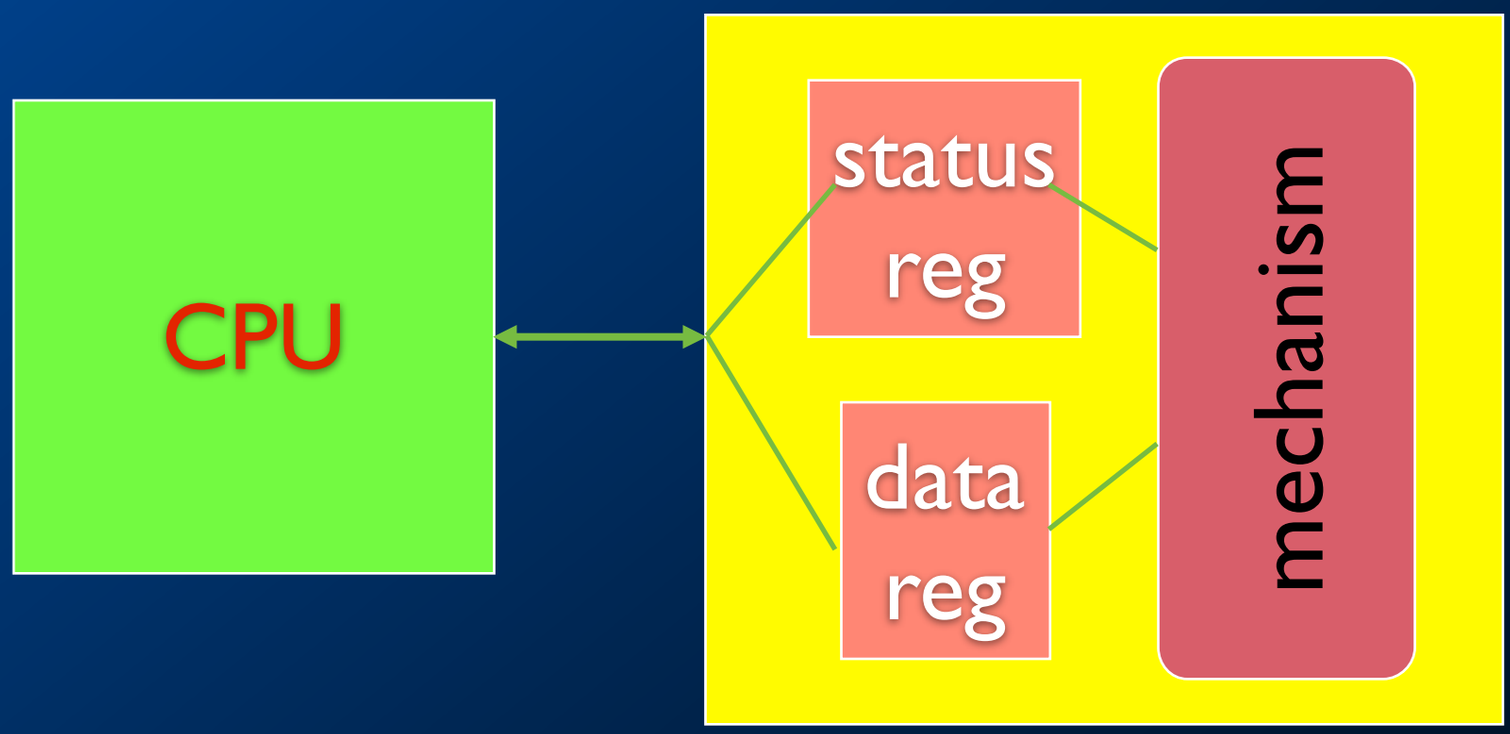
Bus-based computering systems

Agenda

- Input and output
- Busses
- Memory Architectures

I/O devices

- Usually includes some non-digital component.
- Typical digital interface to CPU:



Application: 8251 UART

- Universal asynchronous receiver transmitter (UART) : provides serial communication.
- 8251 functions are integrated into standard PC interface chip.
- Allows many communication parameters to be programmed.

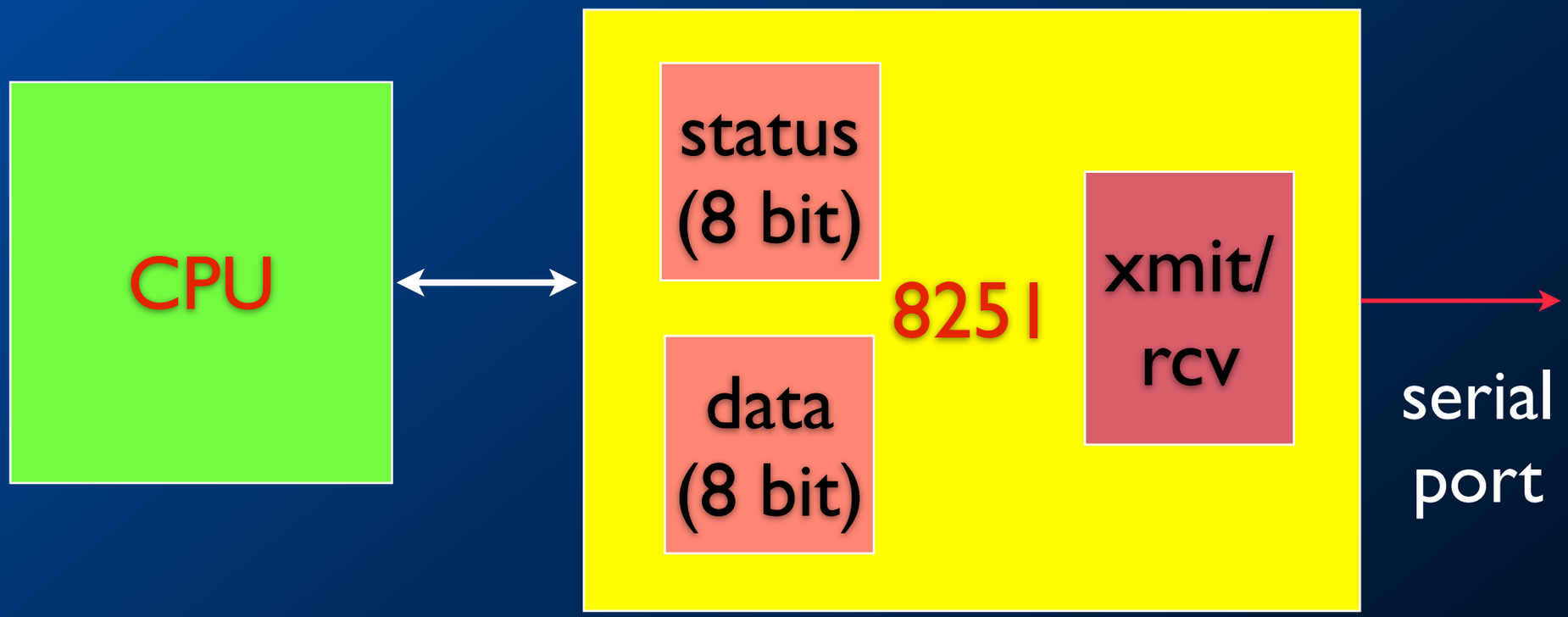
Serial communication

- Characters are transmitted separately:

Serial communication parameters

- Baud (bit) rate.
- Number of bits per character.
- Parity/no parity.
  - Even/odd parity.
- Length of stop bit (1, 1.5, 2 bits).

8251 CPU interface



Programming I/O

- Two types of instructions can support I/O:
  - special-purpose I/O instructions;
  - memory-mapped load/store instructions.
- Intel x86 provides in, out instructions. Most other CPUs use memory-mapped I/O.
- I/O instructions do not preclude memory-mapped I/O.

ARM memory-mapped I/O

- Define location for device:

```
DEV1 EQU 0x1000
```

- Read/write code:

```
LDR r1,#DEV1 ; set up device adrs
LDR r0,[r1] ; read DEV1
LDR r0,#8 ; set up value to write
STR r0,[r1] ; write value to device
```

Peek and poke

- Traditional HLL interfaces:

```
int peek(char *location) {
    return *location; }

void poke(char *location, char newval) {
    (*location) = newval; }
```

Busy/wait output

- Simplest way to program device.
  - Use instructions to test when device is ready.

```
current_char = mystring;
while (*current_char != '\0') {
   poke(OUT_CHAR,*current_char);
   while (peek(OUT_STATUS) != 0);
   current_char++;
}
```

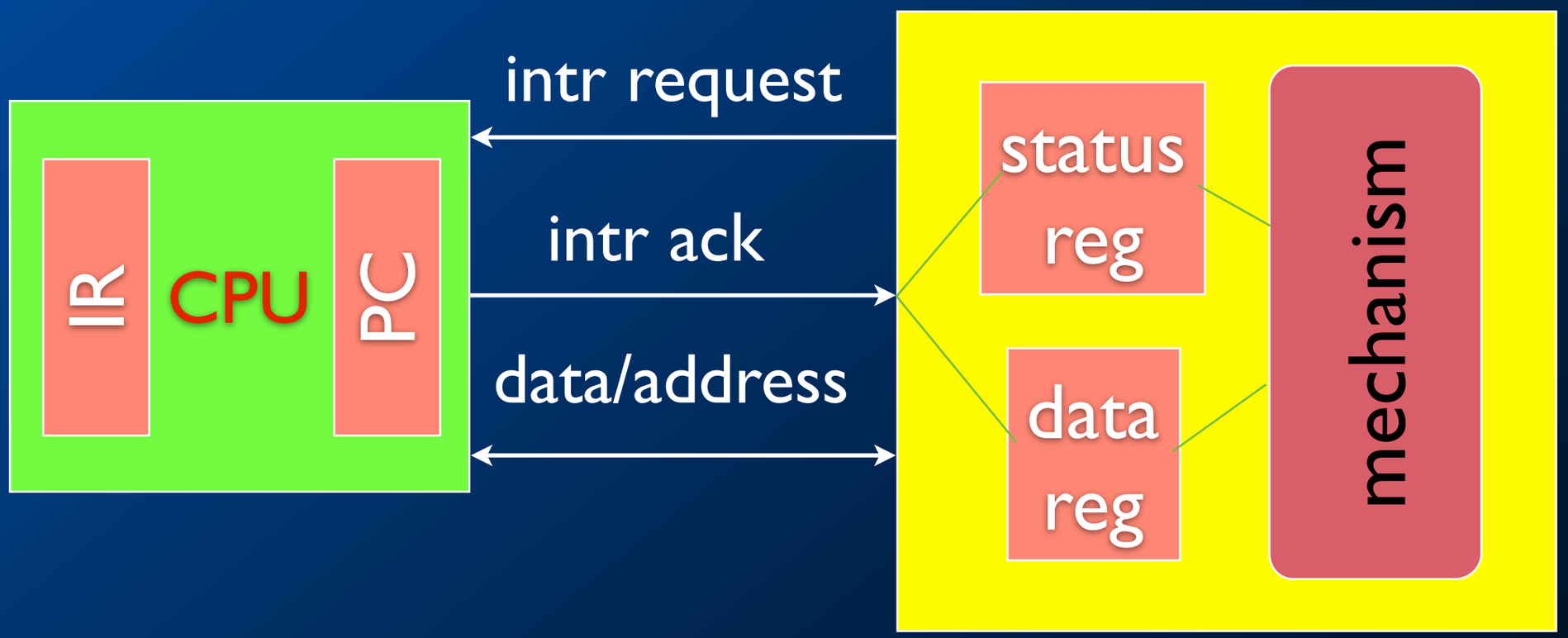
Simultaneous busy/wait input and output

```
while (TRUE) {
    /* read */
    while (peek(IN_STATUS) == 0);
    achar = (char)peek(IN_DATA);
    /* write */
    poke(OUT_DATA,achar);
    poke(OUT_STATUS,1);
    while (peek(OUT_STATUS) != 0);
}
```

Interrupt I/O

- Busy/wait is very inefficient.
  - CPU can't do other work while testing device.
  - Hard to do simultaneous I/O.
- Interrupts allow a device to change the flow of control in the CPU.
  - Causes subroutine call to handle device.

