

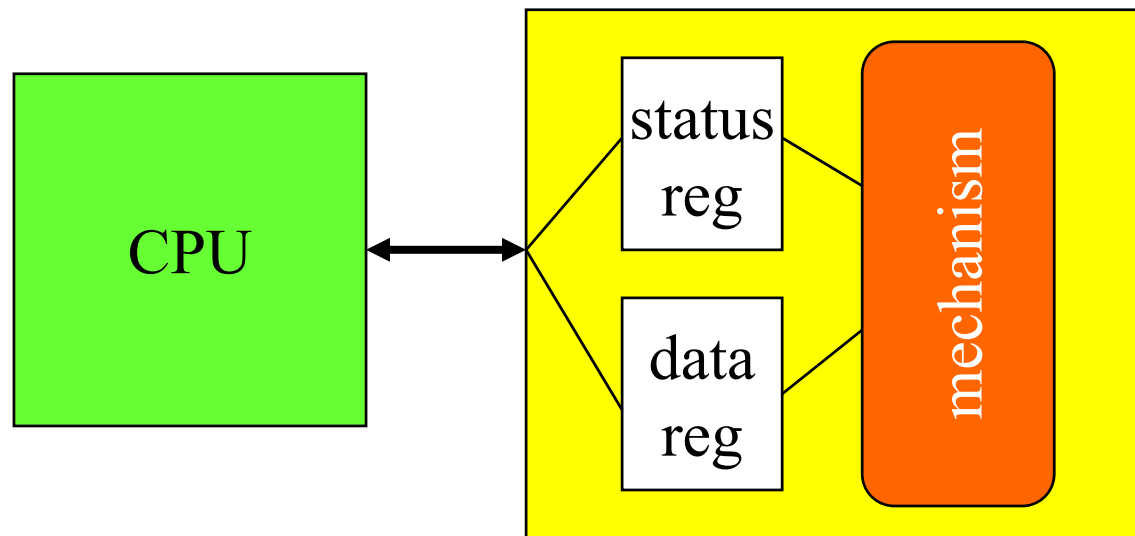
CPUs



- ⌘ Input and output.
- ⌘ Supervisor mode, exceptions, traps.
- ⌘ Co-processors.
- ⌘ Caches.
- ⌘ Memory management.
- ⌘ CPU performance
- ⌘ CPU power consumption.
- ⌘ Example: data compressor

