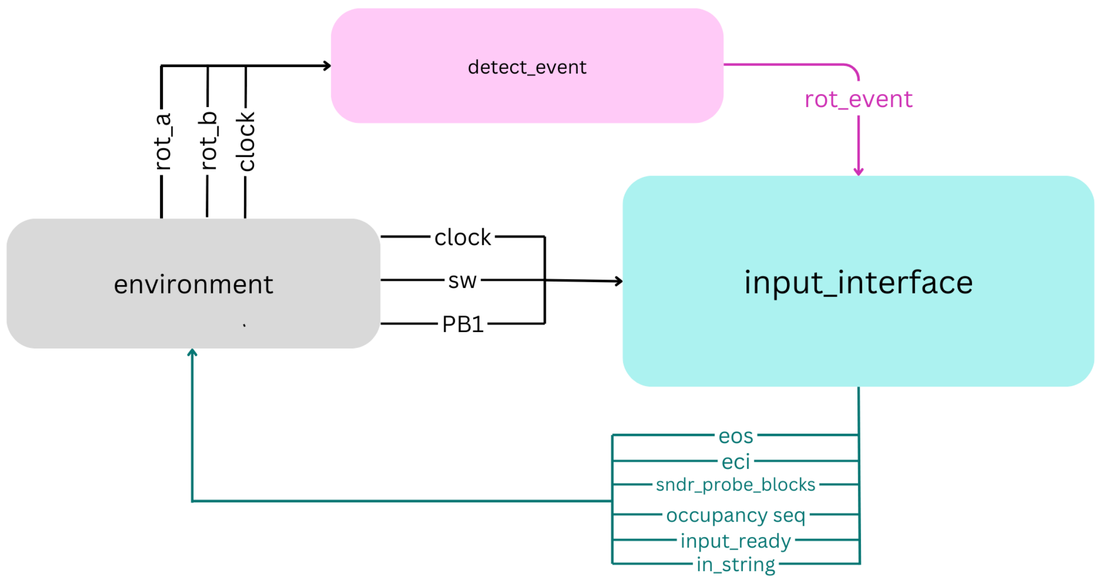


Figure 3.3: Connections for Input Interface

marks the least significant end of the input string in the *input_data*. This tells the *sender* and *receiver* how large is the input string. All the parameters are extracted into separate variables from the *input_data* and sent to the *environment*, where they are used to calculate the number of cycles needed for each step during communication. Only the variables needed are sent to the *sender* and *receiver* respectively. For eg., *sender* does not need the full occupancy sequence, it just needs the total number of CRFill and CRProbe cycles. **Both input string and the parameters are sent to the sender whereas only parameters are sent to the receiver.**

3.4 Communication Block Modules

3.4.1 Controller

The ***controller*** module represents our cache controller. The controller models a blocking cache with random replacement policy which supports two types of operations - **access** and **flush**. These are encoded using the *opcode* variable (0 for access, 1 for flush). For ease of understanding, we can divide the *controller* in 4 parts -

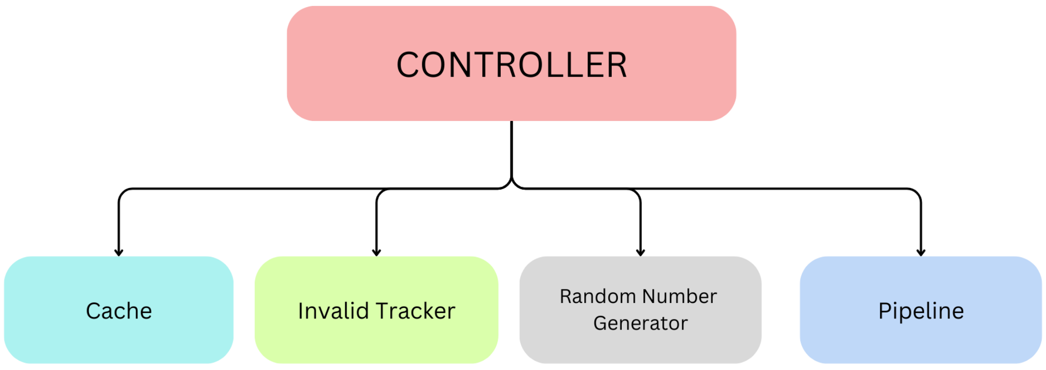


Figure 3.4: Parts of controller

- **Cache** - We have made the *cache* to be an internal variable (a 2D array of size (TAG_LEN+1)* NUM_BLOCKS[3]) in the *controller*. Another way to look at it is to consider the complete module as the cache containing an internal variable to store the data and some logic to access this data. Initially *cache* is in undefined state. We take one "reset" cycle to fill the cache, since we want to start with an occupied cache.

- **Random Number Generator** - The *controller* instantiates the *random_gen* module which is an LFSR, based 16-bit pseudo random number generator [2] [5]. We use only the last LOG_NB bits to get a random number between 0 and NUM_BLOCKS-1 (included).

- **Invalid Tracker** - To keep track of invalid blocks in the cache we maintain a FIFO queue. This is implemented using an array *invalid_index* of size LOG_NB * NUM_BLOCKS, i.e., it can keep atmost NUM_BLOCKS elements each of size LOG_NB bits. We use it to store the indices of the invalid blocks in the cache. We maintain two pointers *invalid_index_RP* and *invalid_index_WP* which point at the front and the back of the queue. These pointers move in a circular fashion, i.e. they wrap around once they reach the end of the array. At any point of time, if an invalid cache block is to be filled, the cache block with index *invalid_index[invalid_index_RP]* will be used. Similarly when a cache block with index *i* becomes invalid, we set *invalid_index[invalid_index_WP]* as *i*. We also maintain a variable *invalid_count* to keep track of the number of invalid blocks in the cache.

- **Pipeline** - For executing the access and flush operations, we have a two-stage blocking pipeline which works as follows -

  - Stage 1 - Search for the requested block in the cache. This can be done in a single cycle by doing all the comparisons in parallel. A variable *hit* of size NUM_BLOCKS bits encodes the hit/miss result for the i^{th} block (i^{th}

[3]Note that for simplicity, we have omitted the data bits in the cache since we do not need them for communication.

bit is 1 if that block matches, 0 if not). If *hit* = 0, it means a cache miss, otherwise it is a cache hit.

– Stage 2 - If opcode for the last cycle was 0 (access) and a miss occurred or, if opcode was 1 (flush) and a hit occurred, the cache is updated accordingly.

