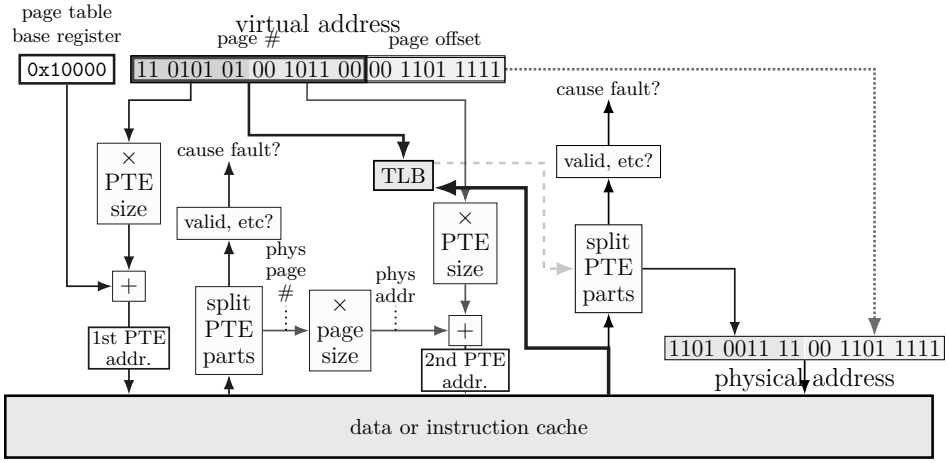
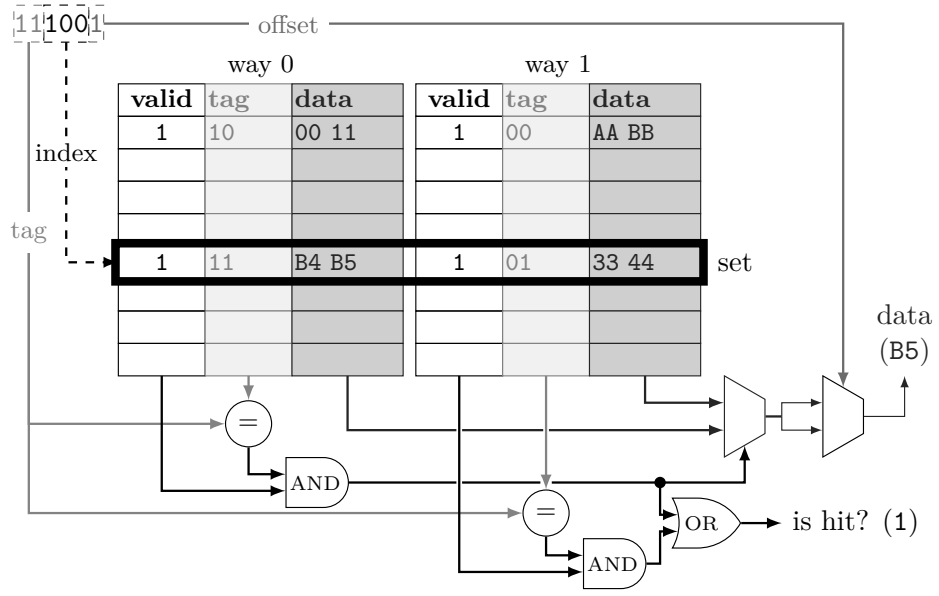
1 page table lookup



2 cache organization



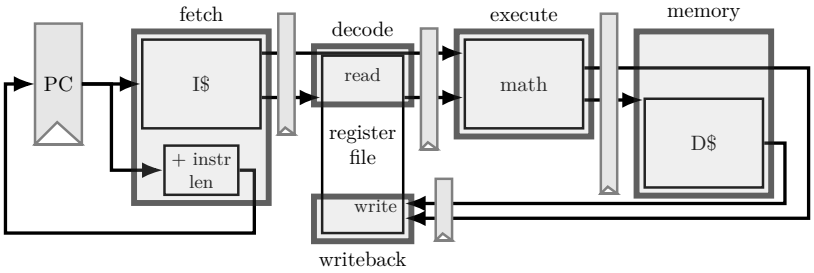
3 networking layers

application	HTTP, SSH, SMTP, …	URLs, …	…	application-defined meanings
transport	TCP, UDP, …	port numbers, …	segments, datagrams	reach correct program, reliablity/streams
network	IPv4, IPv6, …	IP addresses, …	packets	reach correct machine (across networks)
link	Ethernet, Wi-Fi, …	MAC addresses, …	frames	coordinate shared wire/radio
physical	…	…	…	encode bits for wire/radio