Interrupt interface



Interrupt behavior

- Based on subroutine call mechanism.
- Interrupt forces next instruction to be a subroutine call to a predetermined location.
- Return address is saved to resume executing foreground program.

Interrupt physical interface

- CPU and device are connected by CPU bus.
- CPU and device handshake:
- device asserts interrupt request;
- CPU asserts interrupt acknowledge when it can handle the interrupt.

Example: character I/O handlers

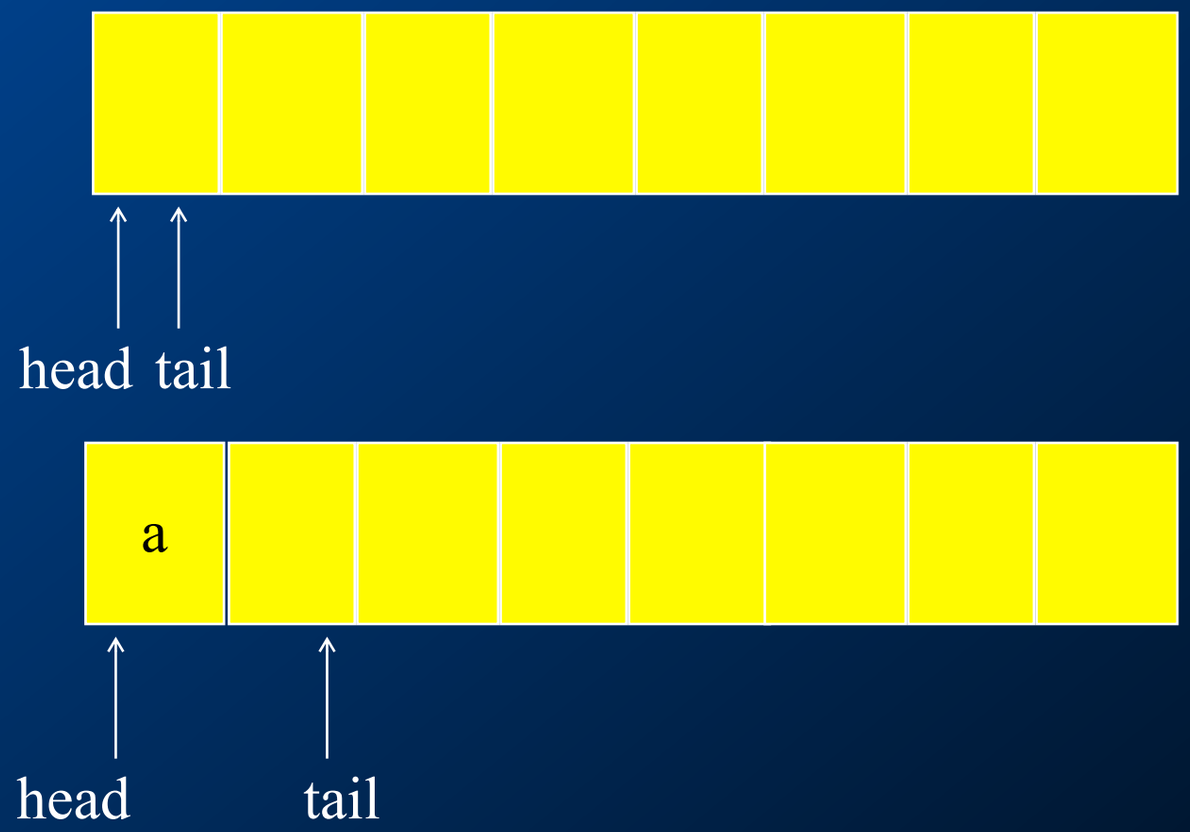
```
void input_handler() {
    achar = peek(IN_DATA);
    gotchar = TRUE;
    poke(IN_STATUS,0);
}
void output_handler() {
}
```

Example: interrupt-driven main program

```
main() {
    while (TRUE) {
        if (gotchar) {
            poke(OUT_DATA,achar);
            poke(OUT_STATUS,1);
            gotchar = FALSE;
        }
    }
    other processing....
}
```

Example: interrupt I/O with buffers

- Queue for characters:

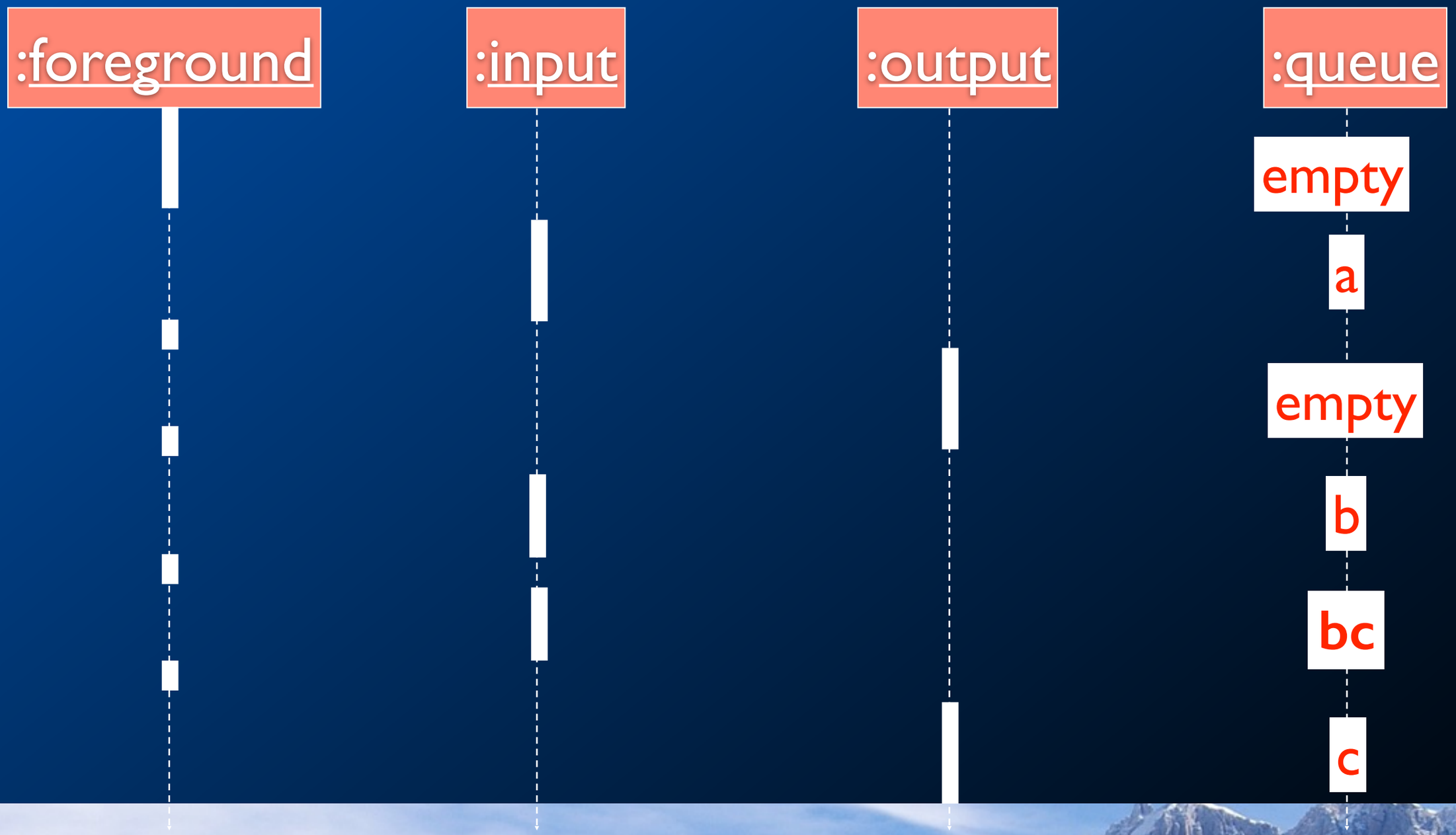


leave one empty slot to allow full buffer to be detected

Buffer-based input handler

```
void input_handler() {
      char achar;
      if (full_buffer()) error = 1;
      else {
            achar = peek(IN_DATA);
            add_char(achar); }
          poke(IN_STATUS,0);
          if (nchars == 1)
          {
            poke(OUT_DATA,remove_char();
            poke(OUT_STATUS,1); }
}
```