Consider the case when two access requests for the same address is sent consecutively, and the first access is a miss. Hit calculation by the second access (stage 1) and the filling of address in the cache by the first access (stage 2) would be done in parallel. This would mean that the second access would again fill the same address in the cache which would be an error. Similar case would occur for two consecutive flushes for the same address and first access is a hit. To avoid such a scenario, the pipeline is stalled. A variable *stall* is set to 1 in stage 2 if the pipeline has to be stalled. Then in the next cycle, *hit* is updated, and then in the next cycle, the stall is reset to 0. At this point we consider the request to be fulfilled. We also respect this behaviour in the *sender/receiver* that requested address changes only when the previous request has been fulfilled.
Also we can note that our MISS_LATENCY = 4 (stage 1, stage 2, stall, reset stall) and HIT_LATENCY = 2 (stage 1, stage 2).

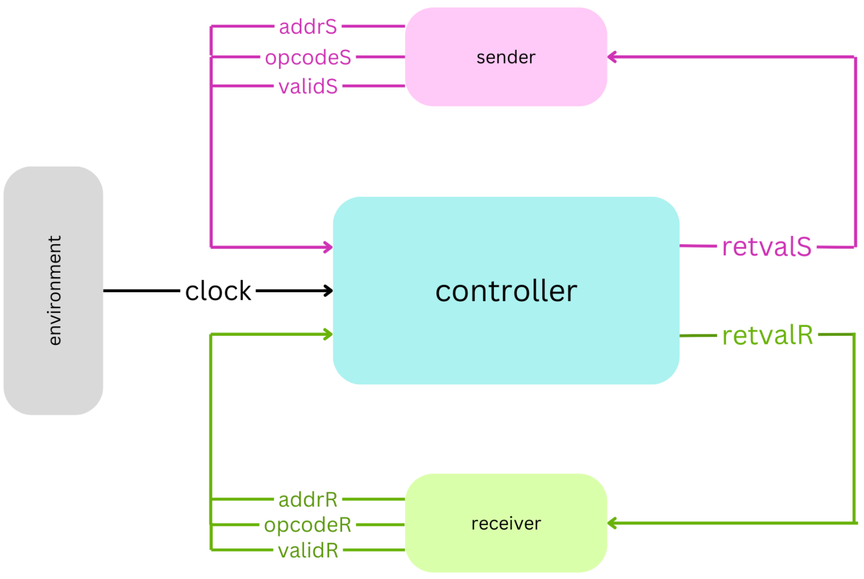


Figure 3.5: Connections for controller

The *controller* has 3 inputs from *sender* and *receiver* each namely - *validS*, *addrS*, *opcodeS* and *validR*, *addrR*, *opcodeR*, respectively. The *validS* variable is set to 1 by the sender when it needs to access/flush a block from the cache, otherwise it remains 0. The *addrS* variable determines the address of the block which is to be accessed/flushed and *opcodeS* determines whether it is an access operation or a flush operation. The variables, *validR*, *addrR* and *opcodeR* have same functions but for the receiever. The *controller* chooses which opcode and address to use according to *validS* and *validR* values. Whichever of the two is 1, the corresponding values will be used. If at any time both *validR* and *validS* are 0, the controller will consider it as a NOP instruction (opcode = 2). Therefore the *sender* and *receiver* need to ensure that both of them never try to access/flush the cache at the same time, i.e., both *validS* and *validR* are not 1 simultaneously. There are two outputs from the controller, one each for sender and receiver, namely, *retvalS* and *retvalR* respectively. Once an access/flush request from the *sender* (or *receiver*) is fulfilled, the *controller* sets *retvalS* (or *retvalR*) as 1. This signals the *sender* (or *receiver*) that a new access/flush request can now be made to the controller. The *sender* (or *receiver*) must ensure that *addrS* (or *addrR*) and *opcodeS* (or *opcodeR*) values do not change until *retvalS* (or *retvalR*) becomes 1. Figure 3.5 shows the connections for the controller.

After *input_ready* becomes 1, both sender and receiver take one "init" cycle to initialise the *index* with the *eos*, i.e., the string is communicated starting from the least significant side.

3.4.2 Sender

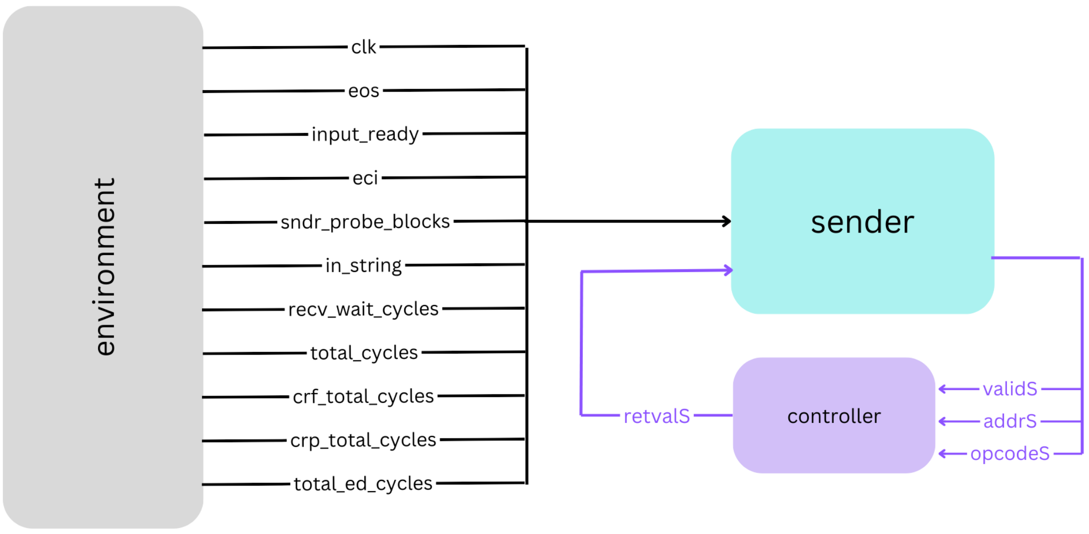


Figure 3.6: Connections for sender

During communication, the ***sender*** works as a **3 state FSM**. Figure 3.7 shows the sender FSM pictorially.
The states of the FSM are determined by a variable *flag* -

- *flag* = 0, CRFill + CRProbe : A counter variable *crf_count* goes from 0 to crf-total-cycles + crp-total-cycles + 1[4]. During this time, *validS* remains 0 throughout this iteration. In the last cycle, *validS* changes to 1 and *flag* changes to 1.