I/O devices

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



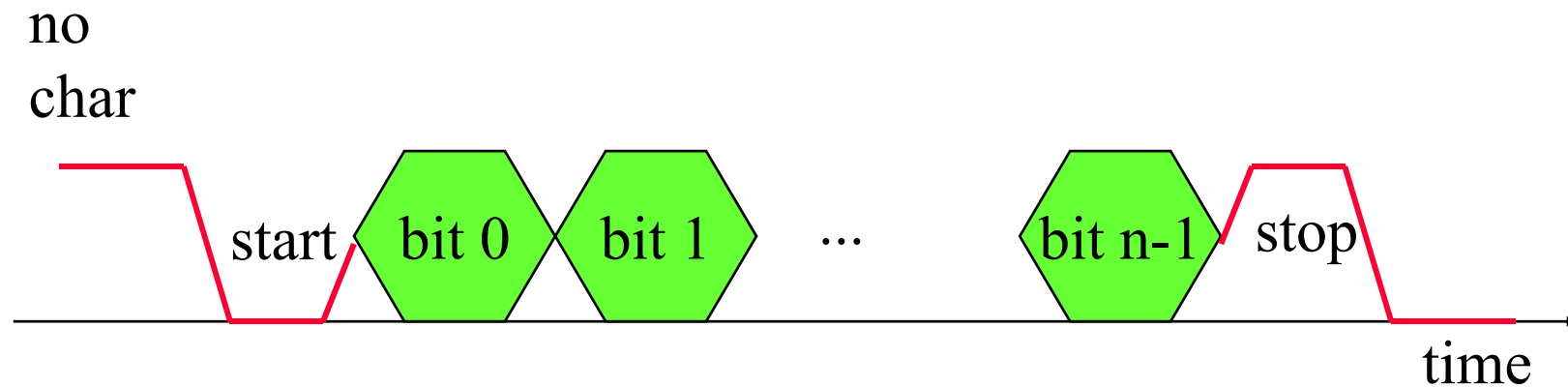
Application: 8251 UART



- ⌘ 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

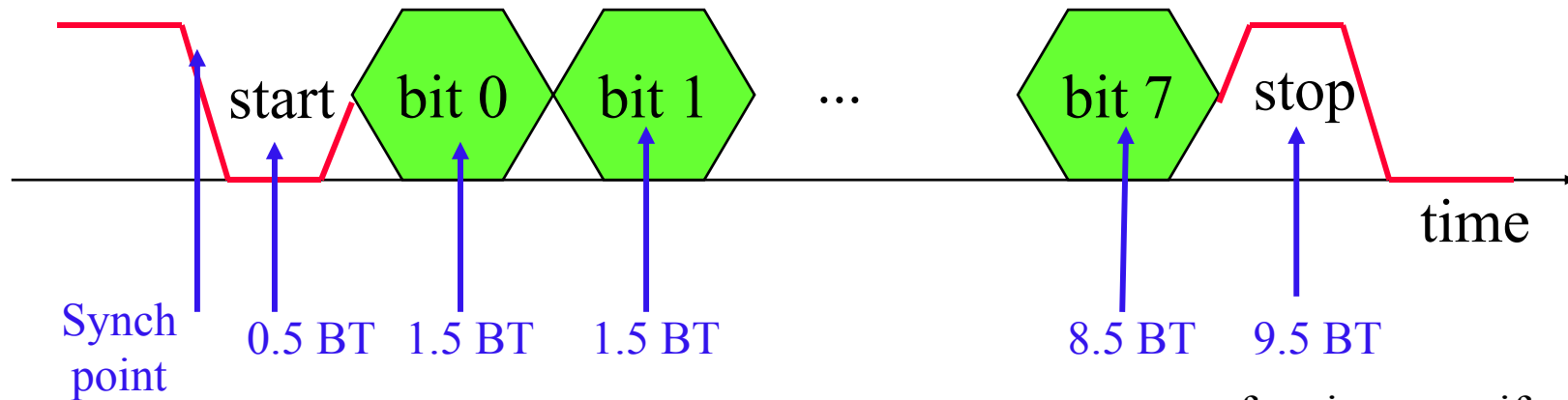


- ⌘ Baud (bit) rate.
- ⌘ Number of bits per **character**.
- ⌘ Parity/no parity.
- ⌘ Even/odd parity.
- ⌘ Length of stop bit (1, 1.5, 2 bits).

Sample points

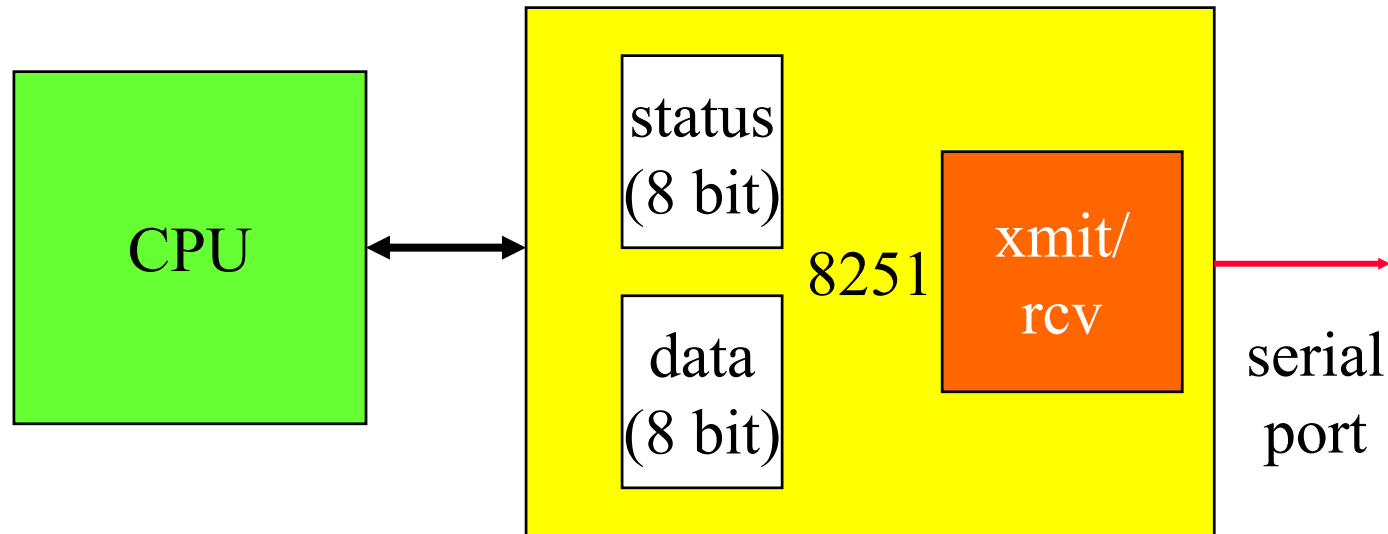
⌘ Example: 10 bits/ character (1 stop bit, 8 bits, no parity bit, 1 stop bit)