I/O sequence diagram



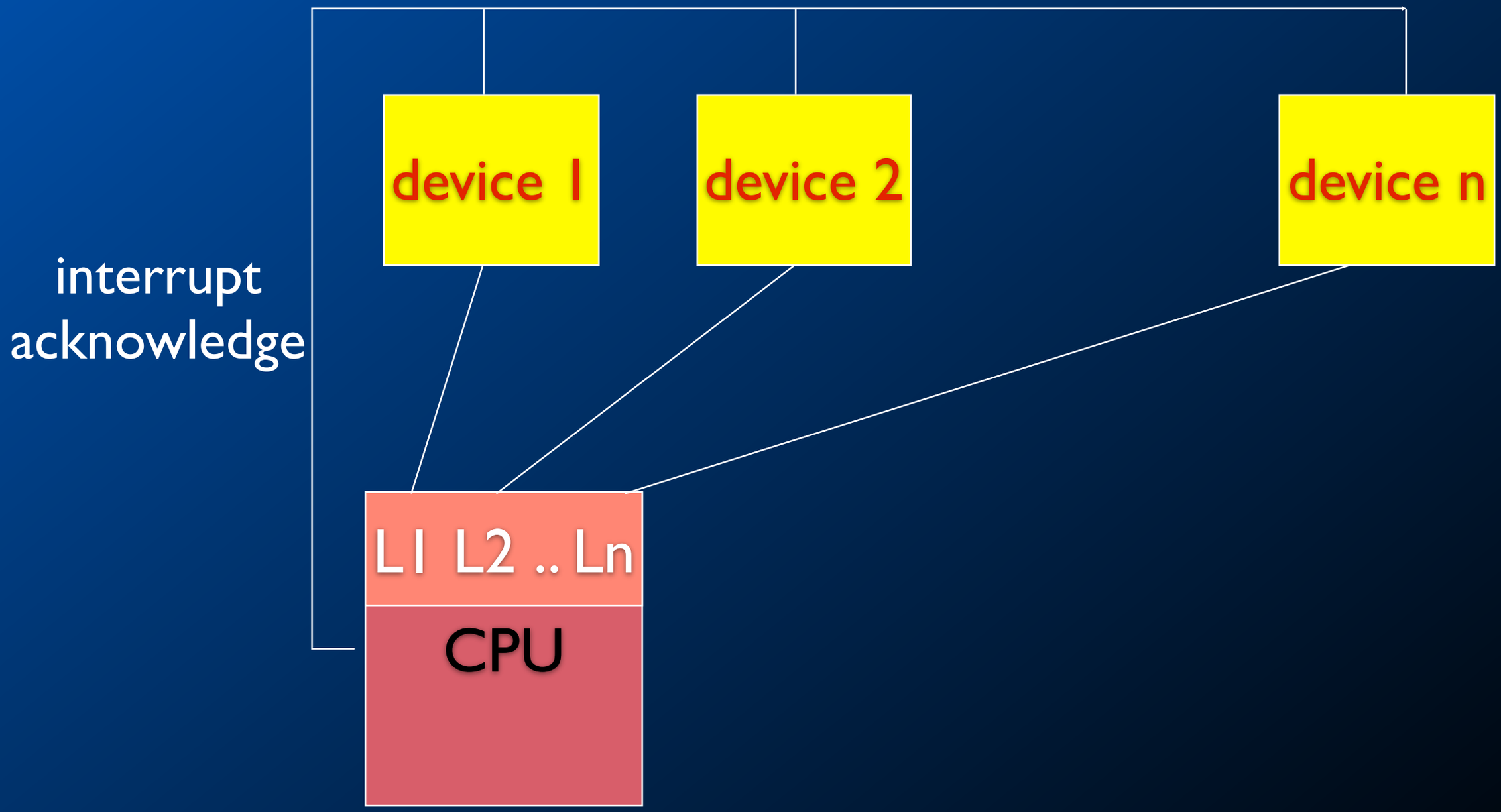
Debugging interrupt code

- What if you forget to change registers?
  - Foreground program can exhibit mysterious bugs.
  - Bugs will be hard to repeat---depend on interrupt timing.

Priorities and vectors

- Two mechanisms allow us to make interrupts more specific:
  - Priorities determine what interrupt gets CPU first.
  - Vectors determine what code is called for each type of interrupt.
- Mechanisms are orthogonal: most CPUs provide both.

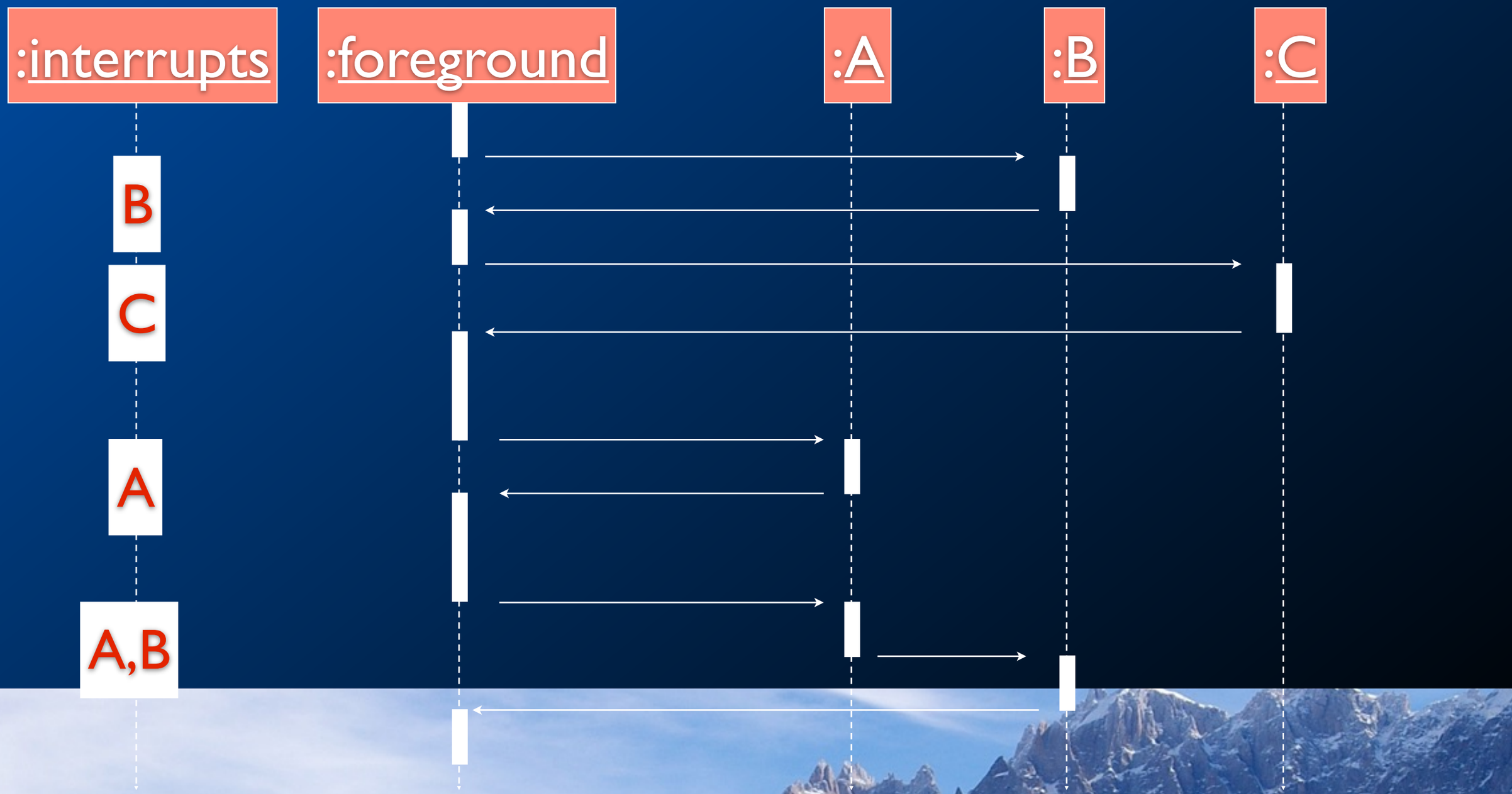
Prioritized interrupts



Interrupt prioritization

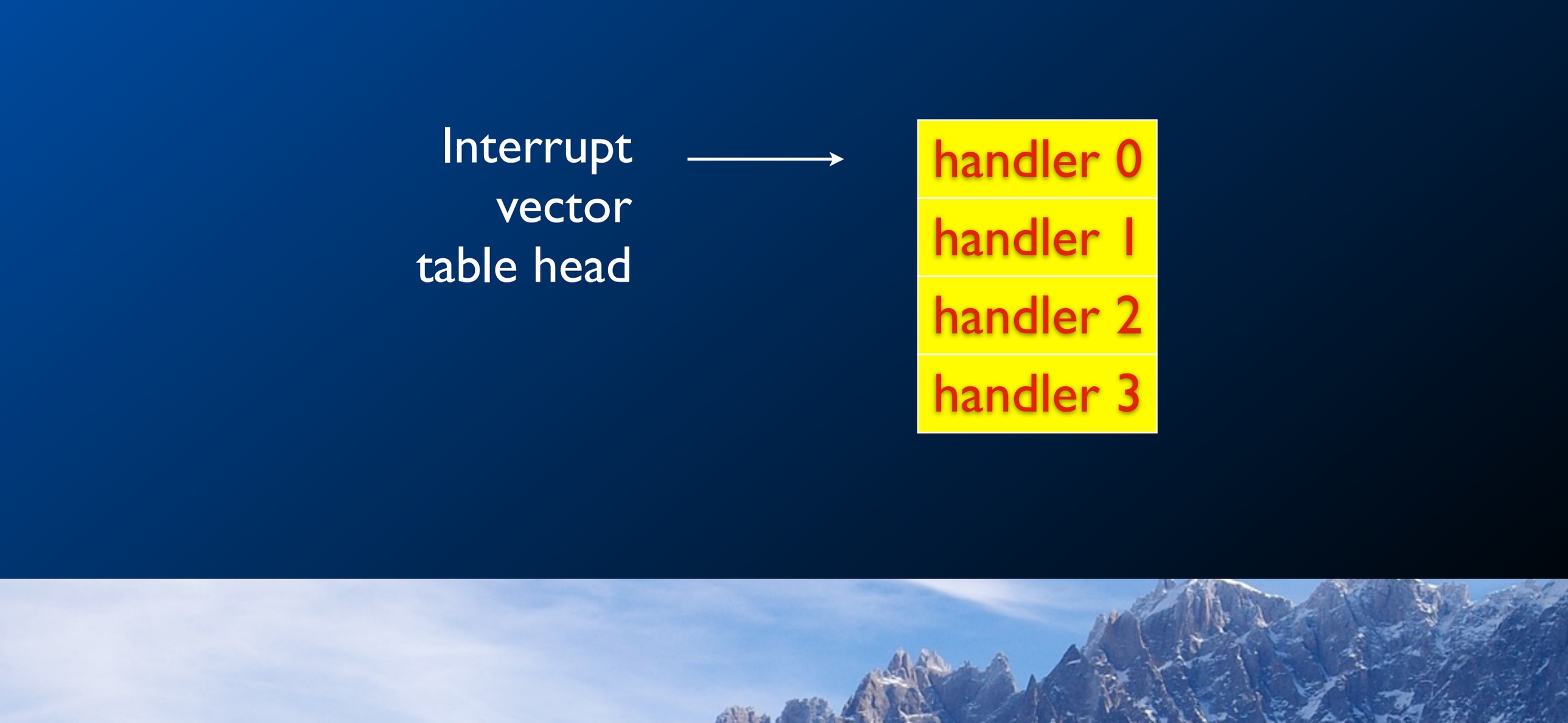
- Masking: interrupt with priority lower than current priority is not recognized until pending interrupt is complete.
- Non-maskable interrupt (NMI): highest-priority, never masked.
- Often used for power-down.