[4]2 extra cycles are needed in receiver

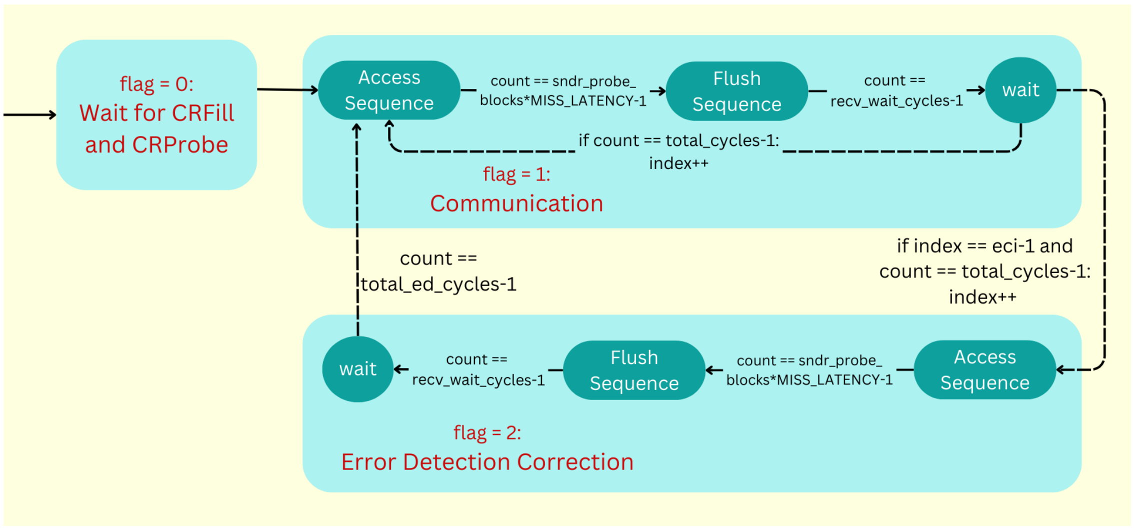

flag = 0:
Wait for CRFill and CRProbe
Access Sequence
count == sndr_probe_ blocks*MISS_LATENCY-1
Flush Sequence
count == recv_wait_cycles-1
wait
if count == total_cycles-1: index++
flag = 1:
Communication
count == total_ed_cycles-1
if index == eci-1 and count == total_cycles-1: index++
wait
count == recv_wait_cycles-1
Flush Sequence
count == sndr_probe_ blocks*MISS_LATENCY-1
Access Sequence
flag = 2:
Error Detection Correction

Figure 3.7: Sender FSM

- *flag* = 1, Communication : For each bit, a counter variable *count* goes from 0 to total-cycles - 1. If the bit to be communicated is 0, *validS* remains 0. Otherwise, if the bit is 1, the sender accesses addresses starting from 0 in increasing order until *count* reaches MISS-LATENCY * sndr-probe-blocks - 1. This is because each access by the sender is always a miss. Then until *count* reaches recv-wait-cycles - 1, sender flushes all its addresses starting from 0. After this *validS* becomes 0. The *opcodeS* and *validS* change according to a combinational logic which depends on *count*. However, logic for *validS* also checks if the bit to be sent is 1 or 0. The *tagS*[5] changes according to *retvalS*. On the above mentioned *count* values – MISS-LATENCY * sndr-probe-blocks - 1, recv-wait-cycles - 1 and total-cycles - 1, *tagS* resets back to 0. Once this iteration is complete, i.e., *count* reaches total-cycles - 1, *index* is incremented and *count* resets to 0.
  A counter called *eci-counter* keeps track of how many bits have been sent since the last error correction step. It is initialised to 0 and once it reaches eci - 1, *flag* changes to 2, and *eci-counter* resets to 0.

- *flag = 2*, Error Detection and Correction : Everything remains same as for *flag* = 1, except for a few changes –
  - A counter variable *ed_count* is used (similar to *count*) which goes from 0 to total-ed-cycles - 1.
  - The bit sent is always 1 in this case, so the combinational logic for *validS* does not check if the bit is 1.
  - The sender waits for a longer time, so *validS* changes to 1 after total-ed-cycles (only if the next bit to be sent is 1).
  - *flag* changes to 1 after error correction step is complete.

[5] *tagS*(or *tagR*) and *addrS* (or *arrR*) can be used interchangeably. Please refer to section 4 for details.

3.4.3 Receiver

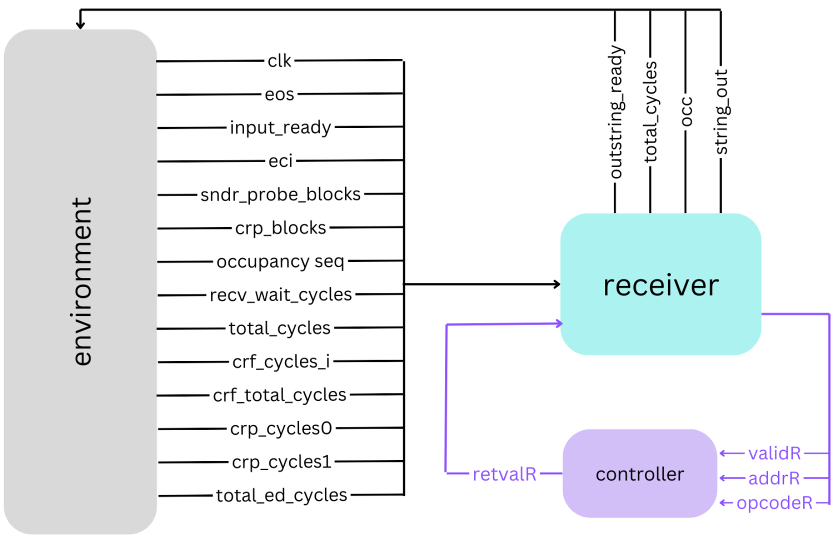


Figure 3.8: Connections for receiver

During communication, the ***receiver*** works as a **4 state FSM**. Figure 3.9 shows the receiver FSM pictorially.