no
char



framing error if not 1

8251 CPU interface



8251 interrupts CPU

1. when receiving a character is done
2. when sending a character is finished

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); polling ,busy-wait
    current_char++;
}
```

Simultaneous busy/wait input and output



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

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 a subroutine call to handle device.

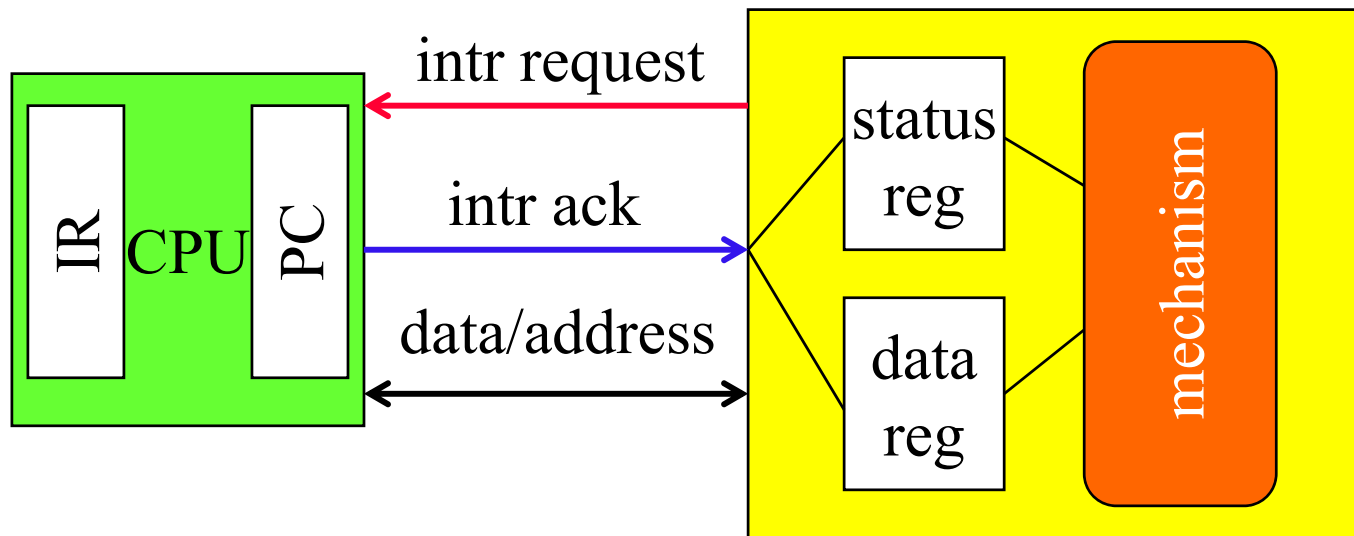
- ☑ Interrupt handler, device driver