Example: Prioritized I/O

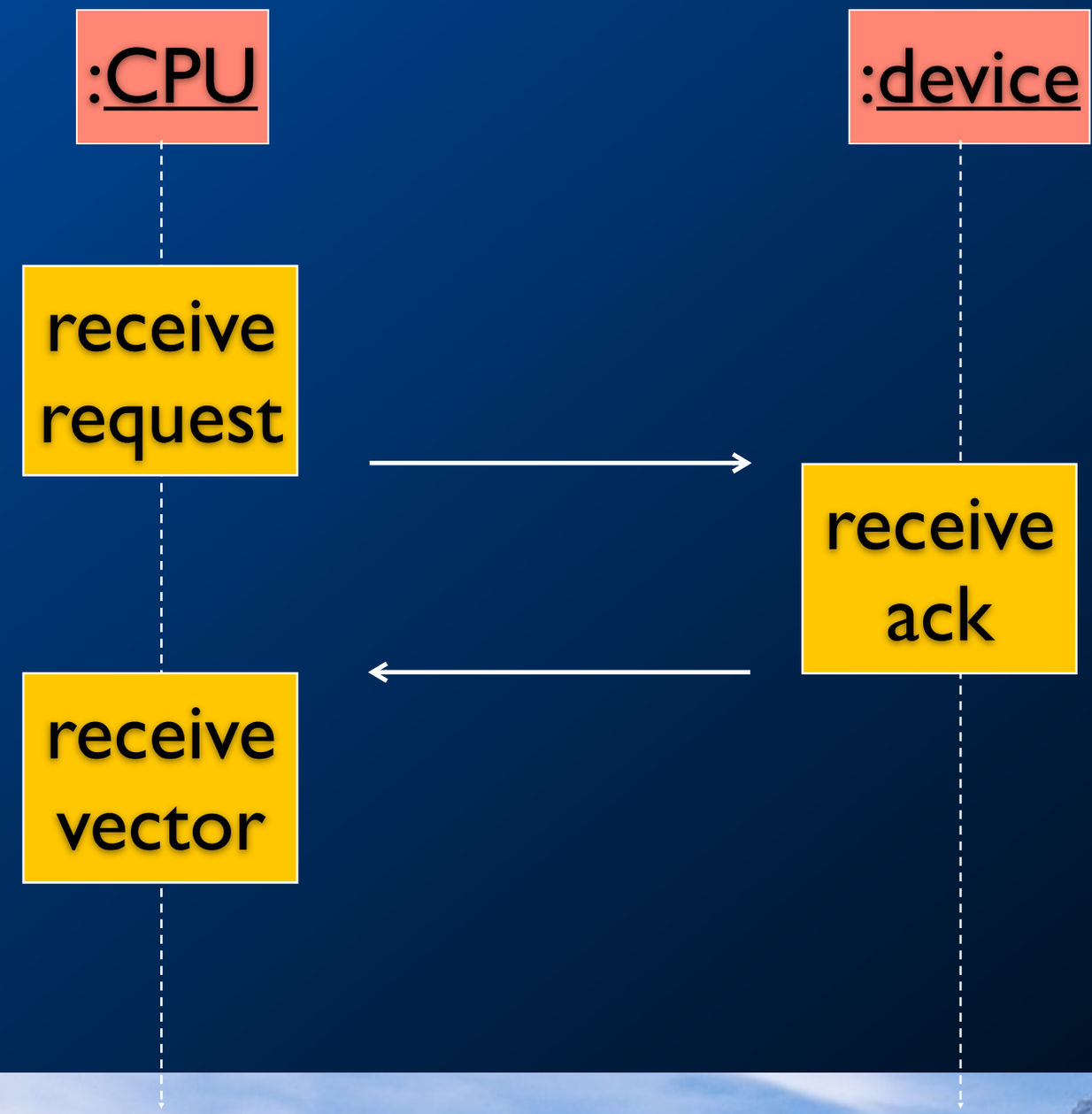


Interrupt vectors

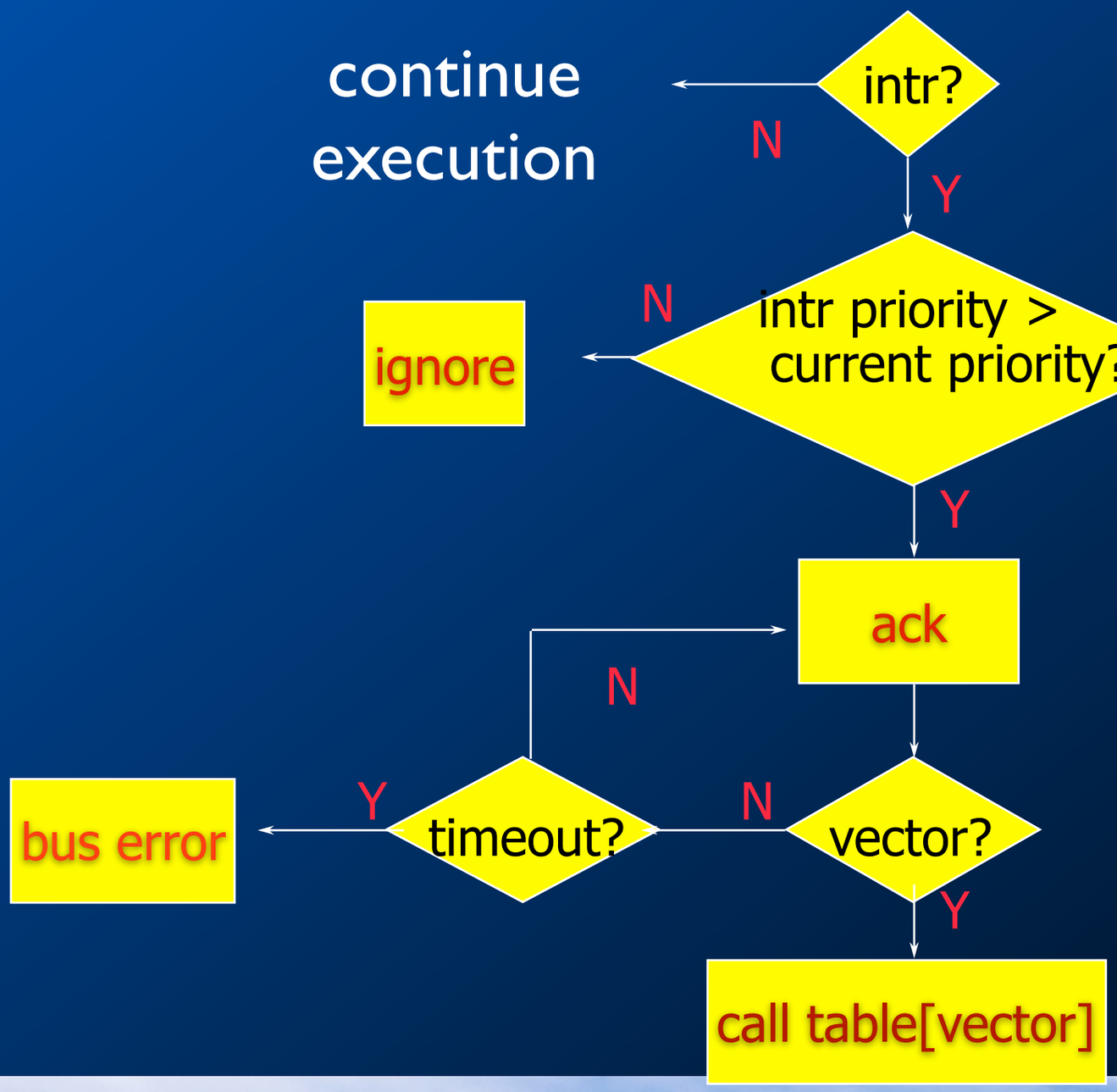
- Allow different devices to be handled by different code.
- Interrupt vector table:



Interrupt vector acquisition



Generic interrupt mechanism



Assume priority selection is handled before this point.

Interrupt sequence

- CPU acknowledges request.
- Device sends vector.
- CPU calls handler.
- Software processes request.
- CPU restores state to foreground program.

Sources of interrupt overhead

- Handler execution time.
- Interrupt mechanism overhead.
- Register save/restore.
- Pipeline-related penalties.
- Cache-related penalties.

Interrupt design guidelines

- While crummy code is just hard to debug, crummy ISRs are virtually undebuggable.
- how?
  - First, don't even consider writing a line of code for your new embedded system until you lay out an interrupt map. List each interrupt, and give an English description of what the routine should do.
  - The map is a budget. It gives you an assessment of where interrupting time will be spent.
  - Approximate the complexity of each ISR.
  - The cardinal rule of ISRs is to keep the handlers short.
  - Short, of course, is measured in time, not in code size. Avoid loops. Avoid long complex instructions (repeating moves, hideous math, and the like).
  - Re-enable interrupts as soon as practical in the ISR. Do the hardware-critical and non-reentrant things up front, then execute the interrupt enable instruction. Give other ISRs a fighting chance to do their thing.

- Figure An interrupt map

	Latency	Max-time	Freq	Description
INT1		1000 µsec	1000 µsec	timer
INT2		100 µsec	100 µsec	send data
INT3		250 µsec	250 µsec	Serial data in
INT4		15 µsec	100 µsec	write tape
NMI	200 µsec	500 µs	once!	System crash

- Fill all of your unused interrupt vectors with a pointer to a null routine.

```
Vect_table:
            dl    start_up            ; power up vector
            dl    null_isr            ; unused vector
            dl    null_isr            ; unused vector
            dl    timer_isr           ; main tick timer ISR
            dl    serial_in_isr       ; serial receive ISR
            dl    serial_out_isr      ; serial transmit ISR
            dl    null_isr            ; unused vector
            dl    null_isr            ; unused vector

null_isr:                             ; spurious intr routine
            jmp   null_isr            ; set BP here!
```

C or assembly?