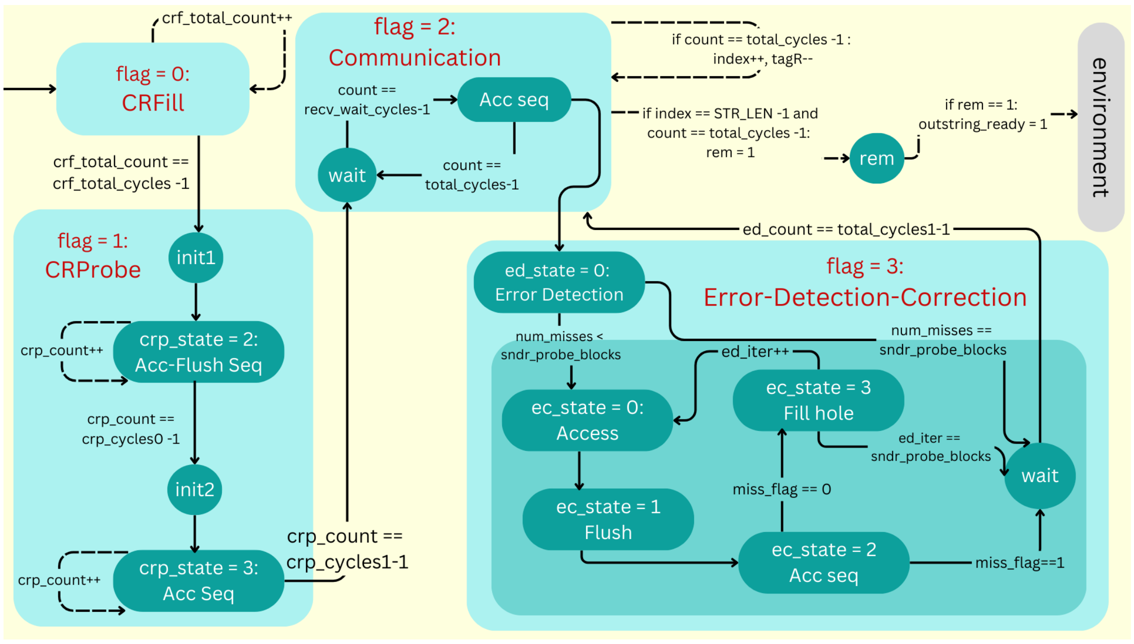


Figure 3.9: Receiver FSM

The states of the FSM are determined by a variable *flag* (similar to sender) -

- *flag* = 0, CRFill : A counter variable *crf_total_count* goes from 0 to crf-total-cycles - 1. This complete step can be subdivided into 3 substeps, one for each term of the occupancy sequence. This is encoded using a variable *crf_state*. Substep i involves accessing crf-blocks$i$ number of blocks exp$i$ number of times. We keep a variable *exp* keep track of the number of iterations, and a counter *crf_count* to keep track of the number of cycles in each iteration. When *crf_state* $= i$, *crf_count* goes from 0 to crf-cycles$i$ - 1 for one iteration, after which *exp* is incremented and *crf_count* resets to 0. During this time, *tagR* goes from start-tag$i$ to end-tag$i$. When *crf_count* resets, *tagR* also resets to start-tag$i$. When *exp* reaches exp$i$ - 1, *crf_state* is changed from i to $i + 1$ and *tagR* is set to *start-tag*$(i + 1)$. Once *crf_total_count* reaches crf-total-cycles - 1, *flag* changes to 1.

- *flag* = 1, CRProbe : This step can be divided into 4 substeps. To keep track of these substeps, we use a variable *crp_state*.
  - *crp_state* = 0, init1 : We take one cycle to set the *tagR* for the CRProbe step and change the *crp_state* to 1.
  - *crp_state* = 1, Access-Flush Sequence : The *tagR* goes from the last accessed address increasing upto MAX_TAG. Once, *tagR* reaches its maximum value, validR is set to 0. For each address first *opcode* is set to 0, then it changes to 1 after retval is received. When retval is received for *opcode* = 1,*tagR* is incremented. We keep a counter *crp_cycle_counter* to count the number of cycles taken for each access. If the retval is received when the counter has not reached MISS_LATENCY-1, it means a cache hit has occurred. We maintain a variable *crp_hits* to count the total number of cache hits in this step. A counter variable *crp_count* keeps track of the number of cycles and goes from 0 to crp-cycles0 - 1, after which *crp_state* changes to 2 and *crp_count* resets to 0. Since, *crp_hits* is equal to the (initial) occupancy of the receiver, we capture this value in *occ_init* variable.
  - *crp_state* = 2, init2 : We take one cycle to set the *tagR* for the next substep. Also we set *validR* to 1, *opcode* to 0, and *crp_state* changes to 3.
  - *crp_state* = 3, Access sequence to fill the holes : *crp_count* variable is reused to count the number of cycles for this step. It goes from 0 to crp-cycles1 - 1, after which *flag* changes to 2. During this time, *tagR* goes from MAX_TAG - *crp_hits* + 1 to MAX_TAG. Note that the number of blocks accessed is same as the number of holes created in *crp_state* = 1.
- *flag* = 2, Communication : The counters work same as the *sender* communication state – *count* keeps track of total number of cycles for each bit, and *eci_counter* keeps track of when to start the error correction step. Thus when *count* reaches total-cycles - 1, *index* is incremented, and when *eci_counter* reaches eci - 1, *flag* is changed to 3. First the receiver waits (*validR* remains 0) until *count* reaches recv-wait-cycles-1. Then *validR* becomes 1 and receiver accesses the blocks starting from *tagR* = MAX_TAG to *tagR* = MAX_TAG - *crp_hits* + 1 in decreasing order. A counter *cycle_counter* counts the number of cycles for each access (similar to the *crp_cycle_counter*) and when its value reaches MISS_LATENCY-1, it means that a cache miss has occurred and therefore the bit at *index* in the

output string is set to 1 and it remains same after that point. A variable *rem* (initialised to 1) is set to 0 when the communication of last bit is completed, i.e., *index* = STR_LEN-1 and *count* = total-cycles - 1. When *rem* becomes 0, on the next cycle *receiver* does not go inside the FSM. Instead it sets a signal *outstring_ready* to 1 signaling the **output block** modules that the output string is ready to be processed and communication has completed.

- *flag* = 3, Error detection and correction : A counter variable *ed_count* goes from 0 to total-ed-cycles - 1, after which *flag* is changed to 2. This step can be subdivided into 2 substeps - error detection and error correction. These are encoded using a variable *ed_state* (0 for detection, 1 for correction). Error detection works similar to normal communication of one bit except in this case it also counts the number of misses observed in a variable *num_misses*. When *ed_count* reaches total-cycles-1, *ed_state* is changed to 1. If at this point *num_misses* < sndr-probe-blocks, we perform the error correction step, otherwise *validR* becomes 0 and nothing is done.
  For the error correction step, we keep an counter *ed_iter* which keeps track of the number of iterations of error correction. There is an upper bound on this which is equal to the sndr-probe-blocks. One iteration can be divided into 4 substeps which are encoded using *ec_state* variable -
  - *ec_state = 0* : Access a new block, set the opcode to 1 for next substep. The new block accessed is *tagR* = MAX_TAG - *crp_hits*, i.e., one less than the smallest block accessed by the receiver.
  - *ec_state = 1* : Flush the block accessed in the last state and set *tagR* to MAX_TAG for the next state.
  - *ec_state = 2* : This stage works similar to the receiver's access sequence in the communication state. Addresses starting from *tagR* = MAX_TAG to *tagR* = MAX_TAG - *crp_hits* + 1 in decreasing order are accessed. We use the *cycle_counter* to count the number of cycles for each access. A variable *miss_flag* is set to 1 if a miss is detected. The condition for error correction is such that if *miss_flag* is 1, we do not enter the error correction step. Also, *validR* becomes 0, *ed_iter* and *ec_state* are reset to 0. In subsequent cycles (which we are supposed to waste), *num_misses* is reset to 0, *tagR* is set to MAX_TAG. However, *miss_flag* is only reset to 0, when *ed_count* reaches total-ed-cycles - 1.
    Otherwise, if no cache miss is detected, *crp_state* changes to 3.
  - *ec_state = 3* : Because a hole is detected in the cache, we access the same block address as in the *ec_state = 0* to fill that hole. Also we have to increase the occupancy by 1. Since *crp_hits* is used everywhere to define the *tagR* limits, to maintain consistency, we increment *crp_hits*. Also, note that the final occupancy and initial occupancy will be different. So, we maintain another variable *occ_final* which captures this final occupancy. Once the retval for the access is received, we increment the *ed_iter*.