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 **later** resume executing **foreground program**.

Interrupt interface



Interrupt physical interface



- ⌘ CPU and device are connected by CPU bus.
- ⌘ CPU and device handshake with interrupt request and acknowledgement:
 - ☑ device asserts interrupt request;
 - ☑ CPU asserts interrupt acknowledge when it can handle the interrupt.
- ⌘ An PIC (programmable interrupt controller) connects multiple external interrupts to one of the two ARM interrupt requests

Character I/O handlers



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

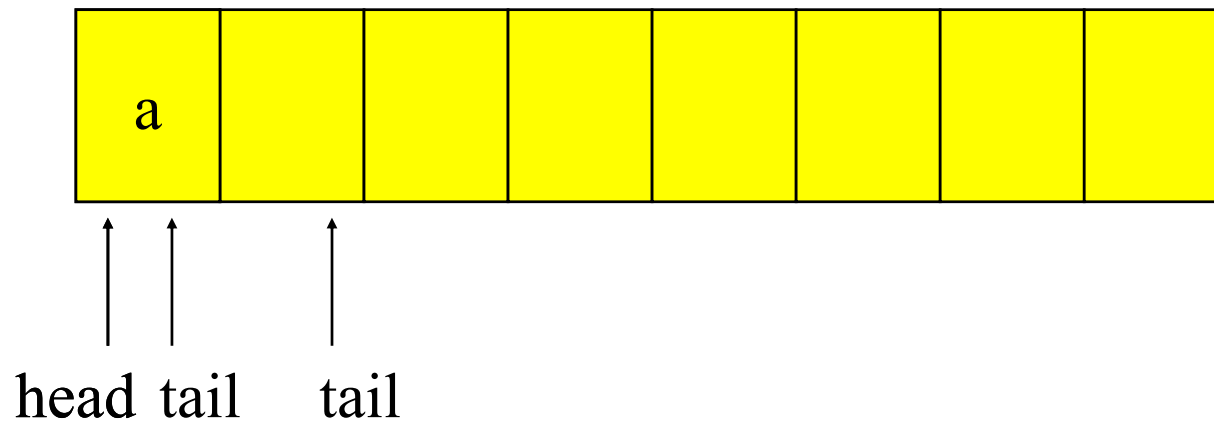
Interrupt-driven main program



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

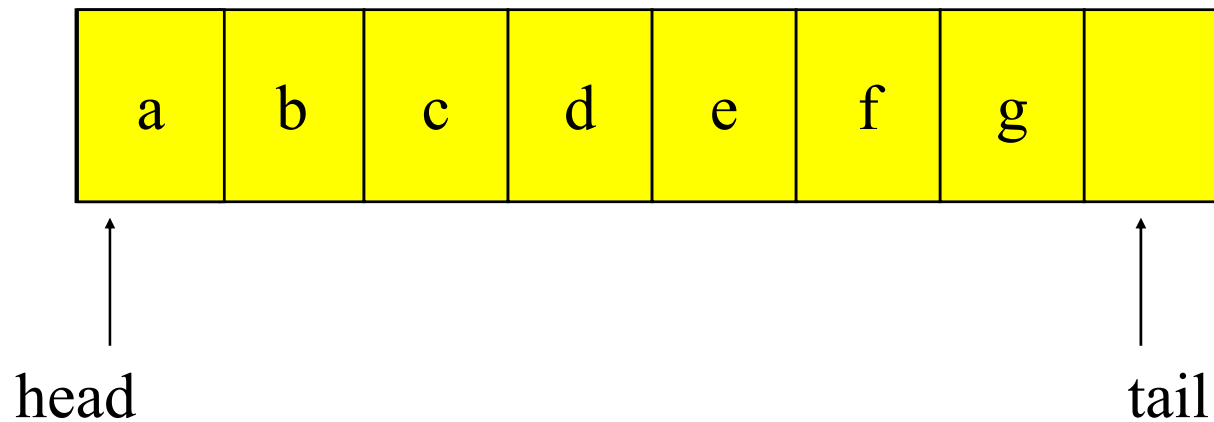
Interrupt I/O with buffers

⌘ Queue for characters:



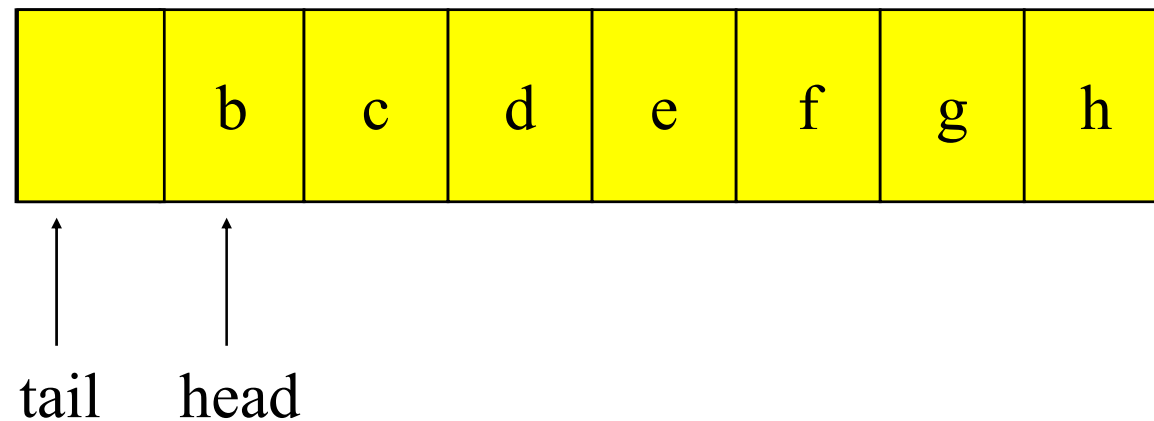
Interrupt I/O with buffers

⌘ Queue for characters:



Interrupt I/O with buffers

⌘ Queue for characters:

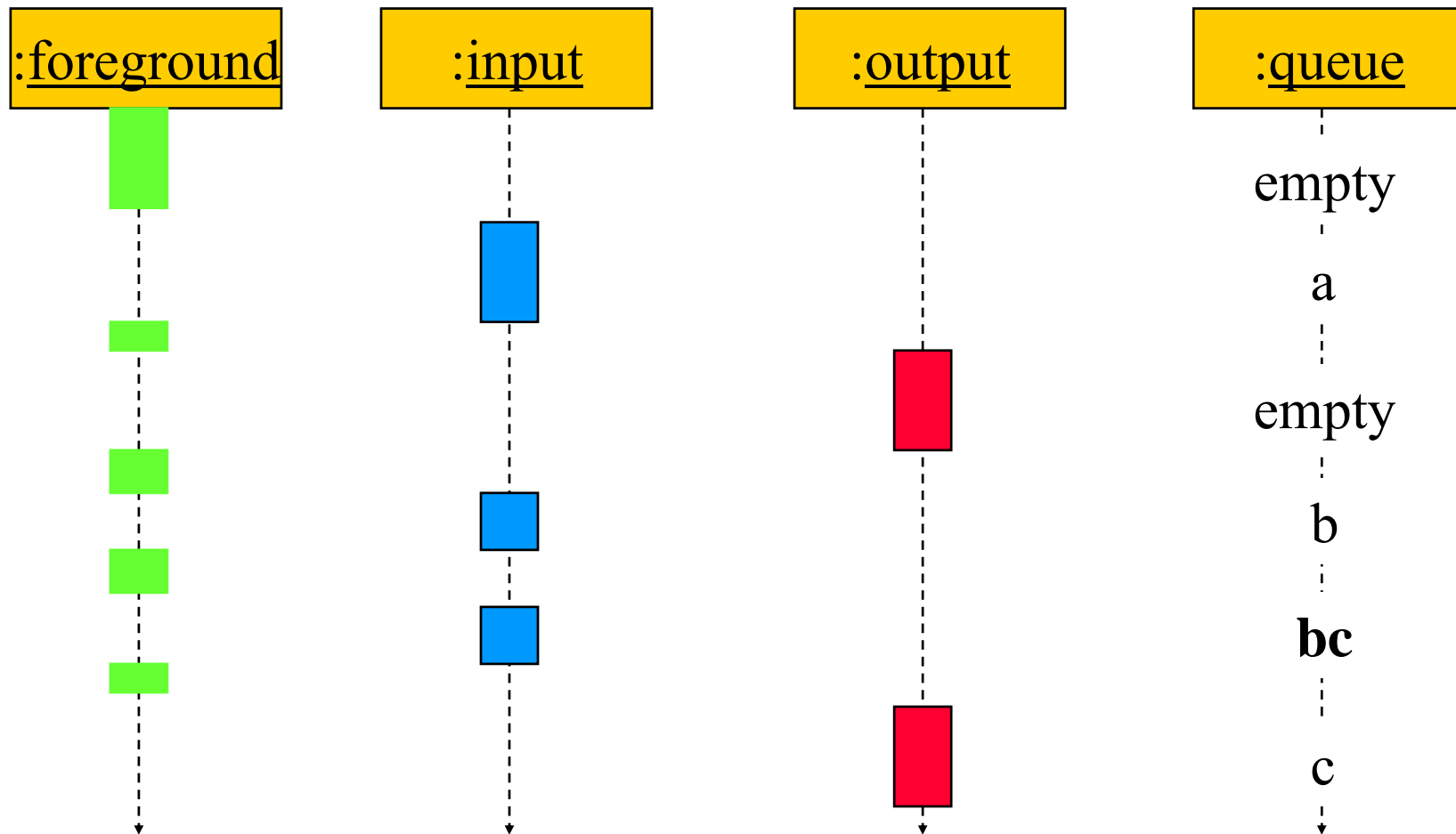


Buffer-based input handler