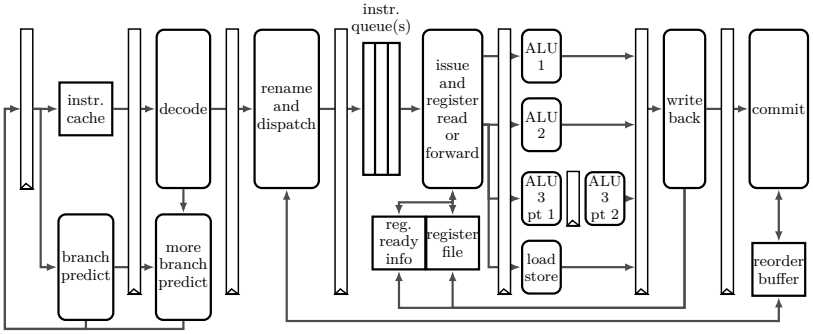
4 Spectre pattern

```
if (x < array1_size)    // bounds check, skipped by branch prediction
    y = array2[array1[x] * 4096]; // access cache set dependent on array1[x]
```

5 pipelined processor



6 OOO processor



7 selected POSIX functions

- given `lock` is a `pthread_mutex_t` and `cv` is `pthread_cond_t`
  - mutex lock/unlock: `pthread_mutex_lock(&lock)`; `pthread_mutex_unlock(&lock)`;
  - `pthread_cond_wait(&cv, &lock)` — unlock lock + wait on cv's queue; when woken up, relock lock and return; can be woken up early by 'spurious wakeup'
  - `pthread_cond_signal(&cv)` — wake up one waiting thread from cv's queue
  - `pthread_cond_broadcast(&cv)` — wake up all waiting threads from cv's queue
  - `pthread_create(&t, NULL, start_function, a)` — create thread (ID stored in `t`) that will run `start_function` with the argument `argument`
  - `pthread_join(t, &ret)` — wait for thread `t` to finish, collect its return value in `ret`
  - create new process copying current: `fork()` — return new pid in parent (old), 0 in child (new)
  - `open(filename, O_RDONLY)` returns a file descriptor that can be used to read from the file with named `filename`
  - `read(fd, buffer, size)` read up to `size` bytes from `buffer` to the file descriptor `fd`, return total bytes read or 0 on end-of-file
  - `lseek(fd, pos, SEEK_SET)` sets the file pointer for the file descriptor `fd` to `pos` bytes from the beginning of the file
  - `waitpid(pid, 0, NULL)` wait for the child process with ID `pid` to terminate

8 assembly

- `OPq %r8, %r9`: perform OP (example: add) on `%r8` and `%r9`, put a resulting number (if any) in `%r9`
- `movq X, Y`: move 64-bit value from X to Y
- `%r8`, `%rax`, etc. — 64-bit register
- `(%r8)` — the value in memory at an address equal to the value of `%r8`

variant
X

Computing ID: _____

Name: _____

Write your name and computing ID above. Write your computing ID at the top of each page in case pages get separated. Sign the honor pledge below.

Generally, we will not answer questions about the exam during the exam time. If you think a question is unclear and requires additional information to answer, please explain how in your answer. For multiple choice questions, write a * next to the relevant option(s) along with your explanation.

On my honor as a student I have neither given nor received aid on this exam.

_____

1. Suppose a system uses:
   - 4096-byte (2^{12} byte) pages
   - three-level page tables
   - 1024-entry page tables for each level, with 4-byte page table entries (so each page table takes up 4096 bytes)
   - 42-bit virtual addresses
   - a 32-entry, 4-way data TLB and a 32-entry, 4-way instruction TLB

(a) (7 points) Suppose a program has a loop like:

```
for (int i = 0; i < 100; i += 1) {
    for (int j = 0; j < 4; j += 1) {
        ptr1[j * N * 4096] += ptr2[j * N * 4096];
    }
}
```

where `ptr1` and `ptr2` are pointers to unsigned char (which are one byte in size).