- If the routine will be in assembly language, convert the time to a rough number of instructions. If an average instruction takes x microseconds (depending on clock rate, wait states, and the like), then it's easy to get this critical estimate of the code's allowable complexity.
- C is more problematic. In fact, there's no way to scientifically write an interrupt handler in C! You have no idea how long a line of C will take. You can't even develop an estimate as each line's time varies wildly. A string compare may result in a run-time library call with totally unpredictable results. A FOR loop may require a few simple integer comparisons or a vast amount of processing overhead.
- You may find one compiler pitifully slow at interrupt handling. Either try another or switch to assembly.

Debugging INT/INTA Cycles

- The device hardware generates the interrupt pulse.
- The interrupt controller (if any) prioritizes multiple simultaneous requests and issues a single interrupt to the processor.
- The CPU responds with an interrupt acknowledge cycle.
- The controller drops an interrupt vector on the data bus.
- The CPU reads the vector and computes the address of the user-stored vector in memory. It then fetches this value.
- The CPU pushes the current context, disables interrupts, and jumps to the ISR.

Finding Missing Interrupts

- You can build a little circuit using a single up/down counter that counts every interrupt, and that decrements the count on each interrupt acknowledge. If the counter always shows a value of zero or one, everything is fine.
- One design rule of thumb will help minimize missing interrupts: re-enable interrupts in ISRs at the earliest safe spot.

Avoid NMI

- Reserve NMI—the non-maskable interrupt—for a catastrophe like the apocalypse. Power-fail, system shutdown, and imminent disaster all good things to monitor with NMI. Timer or UART interrupts are not.
- NMI will break even well-coded interrupt handlers, since most ISRs are non-reentrant during the first few lines of code where the hardware is serviced. NMI will thwart your stack management efforts as well.
- NMI mixes poorly with most tools. Debugging any ISR—NMI or otherwise—is exasperating at best. Few tools do well with single stepping and setting breakpoints inside of the ISR.

Breakpoint Problems

- Though breakpoints are truly wonderful debugging aids, they are like Heisenberg's uncertainty principle: the act of looking at the system changes it. You can cheat Heisenberg—at least in debugging embedded code!—by using real-time trace, a feature available on all emulators and some smart logic analyzers.

Easy ISR Debugging

- What's the fastest way to debug an ISR?
  - Don't.
  - If your ISR is only 10 or 20 lines of code, debug by inspection. Don't fire up all kinds of complex and unpredictable tools.
  - Keep the handler simple and short.

Reentrancy

- In the embedded world a routine must satisfy the following conditions to be reentrant:
  - It uses all shared variables in an atomic way, unless each is allocated to a specific instance of the function.
  - It does not call non-reentrant functions.
  - It does not use the hardware in a non-atomic way.

Atomic Variables

- Both the first and last rules use the word "atomic," which comes from the Greek word meaning "indivisible." In the computer world "atomic" means an operation that cannot be interrupted.
  - mov ax,bx
  - temp=foobar; temp+=1; foobar=temp;
  - foobar+=1;

```
mov ax,[foobar]
inc ax
mov [foobar],ax
```

```
inc [foobar]
```

```
lock inc [foobar]
```

- Rule 2 tells us a calling function inherits the reentrancy problems of the callee.
- Rule 3 is a uniquely embedded caveat. Hardware looks a lot like a variable; if it takes more than a single I/O operation to handle a device, reentrancy problems can develop.

Keeping Code Reentrant

- What are our best options for eliminating non-reentrant code?
  - The first rule of thumb is to avoid shared variables. Globals are the source of no end of debugging woes and failed code. Use automatic variables or dynamically allocated memory.
  - The most common approach is to disable interrupts during non-reentrant code.
  - semaphores

Recursion

- A function is recursive if it calls itself.
- So all recursive functions must be reentrant ... but not all reentrant functions are recursive.

Asynchronous Hardware/Firmware

```
int timer_hi;
interrupt timer(){
        ++timer_hi;}
long timer_read(void){
unsigned int low, high;
low =inword(hardware_register);
high=timer_hi;
return (high<<16+low);}
```

Race Conditions

- One of timer_read's race conditions might be:
  - It reads the hardware and gets, let's say, a value of 0Xffff.
  - Before having a chance to retrieve the high part of the time from variable timer_hi, the hardware increments again to 0x0000.
  - The overflow triggers an interrupt. The ISR runs. timer_hi is now 0
  - 0001, not 0 as it was just nanoseconds before.
  - The ISR returns; our fearless timer_read routine, with no idea an interrupt occurred, blithely concatenates the new 0
  - 0001 with the previously read timer value of 0Xffff, and returns 0X1ffff—a hugely incorrect value.

Options

- The easiest is to stop the timer before attempting to read it.
  - lose time. Turning interrupts off during this period will eliminate unwanted tasking, but increases both system latency and complexity.
- Another solution is to read the timer_hi variable, then the hardware timer, and then reread timer_hi. An interrupt occurred if both variable values aren't identical. Iterate until the two variable reads are equal.
- The upside: correct data, interrupts stay on, and the system doesn't lose counts.
- The downside: in a heavily loaded, multitasking environment, it's possible that the routine could loop for rather a long time before getting two identical reads. The function's execution time is non-deterministic. We've gone from a very simple timer reader to somewhat more complex code that could run for milliseconds instead of microseconds.

- Another alternative might be to simply disable interrupts around the reads.

```
long timer_read(void){
        unsigned int low, high;
        push_interrupt_state;
       disable_interrupts;
        low=inword(Timer_register);
       high=timer_hi;
       if(inword(timer_overflow))
       {       ++high;
               low=inword(timer_register);}
       pop_interrupt_state;
       return (((ulong)high)<<16+(ulong)low);
}
```

ARM interrupts

- ARM7 supports two types of interrupts:
  - Fast interrupt requests (FIQs).
  - Interrupt requests (IRQs).
- Interrupt table starts at location 0.

ARM interrupt procedure

- CPU actions:
  - Save PC. Copy CPSR to SPSR.
  - Force bits in CPSR to record interrupt.
  - Force PC to vector.
- Handler responsibilities:
  - Restore proper PC.
  - Restore CPSR from SPSR.
  - Clear interrupt disable flags.

ARM interrupt latency

- Worst-case latency to respond to interrupt is 27 cycles:
  - Two cycles to synchronize external request.
  - Up to 20 cycles to complete current instruction.
  - Three cycles for data abort.
  - Two cycles to enter interrupt handling state.

Agenda

- Input and output
- Busses
- Memory Architectures

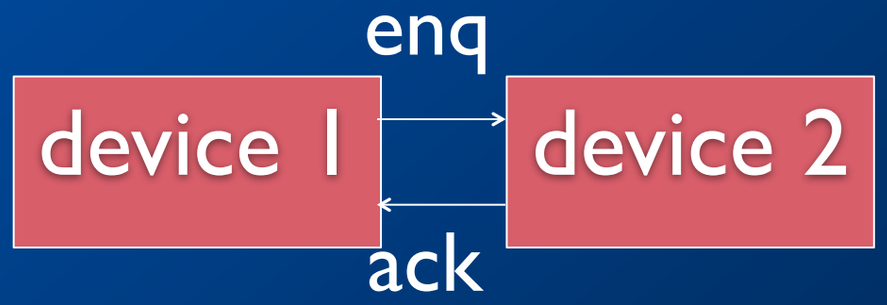
The CPU bus

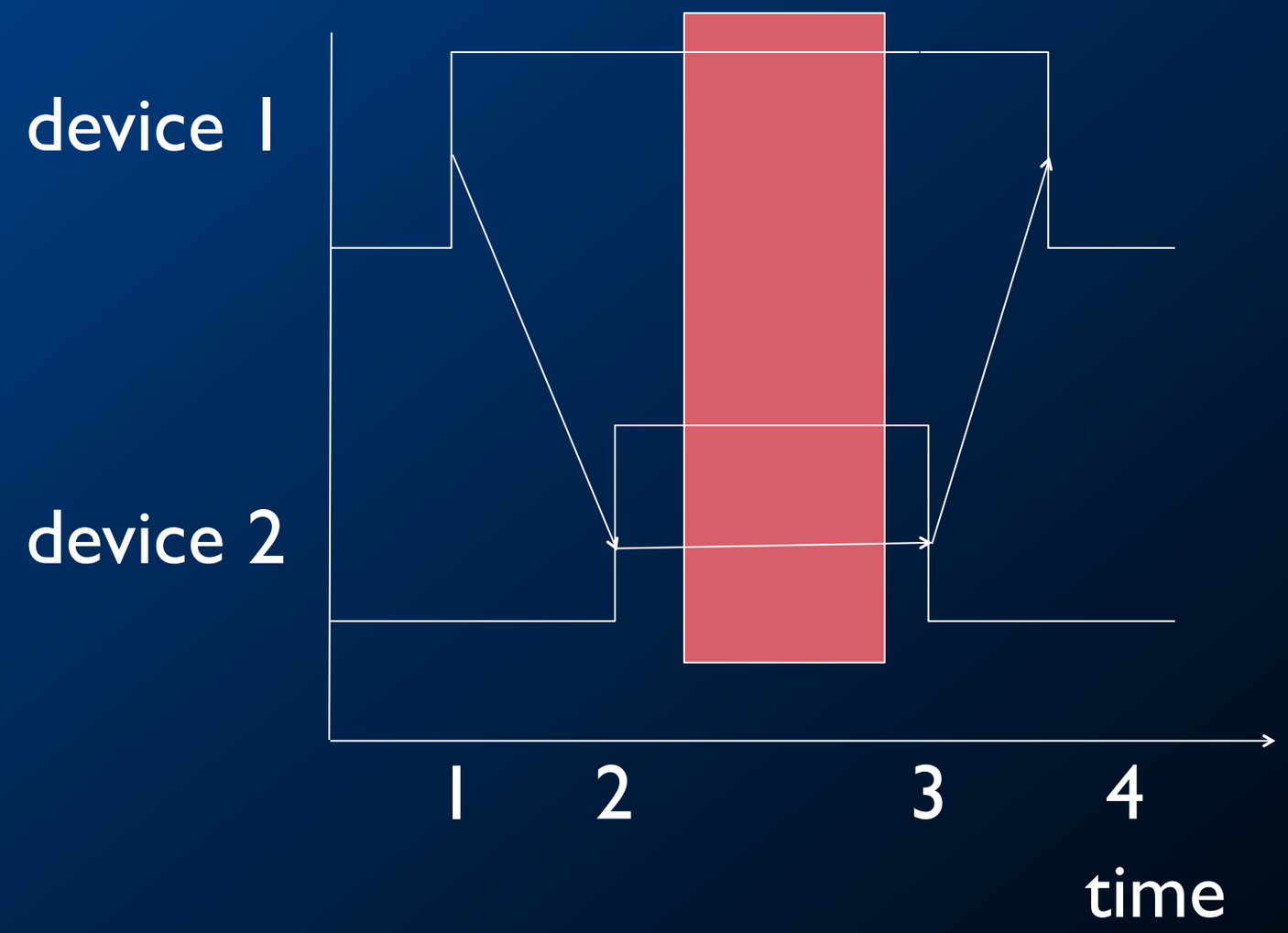
- Bus allows CPU, memory, devices to communicate.
  - Shared communication medium.
- A bus is:
  - A set of wires.
  - A communications protocol.

