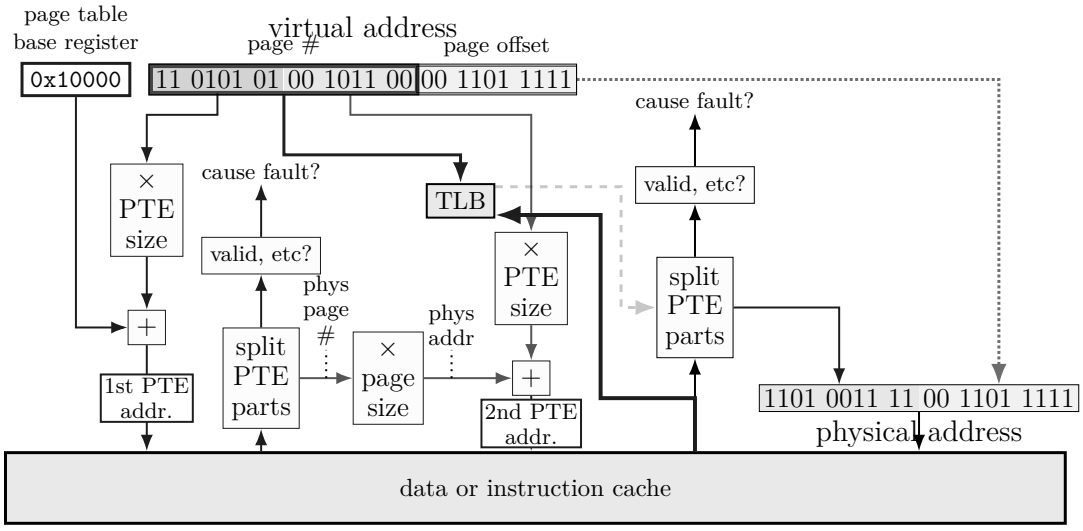
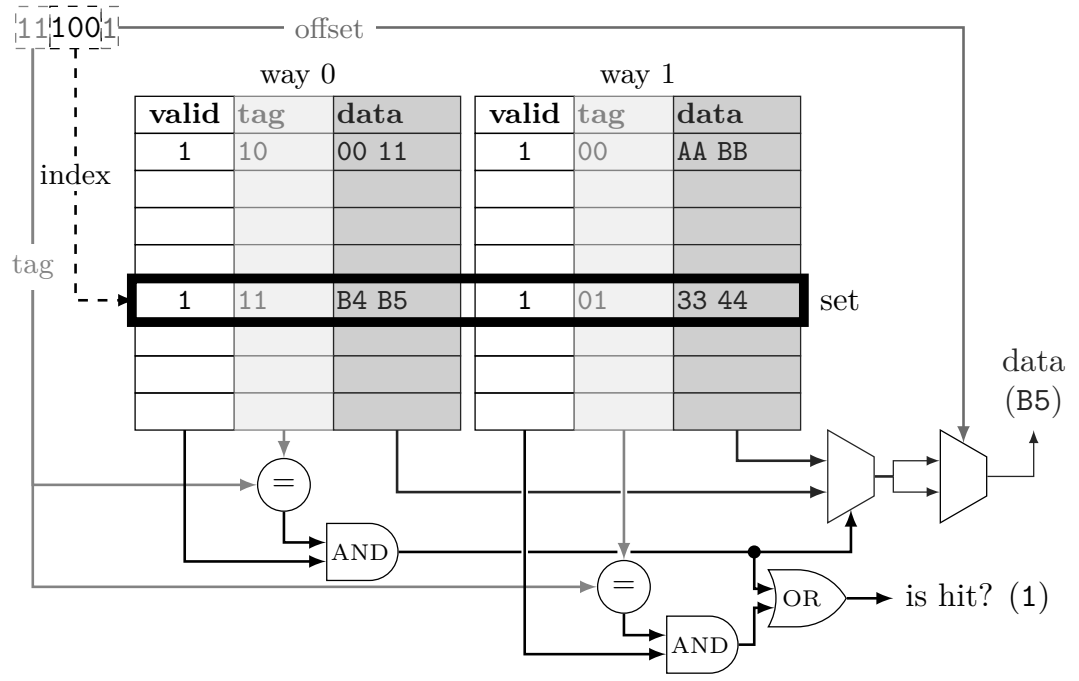
1 page table lookup