3.5 Output Block Modules

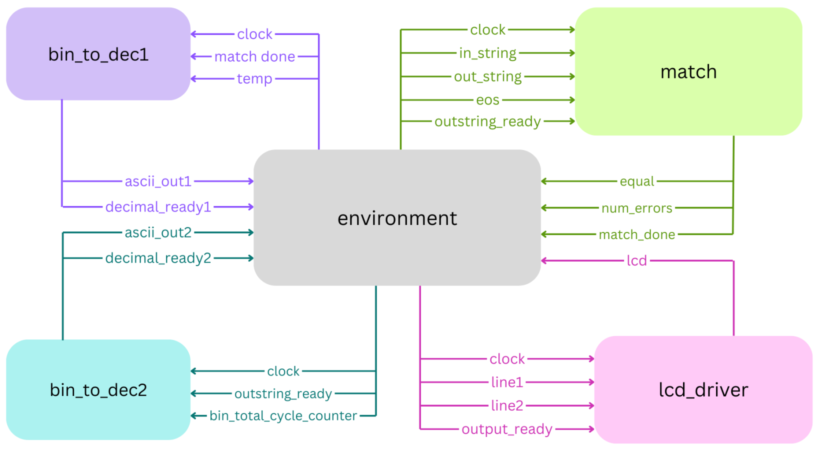


Figure 3.10: Connections for Output Interface

As output of our communication, we use the LCD of the FPGA board to display the **number of errors** in the string generated by the receiver **out_string** when compared with the string that was sent as input. Additionally, we display the **total number of cycles** spent for this communication by the sender and receiver.
The ***match*** module takes inputs the *out_string* created by the receiver, the *in_string* sent by the sender, a signal *outstring_ready* which is 1 when communication is finished, and the eos which marks the least significant bit of the strings from where matching has to be done. It returns *num_errors* equal to the number of bits that mismatch between the strings. If there is a perfect match, then output *equal* is set to 1, otherwise it is 0. Additionally, it sets *match_done* to be 1 when it has finished its processing.
The ***bin_to_dec*** module takes input a binary number *binary_in* and returns the ascii-coded decimal of that number. We use the **Double-Dabble algorithm** [4] to convert a binary number to its binary-coded-decimal. We give a 32-bit binary number *binary_in* as input and *binary_ready* to signify that the input is ready. Then we create a 16-digit decimal number *ascii_out* in which each digit is encoded as 8-bit-ascii representation of that number and when this is done, it sets *decimal_ready* to be 1. We instantiate it twice, first to convert *num_errors* calculated by *match* module into a 4-digit decimal number, and second to convert the TOTAL CYCLES spent for communication into a 16-digit decimal number.
The ***lcd_driver***[6] takes in two 128-bit strings, each character encoded as 8-bit-ascii number, and prints it on the LCD of the FPGA board. In the first line, we display "PASS, Error=0000" if there was no error reported by ***match*** module. Otherwise we display "FAIL, Error=dddd", where *dddd* is the number of errors (in decimal) found by module *match*. The second line prints the TOTAL CYCLES which shows the number of cycles (in 16-digit decimal) spent during the entire communication. Thus, we need to instantiate *bin_to_dec* twice, for number of errors and for TOTAL CYCLES.

[6]We would like to thank Prof. Mainak for allowing us to use and modify his code for lcd_driver.

4. Assumptions

We have made certain assumptions while designing CovertCraft. Some of these are due to limitation of resources in the Spartan 3E FPGA board while others are certain design choices we have made for simplicity.

4.1 Assumptions for Simplicity

- **The *sender* and *receiver* are FSMs in the hardware** and directly send requests to the cache controller. However, in a real system, they are software constructs and their codes would communicate with the cache via a processor.
- We have assumed a **single cache which is shared between the *sender* and *receiver***. Also, in case of miss, there is **no actual DRAM** from which data is to be fetched. In a typical multi-core systems, multi level caches are present where only the Last Level Cache (LLC) is shared across the cores. Typically sender and receiver are scheduled on different cores.
- **Miss and hit latencies are fixed.** This is because our design is deterministic. Since the cache is blocking and data is not fetched from the DRAM, the number of cycles in case of a miss or a hit is fixed. In a non blocking cache with DRAM support, these latencies can be variable. Also, because of fetch from DRAM, miss latency would be much larger than hit latency. Therefore, having a non blocking cache improves efficiency since in case of a miss, pipeline need not be stalled.

4.2 Constraints on Variables

- Cache size (NUM_BLOCKS) must be a power of 2. The random number generator generates a 16 bit number. We need a number in the range 0 to NUM_BLOCKS-1. This can be done efficiently by taking the last $log_2(NUM\_BLOCKS)$ bits if NUM_BLOCKS is a power of 2. Also, this assumption is true for most cache designs.
- Input bit-length must be a multiple of 4. This is because we use slide switches to give 4 bits at a time.
- Largest error which can be printed is 9999, since we have dedicated only 4 digits for the error. Thus maximum string length can not be more than 9999.
- The occupancy sequence cannot have more that 3 terms. For a small cache large occupancy sequence is not needed. Also increasing the number of terms increases the size of the receiver FSM which would then need more circuitry in the hardware during synthesis.
- There is an upper limit on the value of input parameters. This is because we have dedicated a fixed number of bits in the input for each parameter.

5. Input Protocol

We use the rotary center ROT-A & ROT-B, push button PB1 and the slide switches SW3, SW2, SW1 & SW0 to give inputs from the board. We set the slide switches as per the input to be sent and then give a rotation through the rotary center. This takes in 4 bits at a time.