Bus protocols

- Bus protocol determines how devices communicate.
- Devices on the bus go through sequences of states.
    - Protocols are specified by state machines, one state machine per actor in the protocol.
- May contain asynchronous logic behavior.

Four-cycle handshake



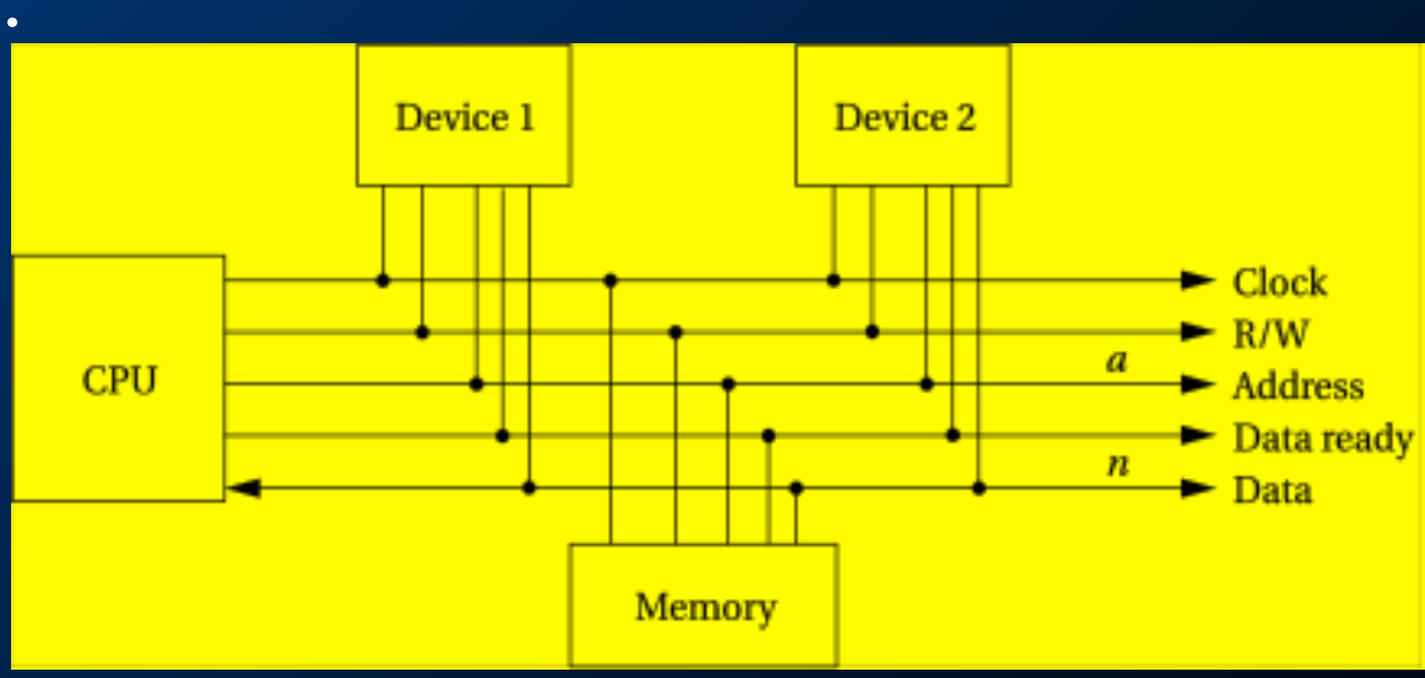


Four-cycle handshake, cont'd.

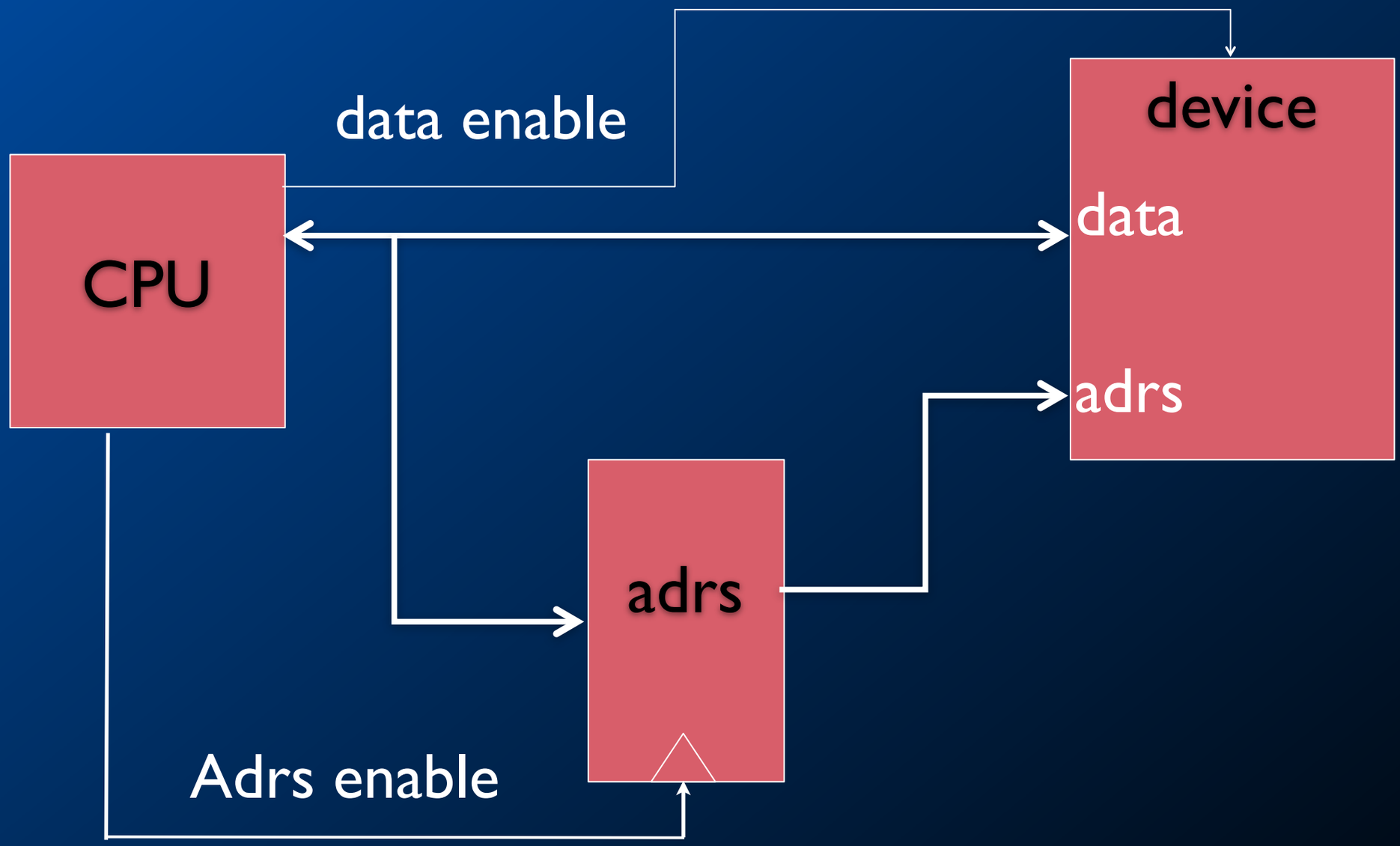
1. Device 1 raises enq.
2. Device 2 responds with ack.
3. Device 2 lowers ack once it has finished.
4. Device 1 lowers enq.

Microprocessor busses

- Clock provides synchronization.
- R/W is true when reading (R/W' is false when reading).
- Address is a-bit bundle of address lines.
- Data is n-bit bundle of data lines.
- Data ready signals when n-bit data is ready.