We discover that on the system described above that each read from `ptr1[j*N*4096]` and `ptr2[j*N*4096]` in the loop results in a TLB miss, even when those accesses to ptr1 and ptr2 are the only thing using the data TLB and no exceptions occur while the above code runs. Which of the following is true about how this scenario could have been set up? **Select all that apply.**

- ☐ `N` could be 32
- ☐ `N` could be a value less than 32
- ☐ `ptr1[0]` and `ptr2[0]` could map to the same physical page
- ☐ `ptr1` and `ptr2` could map to the same virtual page
- ☐ the addresses ptr1[0] and ptr2[0] could differ in the index used for the first-level page table entry lookup
- ☐ the physical pages allocated for ptr1[0] and ptr1[N] could differ in the least significant bit of their physical page number
- ☐ the virtual page numbers for ptr1[0] and ptr2[0] could differ in their least significant bits

(b) (10 points) Suppose a program can access the following (inclusive) virtual address ranges:

- `0x0 000 010 000`—`0x0 000 02F FFF`
- `0x1 000 000 000`—`0x1 000 FFF FFF`
- `0x3 FFF FF0 000`—`0x3 FFF FFF FFF`

The actual memory the program can access totals to 4144 (32+4096+16) pages. In order for the program to access all this space, how many page tables at each level must be allocated? (Assume all accessible memory is mapped in the page tables, so no exceptions will occur when the program tries to access any of the addresses above.)

first-level tables	
second-level tables	
third-level tables	

2. Consider a protocol for transmitting data from a sender to receiver over a network which provides the 'mailbox model' of functionality. The protocol divides the data to be send into 1000 byte chunks and sends each chunk as follows:

   1. the sender sends a packet containing 1000 bytes of data and a sequence number
   2. when the receiver receives a data packet, it sends back a packet acknowledging that with the sequence number of the data packet it just received
   3. if the sender does not receive an acknowledgment within 100 milliseconds, it restarts from step 1

   After the steps above are complete, the sender immediately starts sending the next chunk.

   Normally, the network takes 20 milliseconds to send a packet from the sender to receiver and vice-versa.

(a) (3 points) If the data to be sent totals 100 000 bytes and the network behaves normally (without losing packets), how long will it take from when the sender starts sending data until when the receiver receives all data? You may leave your answer as an unsimplified arithmetic expression.

(b) (5 points) Suppose the data to be sent totals 100 000 bytes, but the every third packet starting with the first that the receiver sends is lost (so the first packet, fourth, seventh, and so on). How much **longer** will it take from when the sender starts sending data until the receiver receives all data? You may leave your answer as an unsimplified arithmetic expression

3. Consider the following C code:

```
struct Socket {
    unsigned char buffer[4096];
    int bufferPos;
    int bufferReady;
    ...
};
struct Socket sockets[4096];

unsigned char ReadByte(unsigned socketId) {
    if (socketId < 4096) {
        struct Socket *sock = &sockets[socketId];
        if (sock->bufferReady > 0) {
            unsigned char result = sock->buffer[sock->bufferPos];
            sock->bufferReady = sock->bufferReady - 1;
            sock->bufferPos = sock->bufferPos + 1;
            return result;
        }
    }
    return 0;
}
```

For the following questions, consider performing a Spectre-style attack on the above function where the attacker has control of `socketId` and can use side-channels to obtain information about the data cache accesses the function performs. As a result of the attack, the attacker expects to gain information about the value of a memory location they should not have access to (outside the bounds of any of the structs the ReadByte function is intended to access).

(a) (5 points) As part of the Spectre attack pattern, an attacker can get information about what set in the cache was evicted by an access. Data cache accesses to which of the following would provide the best information to the attacker in the above function?