```
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?

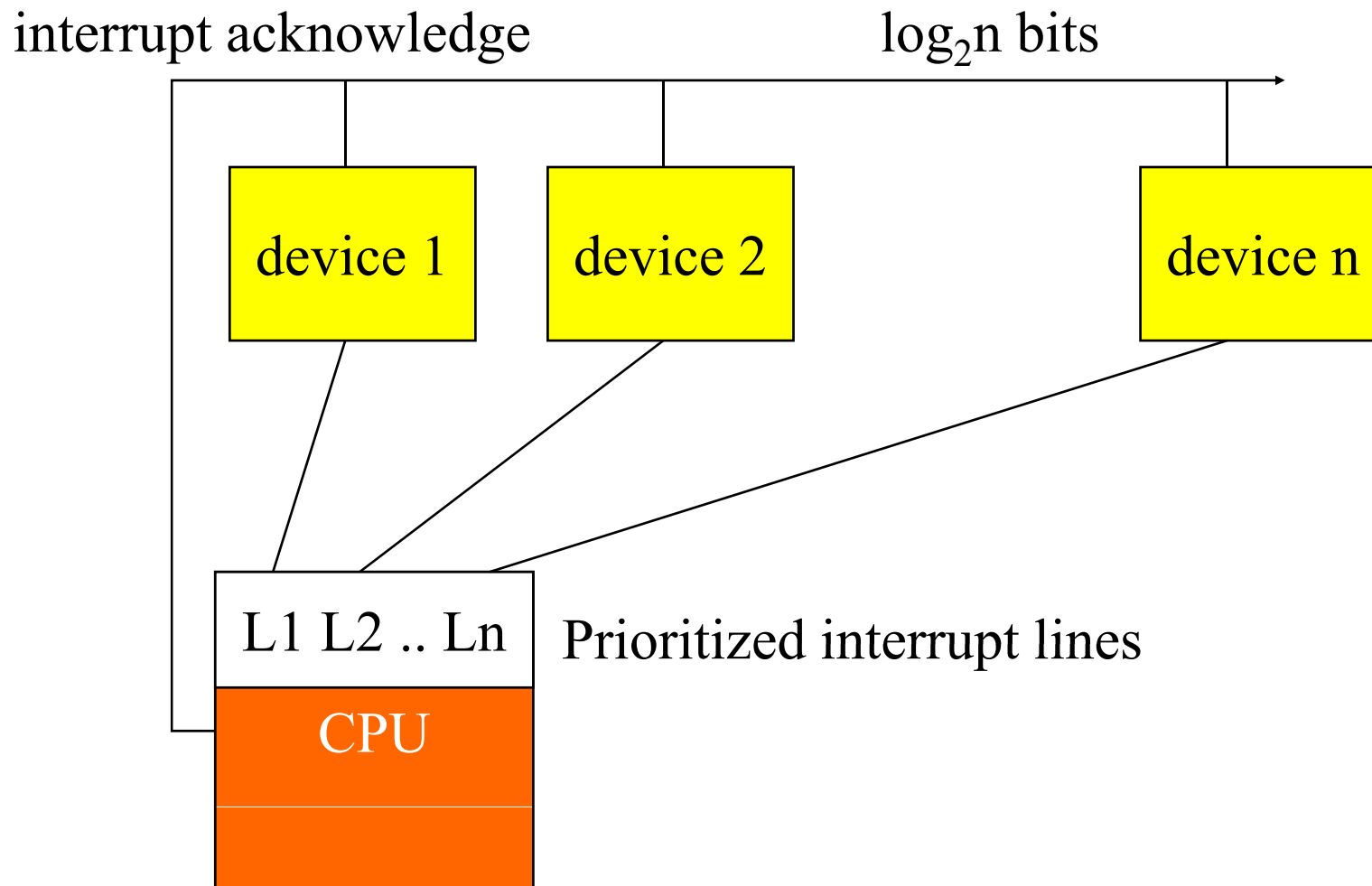
- ☑ Foreground program can exhibit mysterious bugs.
- ☑ Bugs will be hard to repeat---depend on interrupt timing.