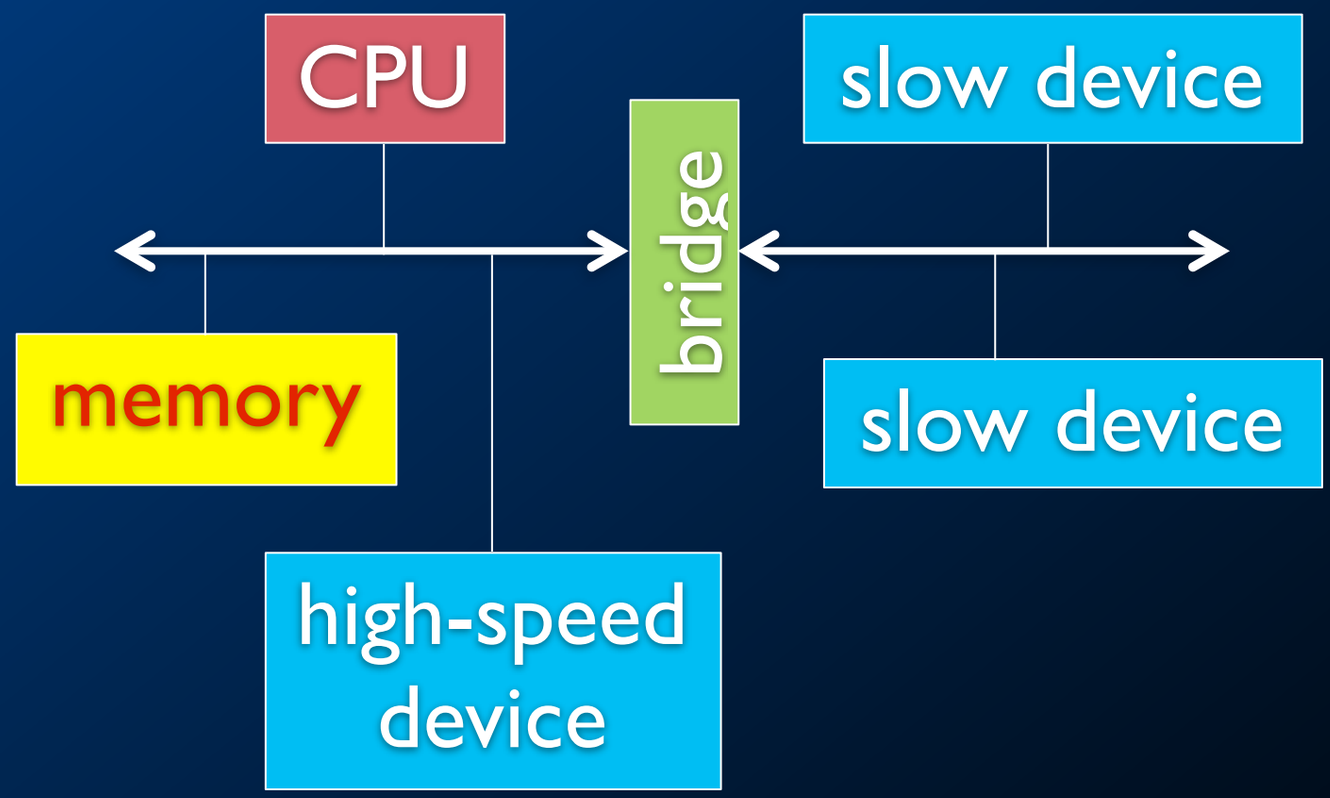


Bus multiplexing

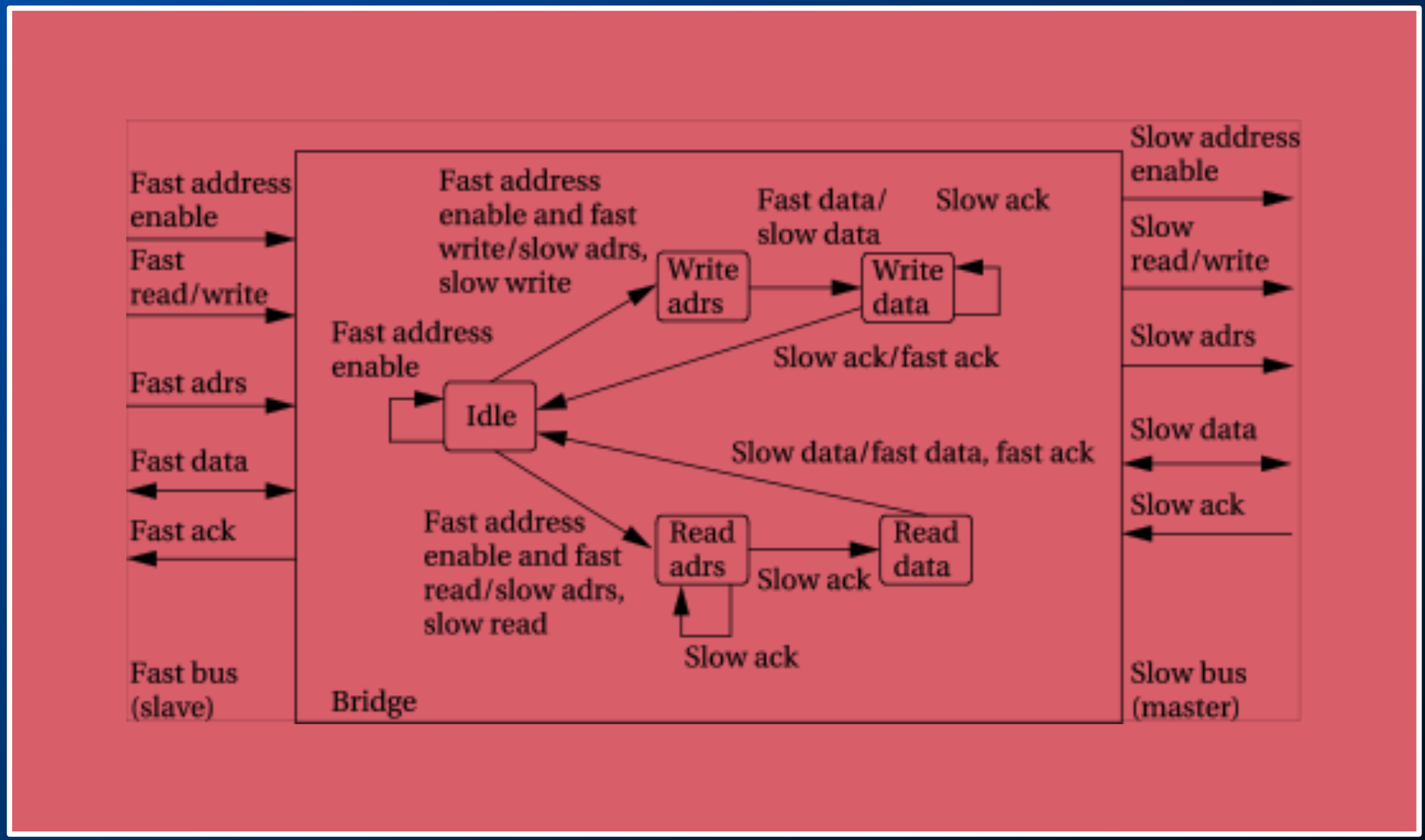


System bus configurations

- Multiple busses allow parallelism:
  - Slow devices on one bus.
  - Fast devices on separate bus.
- A bridge connects two busses.

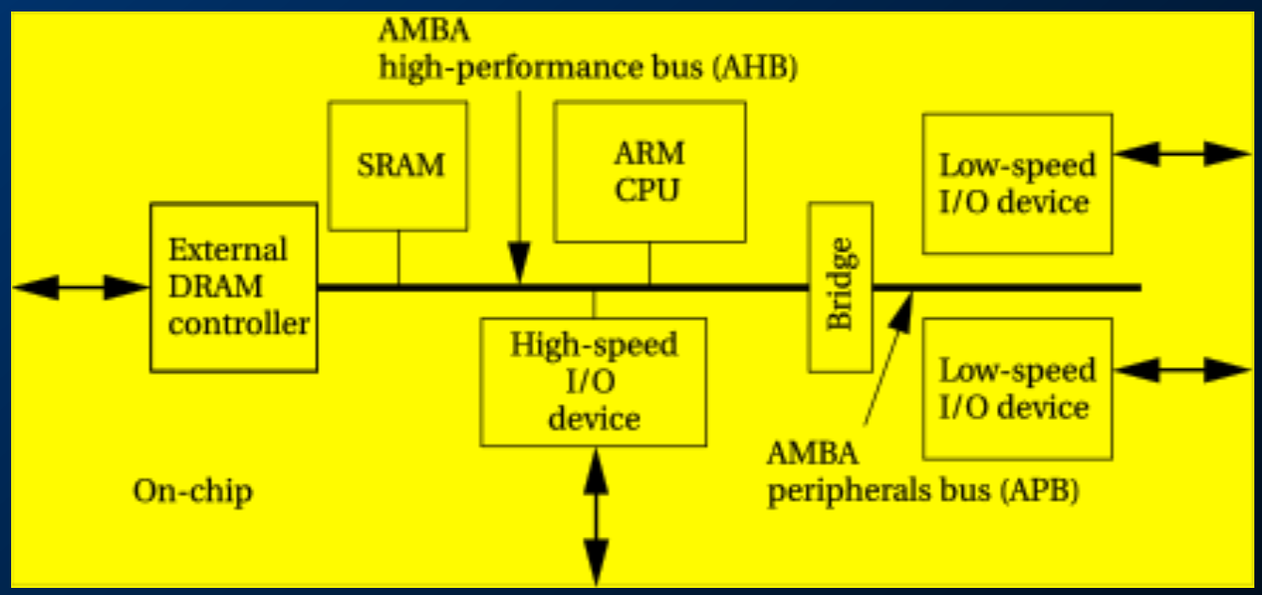


Bridge state diagram



ARM AMBA bus

- Two varieties:
  - AHB is high-performance.
  - APB is lower-speed, lower cost.
- AHB supports pipelining, burst transfers, split transactions, multiple bus masters.
- All devices are slaves on APB.

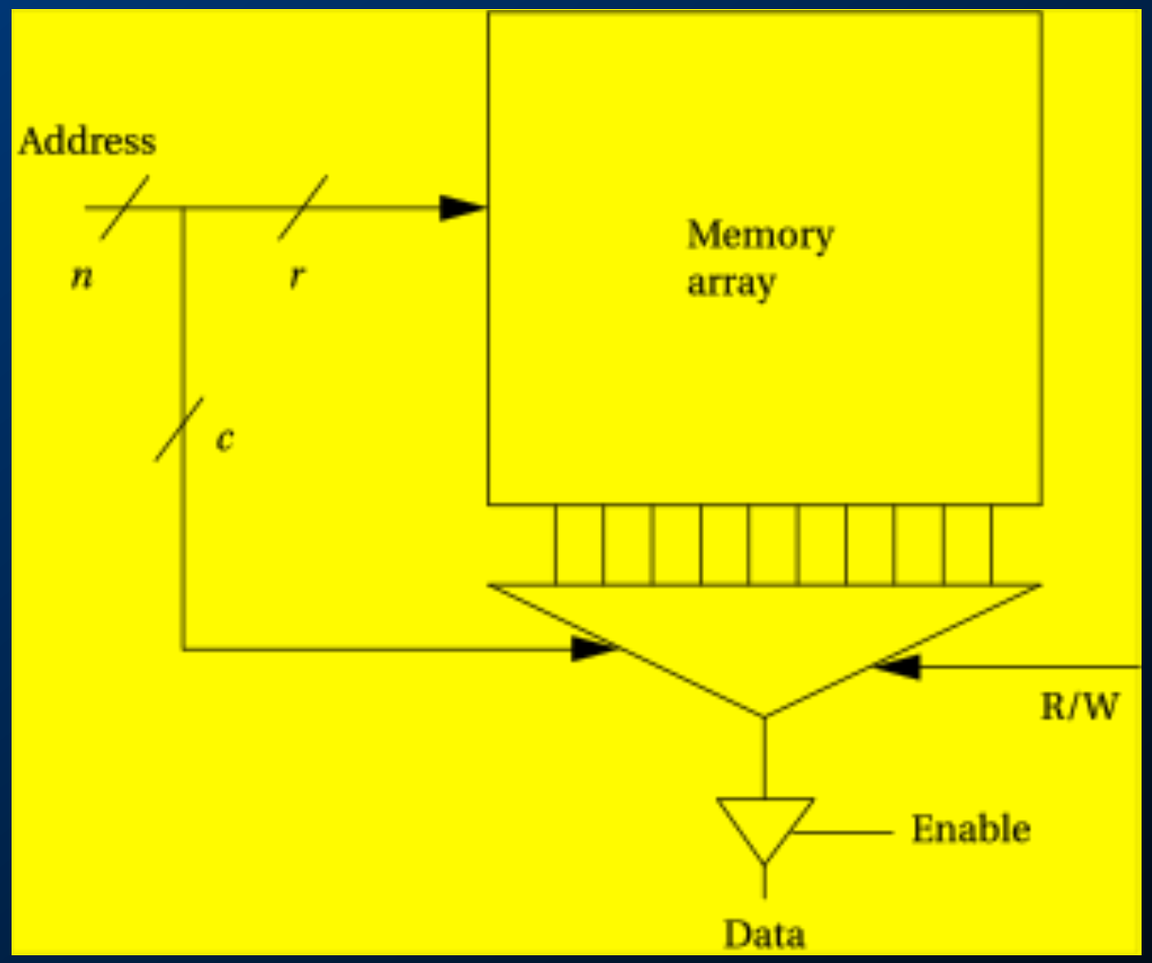


Agenda

- Input and output
- Busses
- Memory Architectures

Memory components

- Several different types of memory:
  - DRAM.
  - SRAM.
  - Flash.
- Each type of memory comes in varying:
  - Capacities.
  - Widths.