- ◯ `socketId` if socketId is copied to the stack to be used a local variable
- ◯ `sock` if the pointer sock is stored on the stack
- ◯ `sock->buffer[sock->bufferPos]`
- ◯ `sock->bufferReady` and/or `sock->bufferPos`
- ◯ `result` if the char result is stored on the stack
- ◯ the return address of `ReadByte()`

(b) (5 points) Which of the following changes would mitigate the Spectre attack described in the previous question? **Select all that apply.**

- ☐ flushing the data cache before the function runs
- ☐ converting `sockets[socketId]` to `networkInterfaces[socketid % 4096]` and making similar modifications to all other array accesses
- ☐ converting `sockets[ifaceId]` to `socketId > 4096 ? NULL : sockets[socketId]` and making similar modifications to all other array accesses
- ☐ making it so the attacker's code runs with a different cache than the code being attacked (such as by having separate caches for kernel and user mode)
- ☐ making it so the attacker's code runs with different page tables than the code being attacked (such as by having separate page tables for kernel and user mode)

(c) As part of performing the attack, the attacker may want to 'train' the branch predictor to predict that certain conditionals will or will not be true. For each of the following conditionals, identify if the attacker would want it to be predicted as true, false, or if it does not matter to the attacker.

i. (2 points) `socketId < 4096`
◯ true ◯ false ◯ does not matter

ii. (2 points) `sock->bufferReady > 0`
◯ true ◯ false ◯ does not matter

4. A programmer attempts to use multiple threads to compute the sum of values in a binary file. They write two versions, one using fork() and one using pthread_create(): Both versions use these common utility functions and corresponding global variables:

common code

```
#define ARRAY_SZ 1048576
#define FILENAME "array.dat"

int fd = -1, sum = 0;
void SumPart(int part) {
  int arr[ARRAY_SZ/2];
  if (fd == -1)
    fd = open(FILENAME, O_RDONLY);
  // set file pointer 'start' bytes into the file
  lseek(fd, part * ARRAY_SZ/2 * sizeof(int), SEEK_SET);
  read(fd, arr, ARRAY_SZ/2 * sizeof(int));
  for (int i = 0; i < ARRAY_SZ/2; i += 1)
    sum += arr[i];
}

void FinishSum() {
  if (fd != -1)
    close(fd);
  printf("the sum is %d\n", sum);
}
```

Then, the programmer constructed two versions, both of which have bugs:

fork version

```
int main() {
  pid_t pid;
  pid = fork();
  if (pid == 0) {
    SumPart(1);
    exit(0);
  } else {
    SumPart(0);
    waitpid(pid, &status, 0);
  }
  FinishSum();
  return 0;
}
```

pthread version

```
void *SumFront(void *ignored) {
    SumPart(0); return NULL;
}
void *SumBack(void *ignored) {
    SumPart(1); return NULL;
}
int main() {
    pthread_t threads[2];
    pthread_create(&threads[0], NULL,
        SumFront, NULL);
    pthread_create(&threads[1], NULL,
        SumBack, NULL);
    pthread_join(threads[0], NULL);
    pthread_join(threads[1], NULL);
    FinishSum();
    return 0;
}
```

(a) (3 points) In the fork version, suppose the child process runs and exits before the parent runs after fork. At what offset in array.dat will the parent's read() take place?

- ○ 0
- ○ ARRAY_SIZE/2
- ○ ARRAY_SIZE
- ○ something else: _____
- ○ nowhere because the file will be closed

(b) (3 points) In the fork() version, suppose the child process runs first after fork() up until the read() in SumPart() and the parent does not run, then the parent runs until waitpid(), then the child completes running. What portion of the file will the child see in its `arr` local variable when it computes its partial sum?

- ○ first half
- ○ second half
- ○ something else (such as a mix of the first and second half)

(c) (6 points) A user running the fork() above finds that it outputs a sum, but it never has the correct value. What could have happened? (Assume fork(), open(), etc. did not fail.) **Select all that apply.**

- ☐ Lack of synchronization on `sum` causes some updates (by both parent and child) to `sum` to get lost
- ☐ Parent overwrites the child's computed `sum`
- ☐ Child overwrites the parent's computed `sum`
- ☐ Child and parent `sum` variables are in separate locations in physical memory, and only one of them is output by FinishSum()
- ☐ the read()s in the child and parent both read the back half of the array
- ☐ the read()s in the child and parent both read the front half of the array

(d) (7 points) A user running the pthread version finds that it sometimes outputs the wrong sum. Identify potential bugs that could cause this problem. (Assume pthread_create(), open(), etc. did not fail.) **Select all that apply.**