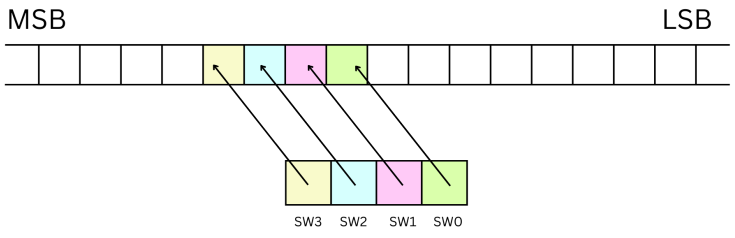


Figure 5.1: Method of providing input from the board

The protocol to send a 4k-bit long input {4k-1,4k-2,.....,1,0}, where the 4k-1th bit is the MSB and 0th bit is the LSB, is as follows. We send the input starting from the most significant bit side to the least significant side. In i^{th} rotation (assume first rotation has $i = 0$), we will send bits at positions {4(k-i)-1, 4(k-i)-2, 4(k-i)-3, 4(k-i)-4}. For this, we will set $SWj = (4(k-i)+j-4)$. Figure 5.1 shows a pictorial view of this protocol.

Structure of *input_data*

The first 12 bits are the eci. Then the next 4 bits sent are the sndr-probe-blocks. Then we send the occupancy sequence. We have assumed it to have atmost 3 terms. So the next 8 bits are crf-blocks0, followed by its exp0 in the next 4 bits. So the next 8 bits are crf-blocks0, followed by its exp0 in the next 4 bits. Similarly the remaining two terms are sent. After this we send the in_string. We have assumed it to be atmost 1024 bits long. So we can terminate the input at any point either by pressing the push button, or it terminates by itself if 1024 length of string has been sent. Figure 5.2 demonstrates the input_data pictorially.

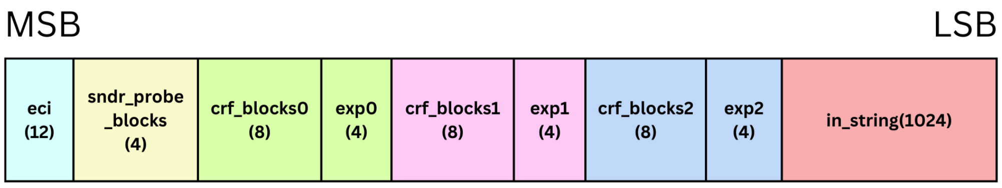


Figure 5.2: Structure of *input_data*

6. Evaluation Metrics

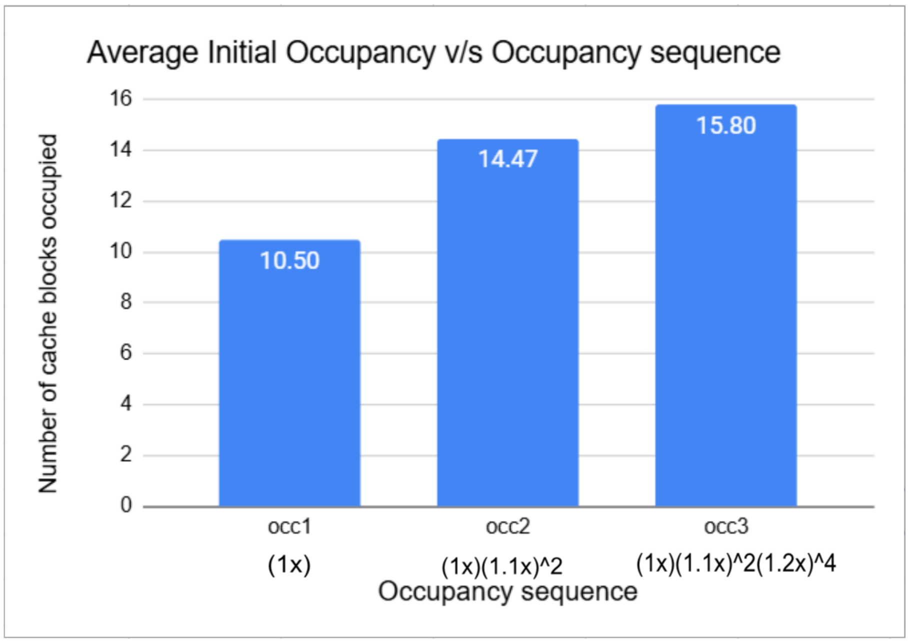
We fixed the string length to 1024 bits and varied the parameters: occupancy sequence, the sndr-probe-blocks and the eci to maximize bandwidth and minimize error rate. We used the average bandwidth and average error of 5 readings for each set of parameters to formulate results. We also captured the **initial occupancy** achieved by the receiver just after the CRProbe step, before any eci. This helped us improve the occupancy sequence that would have high chances of bringing 100% cache occupancy right after CRFill.

6.1 Observations from Experimentation

- **occupancy sequence**: Varying the occupancy sequence does not affect the bandwidth much, as CRFill step happens only once. But it has great effect on increasing the initial occupancy of cache by the receiver and hence reducing error rate. We tried three different occupancy sequences: $\{(1x), (1x)(1.1x)^2, (1x)(1.1x)^2(1.2x)^4\}$ and following are the results:

Occupancy Sequence	Average Initial Occupancy
$(1x)$	10.5
$(1x)(1.1x)^2$	14.47
$(1x)(1.1x)^2(1.2x)^4$	15.8

Table 6.1: Varying the occupancy sequence



The sequence $\mathbf{(1x)(1.1x)^2(1.2x)^4}$ ensures almost 100% cache occupancy during the CRFill step with very high probability. For a larger size cache, a bigger occupancy sequence may be needed because for such a cache the probability of 100% occupancy is less.

- **sndr-probe-blocks**(SPB): For having minimum errors during the communication, we want every disturbance by the sender to be recorded by the receiver. So we would like to have sndr-probe-blocks as large as possible. But this also impacts the bandwidth adversely, because that many more cycles would be required for every bit. And once the receiver's cache occupancy reaches 100%, flushing any 1 block in the cache by the sender would communicate the bit correctly. We tried two different values for sndr-probe-blocks: 2 and 4, since ours is a small 16-entry cache. We found out that if initial occupancy was not 100%, SPB = 4 gave lesser average error than SPB = 2 but it had a lower bandwidth. However if the initial occupancy reached 100%, then even just 2 blocks would give zero error. Refer to graph Figure 6.2.

- **eci**: A smaller eci ensures that if any hole(s) are created, they are filled sooner. Also occupancy increases in such cases after the error correction step. This reduces the errors drastically (refer to Figure 6.2). However, error correction takes a large number of cycles for each marker bit and hence more frequent error correction (i.e., smaller eci) would result in a lower bandwidth (refer to Figure 6.1). Therefore, if we ensure a high initial occupancy (i.e., close to 100%), having a larger eci is more efficient. We tested for three different eci values: 8, 32, and 128.