Random-access memory

- Dynamic RAM is dense, requires refresh.
  - Synchronous DRAM is dominant type.
  - SDRAM uses clock to improve performance, pipeline memory accesses.
- Static RAM is faster, less dense, consumes more power.

Read-only memory

- ROM may be programmed at factory.
- Flash is dominant form of field-programmable ROM.
  - Electrically erasable, must be block erased.
    - Random access, but write/erase is much slower than read.
    - NOR flash is more flexible.
    - NAND flash is more dense.

Flash memory

- Non-volatile memory.
  - Flash can be programmed in-circuit.
- Random access for read.
- To write:
  - Erase a block to 1.
  - Write bits to 0.

Flash writing

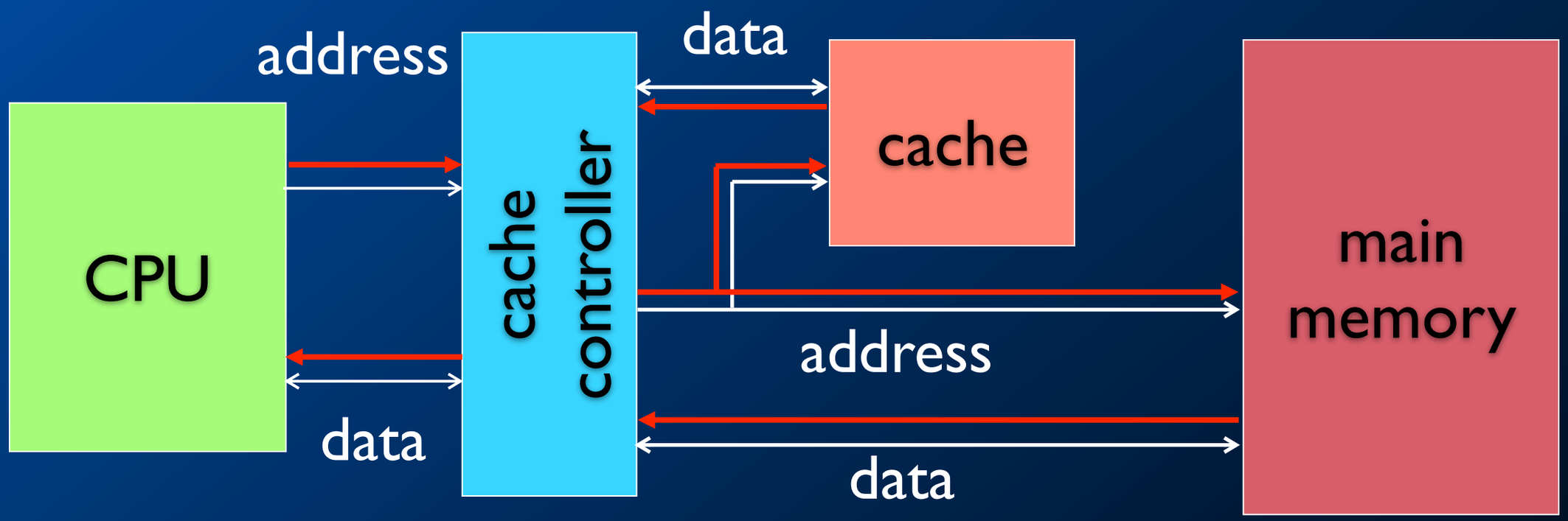
- Write is much slower than read.
  - 1.6 µs write, 70 ns read.
- Blocks are large (approx. 1 Mb).
- Writing causes wear that eventually destroys the device.
  - Modern lifetime approx. 1 million writes.

Types of flash

- NOR:
  - Word-accessible read.
  - Erase by blocks.
- NAND:
  - Read by pages (512-4K bytes).
  - Erase by blocks.
- NAND is cheaper, has faster erase, sequential access times.

	NOR	NAND
写入/擦除一个块的操作时间	1～5s	2～4ms
读性能	1200～1500KB	600～800KB
写性能	<80KB	200～400KB
接口/总线	SRAM接口/独立的地址数据总线	8位地址/数据/控制总线，I/O接口复杂
读取模式	随机读取	串行地存取数据
成本	较高	较低，单元尺寸约为NOR的一半，生产过程简单，同样大小的芯片可以做更大的容量
容量及应用场合	1～64MB，主要用于存储代码	8MB～4GB，主要用于存储数据
擦写次数(耐用性)	约10万次	约100万次
位交换(bit位反转)	少	较多，关键性数据需要错误探测/错误更正(EDC/ECC)算法
坏块处理	无，因为坏块故障率少	随机分布，无法修正

Caches and CPUs



Cache operation

- Many main memory locations are mapped onto one cache entry.
- May have caches for:
  - instructions;
  - data;
  - data + instructions (unified).
- Memory access time is no longer deterministic.

Terms

- Cache hit: required location is in cache.
- Cache miss: required location is not in cache.
- Working set: set of locations used by program in a time interval.

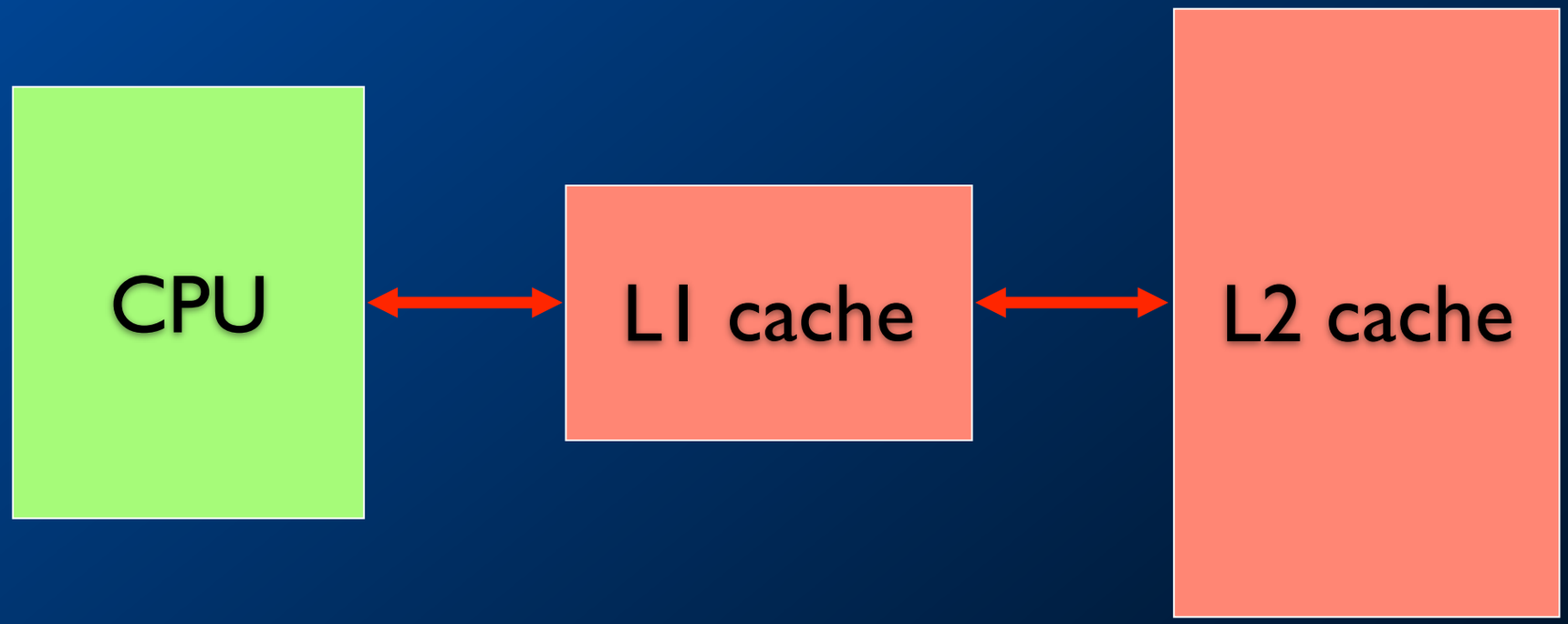
Types of misses

- Compulsory (cold): location has never been accessed.
- Capacity: working set is too large.
- Conflict: multiple locations in working set map to same cache entry.

Memory system performance

- h = cache hit rate.
- t_{cache} = cache access time, t_{main} = main memory access time.
- Average memory access time:
- $t_{av} = ht_{cache} + (1-h)t_{main}$

Multiple levels of cache



Multi-level cache access time

- h_1 = cache hit rate.
- h_2 = hit rate on L2.
- Average memory access time:
- $t_{av} = h_1 t_{L1} + h_2 t_{L2} + (1 - h_2 - h_1) t_{main}$

Replacement policies

- Replacement policy: strategy for choosing which cache entry to throw out to make room for a new memory location.
- Two popular strategies:
  - Random.
  - Least-recently used (LRU).

Cache organizations

- Fully-associative: any memory location can be stored anywhere in the cache (almost never implemented).
- Direct-mapped: each memory location maps onto exactly one cache entry.
- N-way set-associative: each memory location can go into one of n sets.

Cache performance benefits

- Keep frequently-accessed locations in fast cache.
- Cache retrieves more than one word at a time.
  - Sequential accesses are faster after first access.

Memory management units

- Memory management unit (MMU) translates addresses:

Memory management tasks

- Allows programs to move in physical memory during execution.
- Allows virtual memory:
  - memory images kept in secondary storage;
    - images returned to main memory on demand during execution.
- Page fault: request for location not resident in memory.

Thank you!