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

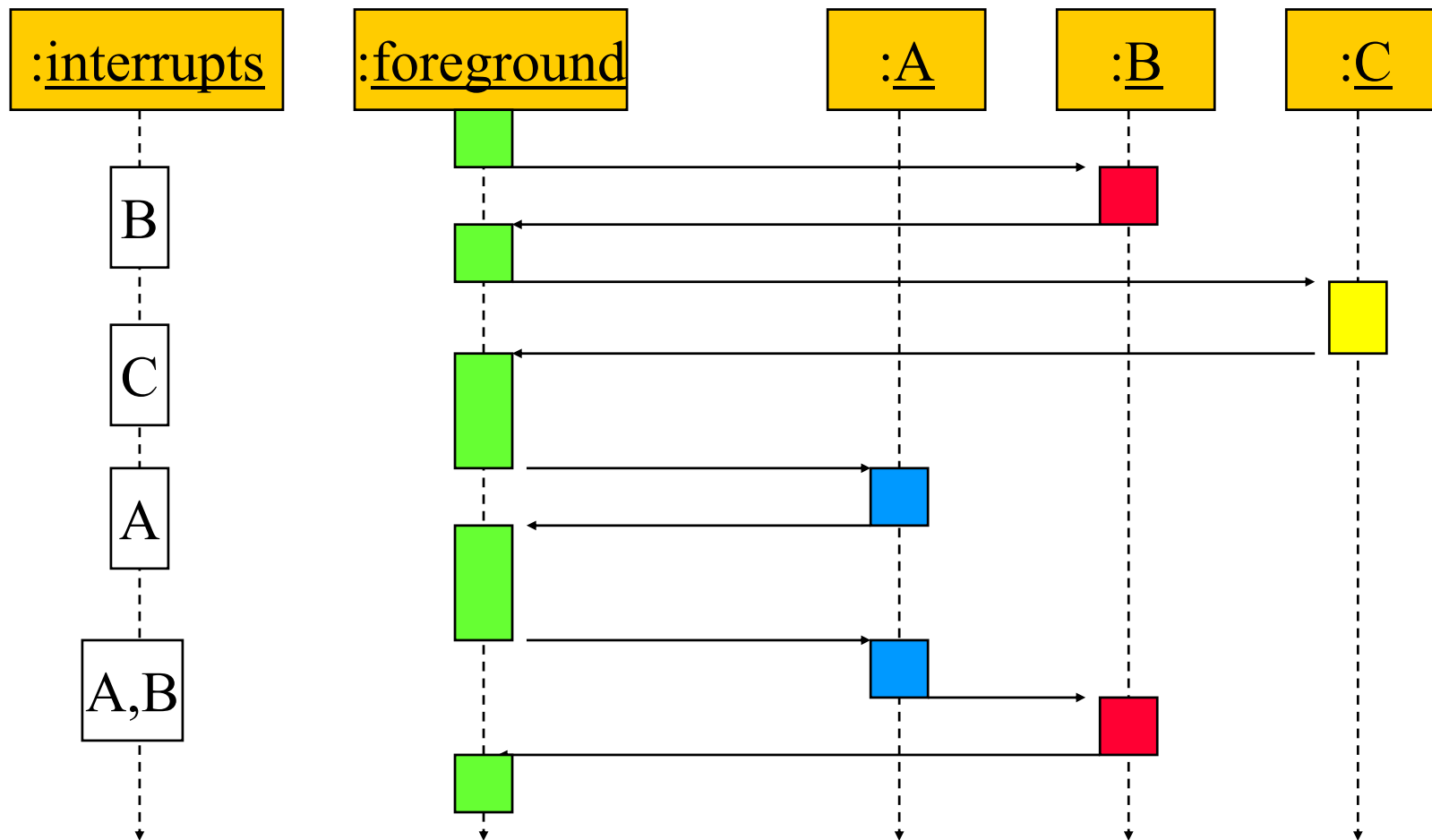


Interrupt prioritization



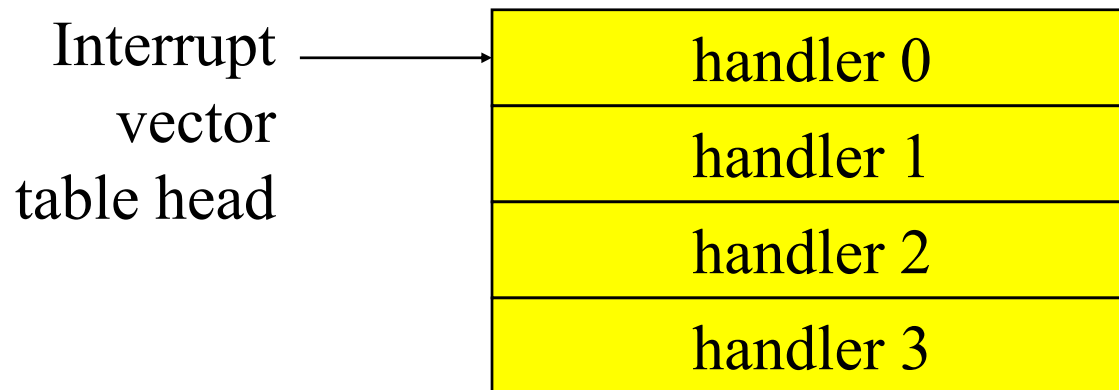
- ⌘ **Masking**: interrupt with priority lower than current priority is