Following are the results of varying the sndr-probe-blocks and eci:

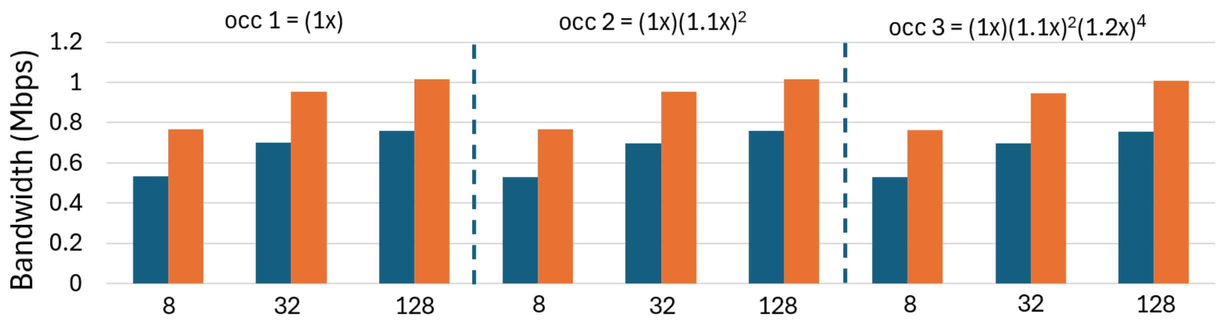
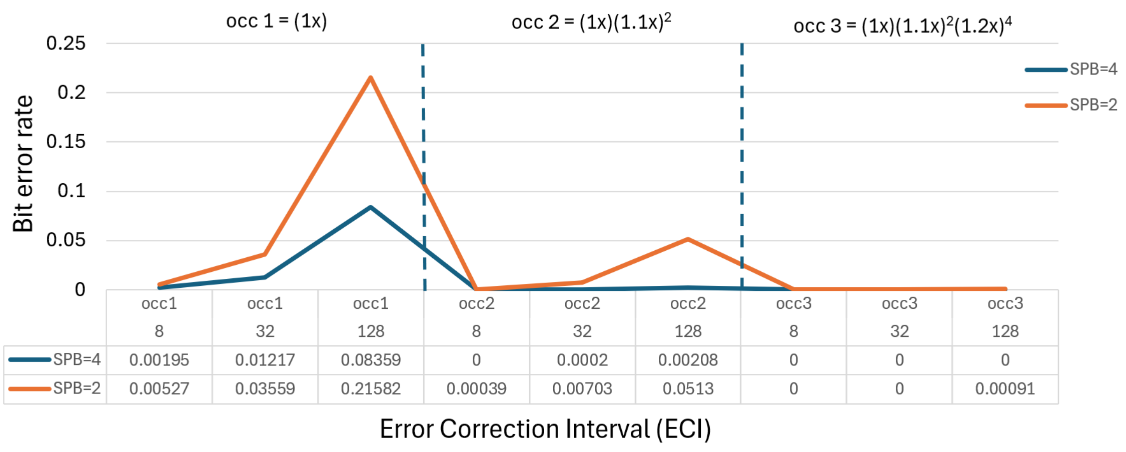


Figure 6.1: Variation of Bandwidth with eci, SPB and occ sequence



	occ1 8	occ1 32	occ1 128	occ2 8	occ2 32	occ2 128	occ3 8	occ3 32	occ3 128
SPB=4	0.00195	0.01217	0.08359	0	0.0002	0.00208	0	0	0
SPB=2	0.00527	0.03559	0.21582	0.00039	0.00703	0.0513	0	0	0.00091

Figure 6.2: Variation of BER with eci, SPB and occ sequence

Occ Seq	ECI	SPB[1]	Avg Init Occ	Bit Error Rate $\times 10^3$	Bandwidth(Mbps)
(1x)	8	2	11.27	5.27	0.768
(1x)	8	4	10.8	1.95	0.532
(1x)	32	2	10.2	36.59	0.954
(1x)	32	4	11	12.17	0.699
(1x)	128	2	9.73	215.82	1.017
(1x)	128	4	10	83.59	0.759
$(1x)(1.1x)^2$	8	2	14.33	0.39	0.766
$(1x)(1.1x)^2$	8	4	14.8	0	0.531
$(1x)(1.1x)^2$	32	2	14.73	7.03	0.952
$(1x)(1.1x)^2$	32	4	14.47	0.2	0.698
$(1x)(1.1x)^2$	128	2	13.93	51.3	1.014
$(1x)(1.1x)^2$	128	4	14.53	2.08	0.757
$(1x)(1.1x)^2(1.2x)^4$	8	2	15.87	0	0.763
$(1x)(1.1x)^2(1.2x)^4$	8	4	15.73	0	0.529
$(1x)(1.1x)^2(1.2x)^4$	32	2	15.87	0	0.947
$(1x)(1.1x)^2(1.2x)^4$	32	4	15.8	0	0.695
$(1x)(1.1x)^2(1.2x)^4$	128	2	15.8	0.91	1.008
$(1x)(1.1x)^2(1.2x)^4$	128	4	15.73	0	0.754

Table 6.2: Varying various parameters

- We found out that the occupancy sequence $\mathbf{(1x)(1.1x)^2(1.2x)^4}$ gave us almost 100% cache occupancy each time.[2].
- The values {eci = **128**, sndr-probe-blocks = **2**} was optimal among our tested values for 1024-bit string, in minimising bit-error-rate and maximising bandwidth.

[1]SPB refers to sndr-probe-blocks
[2]For a small cache, achieving 100% occupancy is easier

6.2 Bandwidth Calculation

Since the miss and hit latencies are deterministic, end-to-end communication bandwidth is also deterministic. The bandwidth calculation can be done as follows[3] -

- **CRFill cyles** (crf-total-cycles) = $\sum_i$ (crf-blocks$i$ * exp$i$ * ML)
- **CRProbe blocks** (crp-blocks)= max$_i${ crf-blocks$i$}
- **CPRobe cycles** (crp-total-cycles) = (ML+HL)*crp-blocks + NUM*ML + 2
- **Communication cycles per bit** (total-cycles) = SPB * (ML + HL) + SPB * ML + (NUM_BLOCKS - SPB) * HL = 2*ML*SPB + HL*NUM
- **Total Cycles for Communication** = LEN*total-cycles
- **Error Detection Correction cycles per marker bit** (total-ed-cycles) = total-cycles + SPB * (ML + HL + HL(NUM-1) + ML + ML) = 2*ML*SPB+HL*NUM + SPB*(3*ML+NUM*HL) = SPB*(5*ML+NUM*HL) + NUM*HL
- **Number of marker bits sent** (mbits) = max $\left(0, \left\lfloor \frac{\text{LEN}-1}{\text{ECI}} \right\rfloor\right)$
- **Total Cycles for error correction detection** = mbits * total-ed-cycles
- **Total end-to-end cycles** (TC) = crf-total-cycles + crp-total-cycles + LEN*total-cycles + mbits * total-ed-cycles
- **Clock cycle time** = T
- **Total time taken** (for LEN bits) = $TC * T$
- **Bandwidth** = bits sent per second = $LEN/TC * T$