- ☐ Lack of synchronization on 'sum' causes some updates (from both threads) to 'sum' to get lost
- ☐ SumFront thread overwrites the SumBack thread's computed sum
- ☐ SumBack thread overwrites the SumFront thread's computed sum
- ☐ SumFront and SumBack threads' `sum` variables are in separate locations in physical memory, and only one of them is output by FinishSum()
- ☐ SumFront and SumBack and main() thread's `sum` variables are in separate locations in physical memory, and only one of them is output by FinishSum()
- ☐ the read()s in the two threads might both read the back half of the array
- ☐ the read()s in the two threads might both read the front half of the array

5. Consider a pipelined processor with the following pipeline stages:

   1. Fetch 1 (start reading instruction, predict next program counter)
   2. Fetch 2 (finish reading instruction)
   3. Decode (read registers)
   4. Execute (perform computations)
   5. Memory 1 (start data cache operation)
   6. Memory 2 (complete data cache operation)
   7. Writeback (write registers)

   Assume the processor uses a combination of forwarding and stalling to resolve data hazards. The processor implements all forwarding paths that do not dramatically increase the cycle time.

(a) Suppose the processor is executing the following instructions:

```
1. addq %r9, %r12
2. subq %r12, %r11
3. movq (%r9), %r12
4. xorq %r12, %r9
5. addq %r12, %r13
6. movq (%r13), %r12
```

i. (3 points) Instruction 4 will obtain `%r12`'s value by _____.

- ◯ forwarding it from instruction 1
- ◯ forwarding it from instruction 2
- ◯ forwarding it from instruction 3
- ◯ reading it from the register file during its decode stage
- ◯ reading it from the data cache during its memory stage

ii. (3 points) Instruction 4 will obtain `%r9`'s value by _____.

- ◯ forwarding it from instruction 1
- ◯ forwarding it from instruction 2
- ◯ forwarding it from instruction 3
- ◯ reading it from the register file during its decode stage
- ◯ reading it from the data cache during its memory stage

iii. (4 points) If instruction 1 (`addq %r9, %r12`) completes its fetch 1 stage in cycle 0, then the instruction 6 (`movq (%r13), %12`) will complete its writeback stage in cycle _____.

6. Consider the following assembly snippet where the conditional jump to the label `elsewhere` is not taken:

```
xorq %r9, %r12
addq %r9, %r10
jne elsewhere
movq (%r10), %rdi
movq %rdi, (%rsi)
subq %rdi, %r12
```

(a) (5 points) Which of the following is true about executing the above assembly snippet on an out-of-order processor with a design similar to that discussed in lecture and used in the OOO assignment? (Consider cases with variable branch predictions, number of instructions processed by stages per cycle, and availability of operand values from prior code.) **Select all that apply.**

- ☐ the movq instructions could be fetched before the conditional jump is finished decoding
- ☐ the movq instructions could be fetched *after* the conditional jump commits
- ☐ the movq instructions could commit before the conditional jump does
- ☐ the addq instruction could complete its register write before the xorq does
- ☐ the subq instruction could execute before the second movq completes its data cache access

(b) (5 points) On an out-of-order processsor, the instruction `movq %rdi, (%rsi)` could perform its data cache write at the same time as _____ performs its arithmetic, data cache operation, or condition code evaluation. **Select all that apply.**

- ☐ `xorq %r9, %r12`
- ☐ `addq %r9, %r10`
- ☐ `jne elsewhere`
- ☐ `movq (%r10), %rdi`
- ☐ `subq %rdi, %r12`

(c) (9 points) An exception could occur such that the processor would indicate to the operating system that the program counter was set to the address of the `movq %rdi, (%rsi)` instruction as a result of _____. **Select all that apply.**

- ☐ `%rdi` being used as a system call argument in an immediately following system call
- ☐ the page table entry corresponding to the program counter value not having its executable permission bit set to true
- ☐ the page table entry corresponding to the program counter value not having its write permission bit set to true
- ☐ the page table entry corresponding to the value of `%rdi` being invalid
- ☐ the page table entry corresponding to the value of `%rsi` being invalid
- ☐ a page table entry corresponding to the current stack pointer not having its write permission bit set to true
- ☐ a network packet being received just as the data cache access for the movq starts
- ☐ the network interface having the capacity to accept a new network packet just as the data cache access of the movq starts
- ☐ the data cache access for the movq being a cache miss

7. (20 points) Suppose we have a system that manages course enrollment. Each course has a list of enrolled students and a information about its locations and instructors.