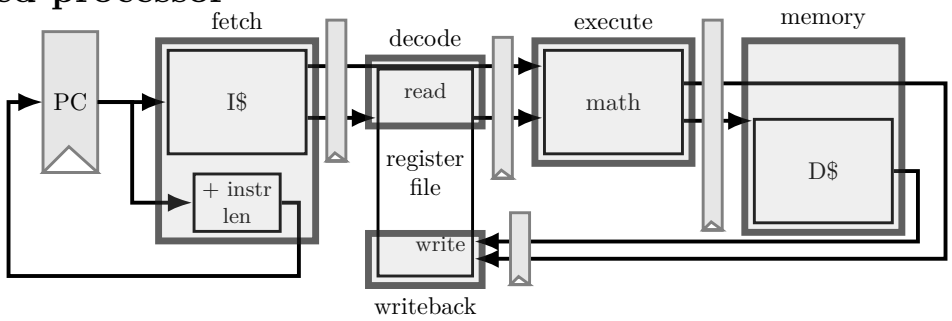
2 cache organization



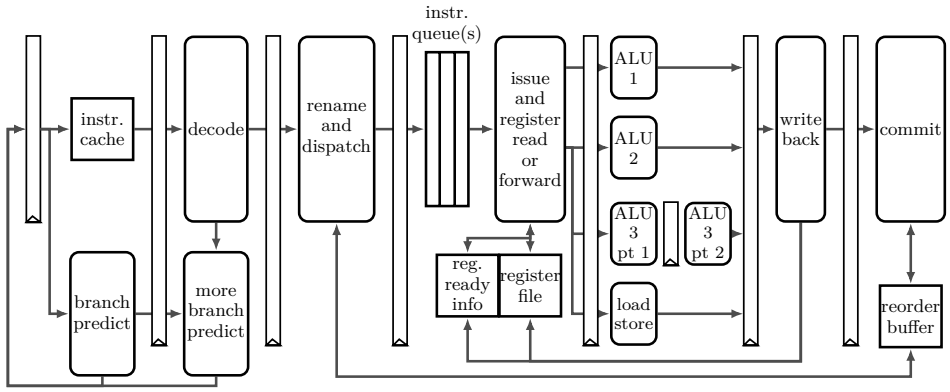
3 networking layers

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 pipelined processor



5 OOO processor



6 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)
  - `waitpid(pid, 0, NULL)` wait for the child process with ID `pid` to terminate
  - `kill(pid, signal_number)` — send signal `signal_number` to process `pid`

7 classic Spectre pattern

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

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`

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. A dual-core processor has a two-level cache hierarchy. The L1 cache uses a two-way set-associative cache with 16 sets, a LRU eviction policy, and 8 byte blocks. The L2 cache uses a direct-mapped cache with 32 sets and 64 byte blocks. Memory reads take 1 cycle if handled by the L1 cache, 2 cycles if handled by the L2 cache, and 10 cycles if handled by main memory.

   For this set of questions, assume all addresses are physical addresses.

(a) (8 points) A process running on Core0 reads two bytes from physical address `0x158A`. Suppose caches started empty and this is the only memory operation in the system. Fill in the tables below to show how the caches will be updated. (You may not need all lines. Omit set indices that are not changed from the empty state. You may leave answers as unsimplified arithmetic expressions.)

Indicate a value retrived from physical addresses 0x1234 through 0x1235 inclusive with "MEM[0x1234-0x1235]".

L1 (Core0)

set index	way	tag	data	valid	lru
	way 0				
	way 1				
	way 0				
	way 1				

L1 (Core1)

set index	way	tag	data	valid	lru
	way 0				
	way 1				
	way 0				
	way 1				

L2

set index	tag	data	valid

(b) (3 points) The very first read in the system takes _____ cycles to complete.

(c) (6 points) A process running on Core0 clears its L1 cache and does a read to physical address `0x85C4`. The process observes the read only took 2 cycles to complete. What is a **different** memory address that a process running on Core1 could have accessed to make this happen?

2. A processor executes a series of `addq` instructions to compute `x = a + (b + c)`, `y = x + (c + d)`, and `z = x + d`. Initially, `a` is in `%r8`, `b` is in `%r9`, `c` is in `%r10`, and `d` is in `%r11`.

```
1. addq %r10, %r9  // b = b + c
2. addq %r9, %r8   // x = a + (b + c)
3. addq %r11, %r10 // c = c + d
4. addq %r8, %r10  // y = x + (c + d)
5. addq %r8, %r11  // z = x + d
```

(a) (4 points) Use register renaming to rewrite these instructions. Update the Arch-Phys register mapping and the free list.