Taking SPB = 2, NUM = 16, ML = 4, HL = 2, ECI = 128, Occupancy sequence = $(1x)(1.1x)^2(1.2x)^4$, i.e., **crf-blocks0** = 16, **crf-blocks1** = 18, **crf-blocks2** = 19, **exp0** = 1, **exp1** = 2, **exp2** = 4, LEN = 1024 and T[4] = 20 ns = 2×10^{-8} -

- **CRFill cycles** = $(16 * 1 + 18 * 2 + 19 * 4) * 4 = 512$
- **CRProbe blocks** = $max(16, 18, 19) = 19$
- **CRProbe cycles** = $(4 + 2) * 19 + 16 * 4 + 2 = 180$
- **Communication cycles per bit** = $2 * 4 * 2 + 2 * 16 = 48$
- **Total Cycles for Communication** = $1024 * 48 = 49152$
- **Error Detection Correction cycles per marker bit** = $2*(5*4+16*2)+16*2 = 136$

[3]SPB refers to sndr-probe-blocks, NUM is the number of blocks in the cache, ML is the miss latency, HL is the hit latency, LEN is the length of the string, ECI refers to the eci

[4]Spartan 3E starter board has a clock with 50 MHz frequency.

- **Number of marker bits sent** $= \lfloor \frac{1024-1}{128} \rfloor) = 7$
- **Total Cycles for error correction detection** $= 7 * 136 = 952$
- **Total end-to-end cycles** $= 512 + 180 + 49152 + 952 = 50796$
- **Clock cycle time** $= 2 * 10^{-8}$
- **Total time taken** (for LEN bits) $= 50796 * 2 * 10^{-8} = 1.01592 * 10^{-3}$
- **Bandwidth** $= 1024/(1.01592 * 10^{-3}) = 1007953\ bits/second = 1.008\ Mbps$

Glossary

crf-blocks*i* Number of blocks accessed for the i^{th} term in the occupancy sequence.

crf-cycles*i* Total number of cycles required for one iteration of i^{th} term in the occupancy sequence.

crf-total-cycles Total number of cycles required in the CRFill step in the worst case $= \sum_i$ (crf_blocksi * expi * MISS_LATENCY).

crp-blocks Total number of blocks accessed by the receiver during the Access-Flush stage of CRProbe step = Total number of distinct blocks accessed by the receiver during the CRFill step = $max_i\{$ crf_blocks*i*$\}$.

crp-cycles0 Total number of cycles required (in the worst case) for the Access-Flush stage of the CRProbe step =(MISS_LATENCY + HIT_LATENCY) * crp_blocks.

crp-cycles1 Total number of cycles required (in the worst case) for finally filling the holes created after the Access-Flush stage in the CRProbe step = NUM_BLOCKS * MISS_LATENCY.

crp-total-cycles Total number of cycles required in the CRProbe step in the worst case = crp_cycles0 + crp_cycles1 + 2.

disturbance set Set of addresses accessed by the sender during communication. In our case, disturbance set = {0,1,2,...,sndr_probe_blocks-1}.

eci Error Correction Interval. Number of bits to be sent between to error correction steps.

end-tag*i* Last tag accessed by the receiver for the i^{th} term of the occupancy sequence. end_tagi = start_tagi - crf_blocks_i + 1.

eos The least significant bit in the string to be communicated. The string stretches from eos to STR_LEN-1.

exp*i* Number of times *crf-blocksi* are accessed during CRFill.

hole An invalid block in the cache.

marker-bit A 1 bit to be sent by the sender apart from the communication at each error-correction-interval for error-detection-correction step.

num-blocks The total number of blocks in the cache.

num-marker-bits The number of marker bits sent during the entire communication. It is max(0,(STR_LEN-1)/eci).

num-misses The number of misses encountered by the receiver from its previous occupancy during the error-detection-correction step.

occupancy sequence Set of addresses which the receiver accesses to fill them in the cache. It can be expressed as $(A_1x)^{B_1}(A_2x)^{B_2}...$, which means that $A_i * num\_blocks$ addresses are accessed B_i times. For each term, we access addresses starting from MAX_TAG and go upto MAX_TAG -$A_i * num\_blocks + 1$.

receiver probe set Set of addresses accessed by the receiver during communication and error-detection-correction steps. In our case, if occupancy is x than receiver probe set = *{MAX TAG, MAX TAG-1,...., MAX TAG-x+1}*.

recv-wait-cycles Total number of cycles per bit during communication (in the worst case) for which receiver waits and sender works = (MISS_LATENCY + HIT_LATENCY) * sndr_probe_blocks.

sndr-probe-blocks Number of blocks accessed by the sender during communication = size of disturbance set.

sndr-wait-cycles Total number of cycles per bit during communication (in the worst case) for which sender waits and receiver works = sndr_probe_blocks * MISS_LATENCY + (NUM_BLOCKS - sndr_probe_blocks) * HIT_LATENCY. This is because in the worst case sender can create atmost sndr_probe_blocks number of holes.

start-tag*i* First tag accessed by the receiver for the i^{th} term of the occupancy sequence. start_tag0 = MAX_TAG, start_tagi = end_tag(i-1) - 1 (for $i \geq 1$).

TOTAL CYCLES The total number of cycles spent during communication of the entire string. It is equal to crf-total-cycles + crp-total-cycles + total-cycles * STR_LEN + total-ed-cycles * num-marker-bits.

total-cycles Total number of cycles per bit required during communication in the worst case =sndr_wait_cycles + recv_wait_cycles.

total-ed-cycles Total number of cycles required during the error detection and correction steps combined.

Bibliography

[1] Fangfei Liu et al. “Last-Level Cache Side-Channel Attacks are Practical”. In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 605–622. doi: 10.1109/SP.2015.43.

[2] Stack Overflow contributors. *Random number generation on Spartan-3E*. [Online; accessed 9-November-2023]. 2009. url: https://stackoverflow.com/questions/757151/random-number-generation-on-spartan-3e.

[3] Zhenghong Wang and Ruby B. Lee. “Covert and Side Channels Due to Processor Architecture”. In: *2006 22nd Annual Computer Security Applications Conference (ACSAC’06)*. 2006, pp. 473–482. doi: 10.1109/ACSAC.2006.20.

[4] Wikipedia contributors. *Double dabble*. [Online; accessed 9-November-2023]. 2023. url: https://en.wikipedia.org/wiki/Double_dabble.

[5] Wikipedia contributors. *Linear-Feedback Shift Register*. [Online; accessed 9-November-2023]. 2023. url: https://en.wikipedia.org/wiki/Linear-feedback_shift_register.