   To support this system, we want a way to allow threads to obtain exclusive access to the list of enrolled students, the list of its instructors, or both. When any thread is waiting for exclusive access to both, we want to ensure it gets access *before* any other waiting thread — this means that any thread waiting for access to just the students or instructors should continue waiting whenever a thread is waiting for access to both.

   To do this, we implement three lock and corresponding unlock operations:

   - LockStudents(course), UnlockStudents(course)
   - LockInstructors(course), UnlockInstructors(course)
   - LockBoth(course), UnlockBoth(course)

   Complete the following incomplete implementation of this scheme using monitors by filling in the blanks.

```
struct Course {
    pthread_mutex_t lock; pthread_cond_t student_cv, instructor_cv, both_cv;
    bool active_students; bool active_instructors;
    int waiting_both; int waiting_students; int waiting_instructors;
    List student_list; List instructor_list;
};

void LockStudents(struct Course *course) {
    pthread_mutex_lock(&course->lock);
    course->waiting_students += 1;

    while (_____________________________________________________

           _____________________________________________________) {
        pthread_cond_wait(&course->student_cv, &course->lock);
    }
    course->waiting_students -= 1;
    course->active_students = true;
    pthread_mutex_unlock(&course->lock);
}

void LockInstructors(struct Course *course) {
    pthread_mutex_lock(&course->lock);
    course->waiting_instructors += 1;
    while (_____________________________________________________

           _____________________________________________________) {
        pthread_cond_wait(&course->instructor_cv, &course->lock);
    }
    course->waiting_instructors -= 1;
    course->active_instructors = true;
    pthread_mutex_unlock(&course->lock);
}
```

(continued on next page)

```
void LockBoth(struct Course *course) {
    pthread_mutex_lock(&course->lock);
    course->waiting_both += 1;
    while (_____________________________________________________

           _____________________________________________________) {
        pthread_cond_wait(&course->both_cv, &course->lock);
    }
    course->waiting_both -= 1;
    course->active_instructors = true;
    course->active_students = true;
    pthread_mutex_unlock(&course->lock);
}

void UnlockStudents(struct Course *course) {
    pthread_mutex_lock(&course->lock);
    course->active_students = false;
    if (course->waiting_both > 0)
        pthread_cond_signal(&course->both_cv);
    else
        pthread_cond_signal(&course->student_cv);
    pthread_mutex_unlock(&course->lock);
}

void UnlockInstructors(struct Course *course) {
    pthread_mutex_lock(&course->lock);
    course->active_instructors = false;
    if (course->waiting_both > 0)
        pthread_cond_signal(&course->both_cv);
    else
        pthread_cond_signal(&course->instructor_cv);
    pthread_mutex_unlock(&course->lock);
}

void UnlockBoth(struct Course *course) {
    pthread_mutex_lock(&course->lock);
    course->active_students = course->active_instructors = false;
    if (course->waiting_both > 0)
        pthread_cond_signal(&course->both_cv);
    else {

        ____________________________________________________

        ____________________________________________________
    }
    pthread_mutex_unlock(&course->lock);
}
```

8. (7 points) Suppose a University has a system for course registration where students run a program on their computers that communicates with a University server.

   In addition to the University having a certificate signed by a third-party certificate authority containing the server's public key, this program uses another set of certificates: As part of distributing that program to students, unique certificates are generated for each student and included with each student's copy of the client program. These certificates contain the student's identity, a public key that was originally generated on the student's computer, and a digital signature generated by the University.

   Based on this information, which statements are most likely true about how the client and server use the certificate generated for a particular student?

   Select all that apply.

   - ☐ the client uses the certificate to verify that it is talking to the correct University server
   - ☐ the client cannot operate without having the private key that corresponds to the public key stored in the certificate
   - ☐ the client cannot operate without having the private key that was used to generate the digital signature in the certificate
   - ☐ the client cannot operate without having a public key that can be used to verify the signature on its certificate
   - ☐ the server uses the certificate to verify that it is talking to the correct student's client machine
   - ☐ the protocol the client uses to communicate is vulnerable to a replay attack unless a new certificate is generated for each communication
   - ☐ the University server has public keys stored for all students so it can check the certificates