1. `addq` _____ , _____ –> _____
2. `addq` _____ , _____ –> _____
3. `addq` _____ , _____ –> _____
4. `addq` _____ , _____ –> _____
5. `addq` _____ , _____ –> _____

Arch Reg	Phys Reg		Free List
%r8	%x1		%x10
			%x11
%r9	%x2		%x12
			%x13
%r10	%x3		%x14
			%x15
%r11	%x4		%x16

(b) (2 points) After rewriting, there are _____ free registers remaining.

(c) (4 points) Which instruction number(s) can the processor issue immediately?

_____

(d) (4 points) How many cycles will it take for all of these instructions to execute? Assume a large number of single-cycle ALU elements.

_____

3. Consider a system with:
   - three-level page table structure, where each page table takes up 1 page of space
   - 65536 (2^{16}) byte pages
   - 8-byte page table entries (so each page table stores 2^{13} entries)
   - a 2-way, 16-entry (2^4 entry) TLB with an LRU replacement policy

(a) (4 points) Suppose a program can access 2^{30} bytes of memory without an exception occurring. What is the minimum size, in bytes, of its page tables? (Be sure to include space needed to store any necessary invalid page table entries.)

(b) (4 points) What is the maximum number of distinct bytes of memory which could be accessed without a TLB miss? (You may leave your answer as an unsimplfied arithmetic expression.)

(c) (4 points) Assume the TLB starts empty. Complete the following table:

virtual address	physical address	TLB hit	TLB index
`0x0074445`	`0x84445`	no	
`0x04f4445`	`0x84445`		
`0x0084440`	`0xa4440`		
`0x007f430`	`0x8f430`		

4. Consider the following C code. The intention of this code is to have `CompileFiles` compile a list of source files, `max_threads` at a time.

```
1  struct FileInfo {
2      char *source_file;
3      bool compile_success;
4      bool joined;
5  };
6
7  void *CompileThread(void *file_info_raw) {
8      struct FileInfo* file_info; file_info = file_info_raw;
9      file_info->compile_success = DoCompile(file_info->source_file);
10     return NULL;
11 }
12
13 void CompileFiles(int max_threads) {
14     struct FileInfo files[NUM_FILES];
15     for (int i = 0; i < NUM_FILES; i += 1) {
16         files[i].source_file = GetSourceFile(i);
17         files[i].joined = false;
18     }
19     pthread_t threads[NUM_FILES];
20     for (int i = 0; i < num_files; i += 1) {
21         if (i - max_threads >= 0) {
22             int to_join = i - max_threads;
23             pthread_join(threads[to_join], NULL); files[to_join].joined = true;
24         }
25         pthread_create(&threads[i], NULL, CompileThread, &files[i]);
26     }
27     for (int i = 0; i < NUM_FILES; i += 1) {
28         if (!files[i].joined) pthread_join(threads[i], NULL);
29     }
30     ReportCompiledFiles(files);
31 }
```

(a) (4 points) The line `file_info->compile_success = DoCompile(file_info->source_file)` modifies a value stored _____.

- ◯ on the heap
- ◯ in the region of memory used for global variables
- ◯ on the stack of a thread which ran `CompileFiles`
- ◯ either on the stack or in a register of a thread which ran `CompileFiles`
- ◯ on the stack of a thread started by a `pthread_create` call in `CompileFiles`
- ◯ on the stack of a thread started by a `pthread_create` call in 'CompileFiles or in a register of such a thread
- ◯ none of the above, explain briefly: _____

(b) (4 points) Instead of starting a new thread for each `DoCompile` call, we could start `max_threads` threads and have each of them potentially make multiple `DoCompile` calls. What would be true about implementing the synchronization this design? **Select all that apply.**

- ☐ of the synchronization tools we learned this semester, it would be most natural to write using barriers for synchronization
- ☐ if using monitors, the threads running `DoCompile` would call `pthread_cond_wait`
- ☐ if using monitors, it would be best to have a separate mutex for each thread started
- ☐ if using monitors, it would be best to have a separate condition variable for each thread started

5. Consider the following C code:

```
struct Process {
    long is_active;
    long parent_process_id;
    long user_id;
    ...
};
struct Process processes[16384];
long GetParentProcessUser(long process_id) {
    if (process_id >= 0 && process_id < 16384) {
        struct Process *this_process;
        this_process = &processes[process_id];
        if (this_process->is_active) {
            struct *parent_process;
            parent_process = &processes[this_process->parent_process_id];
            return parent_process->user_id;
        } else {
            return -1;
        }
    } else {
        return -1;
    }
}
```

Assume that:

- `long`s are 8 bytes and `struct Process` is 128 bytes in size
- `processes[0]` is located at address `0x100000`
- the machine only uses physical addresses
- the machine has 65536 byte (2^{16} byte) direct-mapped data cache with 16-byte blocks

(a) (4 points) Suppose we learn that a call to this function with a process ID returns -1 because that process isn't active and that that call evicts from cache set 48. What is a possible value of the process ID it could have been called with? Assume that only accesses to the `processes` array (directly or via a pointer) use the data cache. You may leave your answer as an unsimplified arithmetic expression.

(b) (6 points) If an attacker provided a `process_id` value of `65536`, then a Spectre-style attack would reveal information about the contents of what physical address? You may leave your answer as unsimplified arithmetic expression. If the Spectre attack could not work in this case, write "none" and explain **briefly**.

6. Consider an in-order, 5-stage pipelined processor.

(a) (3 points) Write a sequence of two instructions which have a data hazard but would **not** cause a processor stall.

(b) (4 points) Write a sequence of instructions that would cause a forward from the memory stage to the decode stage without stalling.

(c) (4 points) Write a sequence of instructions which will have a different set of values forwarded if a branch is predicted correctly than if it is not. Indicate whether any conditional jump in your sequence is taken or not taken, and which values are forwarded when it is predicted correctly and when it is not.

(d) (4 points) Suppose we run a program and count the instructions that complete execution (counting any instruction that runs N times, N times) and find:

- 5% of its instructions are mispredicted branches;
- 10% are correctly predicted branches;
- 2% are memory-reading instructions immediately followed by an instruction that does arithmetic with the value read;
- 3% are arithmetic instructions immediately followed by an instruction that writes the result of the arithmetic to memory;
- 20% are arithmetic instructions immediately followed by an arithmetic instruction that uses the result of the arithmetic; and
- all other instructions are instructions that do not have dependencies requiring stalling or forwarding;

If the processor has a cycle time of 10^{-9} s and there are 10^{9} instructions run in total, how long (to the nearest hundredth of a second) did those take to run?

7. A developer is building a shared memory application that runs as two processes on two separate machines connected by a network.

(a) (4 points) The developer wants to share a buffer of data from one process to the other running on the remote machine. Their first attempt looks like:

```
void share() {
  size_t data[100] = {1, 2, 3, ...};
  send(&data, size_of(size_t*));
}
```

(Assume send() takes a pointer to bytes and a number of bytes to send.)

Why is this guaranteed to fail? **Select all that apply.**

- ☐ The buffer `data` is allocated on the stack and not the heap.
- ☐ A packet could be lost when the message is sent.
- ☐ The sending process does not hold a lock when sending the message.
- ☐ A copy of the data may be stored in a cache.
- ☐ The message does not include the length of the buffer (i.e., 100).
- ☐ The sender didn't call `mmap` to setup new page table entries for the other process.
- ☐ The receiver process may have a different userid with different permissions.
- ☐ The buffer `data` is only in memory on one machine.

(b) (4 points) The developer makes a second attempt, this time trying to share a physical memory address rather than a virtual memory address. The sending process has the following page table with 4096 byte pages. If the buffer is at virtual address `0x9312C4`, what is the correct physical address?

virtual page number	physical page number	valid
0x1	0x15	1
0x93	0x0	0
0x2C4	0x99B	1
0x312	0x44	1
0x931	0x571	1

(c) (4 points) The developer decides to fix the application by sending the data in the buffer instead of dealing with addresses directly. But the application is still not working. The developer thinks packets are getting corrupted in the network. What can the developer do to address packet corruption?

(d) (4 points) The developer fixes the application and gets it working. The main sharing operation now requires one machine to reliably send 20 packets to the other. It takes 50 ms for a packet to go from one machine to the other. When a machine sends a packet, it does not send a packet with new data until it has verified the first one has reached the destination. Assuming no packets are lost or corrupted, how long will it take to for the sharing operation to finish? You may leave your answer as an unsimplified arithmetic expression.

8. A programmer writes the following for a program that starts four threads running the `check_average` function. These threads finish when the average of the measurements exceeds a threshold.

   Assume:

   - the four threads are started correctly
   - each `struct thread_info` is created and initialized correctly
   - the `take_measurement()` function exists
   - all thread/lock functions succeed, and
   - the pointers in each `struct thread_info` point to the same set of locks, condition variables, and values

   Consider the following questions independently (i.e., any modifications do not carry from one subquestion to another.)

```
1  struct thread_info {
2    int              index;
3    pthread_mutex_t* lock;
4    pthread_cond_t*  cv;
5    int*             counts;
6    bool*            changed;
7  }
8
9  void *timer_thread(void *raw_ptr) {
10   struct thread_info* info = (struct thread_info*) raw_ptr;
11   while (1) {
12     pthread_mutex_lock(info->lock);
13     info->counts[info->index] = take_measurement();
14     *info->changed = true;
15     pthread_cond_broadcast(info->cv);
16     pthread_mutex_unlock(info->lock);
17
18     sleep(TIMER_INTERVAL);
19   }
20 }
21
22 void check_average(struct thread_info* info) {
23   // Create a new thread periodically sample new data.
24   pthread_t time_thread;
25   pthread_create(&time_thread, NULL, timer_thread, info);
26
27   pthread_mutex_lock(info->lock);
28   while (1) {
29     while (!*info->changed) { pthread_cond_wait(info->cv, info->lock); }
30     *info->changed = false;
31     // Check the average across threads. If > THRESHOLD, stop.
32     if ((sum(info->counts) / 4) > THRESHOLD) break;
33   }
34   pthread_mutex_unlock(info->lock);
35   printf("Finished! %i\n", info->index);
36 }
```

(see previous page)

(a) (5 points) The `check_average` function does not call `pthread_join` on `time_thread`. This would seem to indicate that the thread won't finish running. Which of the following would be options for fixing this issue without breaking `check_average` otherwise? **Select all that apply.**

- ☐ add a new bool to `thread_info`, intialized to false; set it to true at the end of `check_average` and then call `pthread_join`; in `timer_thread` check the bool in the loop and break if it is true
- ☐ change the `while (1)` in `timer_thread` to `while (!*info->changed)` and call `pthread_join` at the end of `check_average`
- ☐ change the `while (1)` in `timer_thread` to `while (info->cv)` and call `pthread_join` at the end of `check_avefage`
- ☐ make `check_average()` call `pthread_detach` on the thread
- ☐ make `timer_thread` check `(sum(info->counts) / 4) > THRESHOLD` itself and break out of its loop and add a `pthread_join` call at the end of `check_average`
- ☐ no fix is needed; the thread will be terminated when `check_average` exits

(b) (4 points) When the four threads are started, each executing `check_average()`, of the following options what will happen? **Select all that apply.**

- ☐ The program will deadlock because the first `check_average()` thread that acquires the lock will hold it in a `while(1)` loop.
- ☐ The program will deadlock because the `cond_broadcast()` call notifies multiple threads but only one can hold the lock.
- ☐ The program will deadlock because the `timer_thread()` thread will eventually try to acquire the lock while its parent thread holds it.
- ☐ The program will not deadlock because no thread indefinitely holds the lock.
- ☐ The program will not deadlock because all monitors prevent deadlock.
- ☐ The program will not deadlock because `pthread_cond_broadcast()` notifies all waiting threads, one of which will release the lock.

(c) (4 points) A new programmer decides the lock/unlock in `timer_thread()` is unnecessary and removes those two function calls. What could happen? **Select all that apply.**

- ☐ The program will not compile because `info->counts` is modified without holding a lock.
- ☐ All `check_average()` threads always observe `*info->changed == false`.
- ☐ A `check_average()` thread computes an invalid average.
- ☐ One or more `timer_thread()`s stop taking measurements.

(d) (4 points) The original programmer notices the program takes longer to finish than expected. Which change will help? **Select all that apply.**

- ☐ Replace `pthread_cond_broadcast()` with `pthread_cond_signal()`.
- ☐ Remove `*info->changed = false;`.
- ☐ Add a different condition variable used by `timer_thread()`.
- ☐ Use four `changed` flags, one for each `check_average()` thread.

9. Suppose a single-core system is running a single process, Process A, in user mode. Then, the following sequence of events happens:

    1. Process A opens a file for reading
    2. Process A calls `malloc()` which calls the `sbrk` system call
    3. The kernel allocates a physical page for Process A
    4. Process A stores an `int` in the newly allocated memory at virtual address 0x4000050000
    5. The processor stores the `int` in a cache
    6. Process A calls `fork()`, creating Process B
    7. A keyboard button is pressed
    8. Process B calls `exec()`
    9. Process A calls `waitpid` to wait on Process B, and `waitpid` returns immediately
    10. Process B calls `exit`

(a) (4 points) When will an exception occur, causing the kernel to start executing? Write the numbers of the steps listed above.

(b) (4 points) Is it possible for Process B to benefit from the `int` saved in the cache in step 5? Explain your answer.

(c) (4 points) Assume Process B attempted to read the value of the `int` stored at virtual address 0x4000050000 (same as in step 4), and a segmentation fault occurred. In this sequence, '`Process B atteempts to read the` int' at virtual address 0x4000050000 causing a segmentation fault" would best fit immediately after step _____.