not recognized until pending interrupt is complete.
- ⌘ **Non-maskable interrupt (NMI)**: highest-priority, never masked.
 - ☑ Often used for power-down.

Example: Prioritized I/O

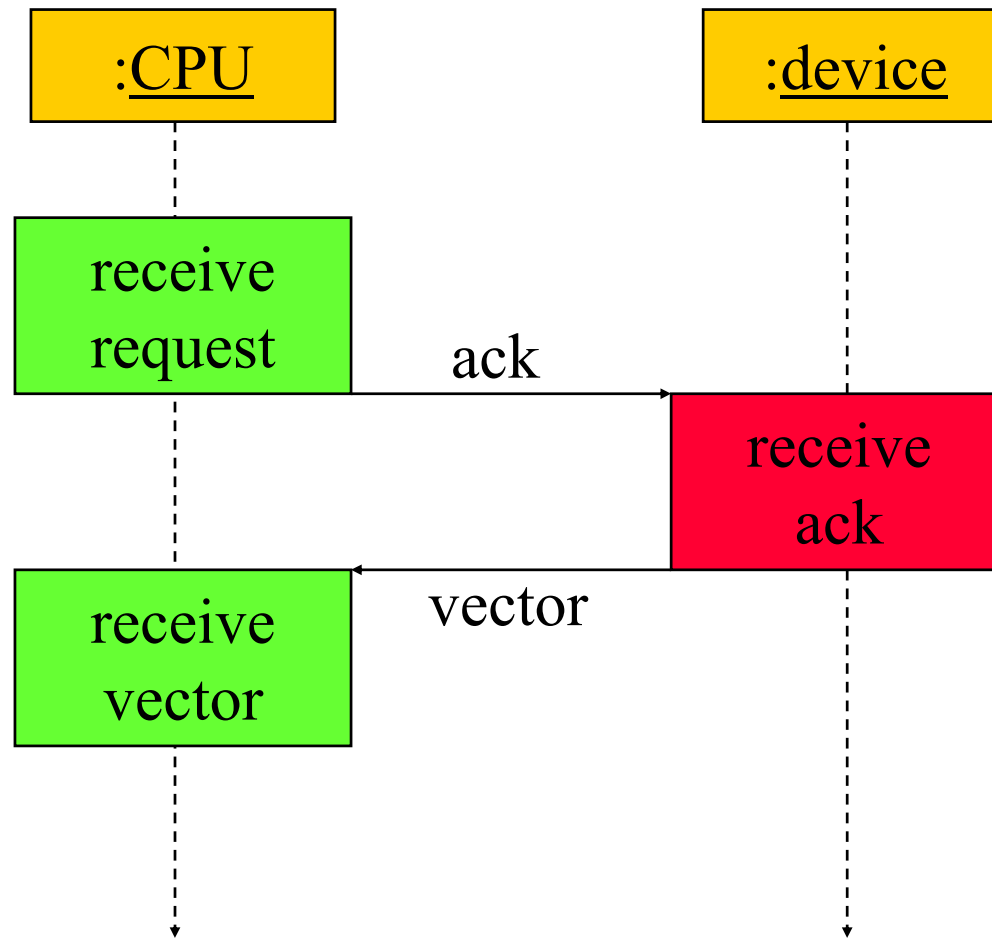


Interrupt vectors

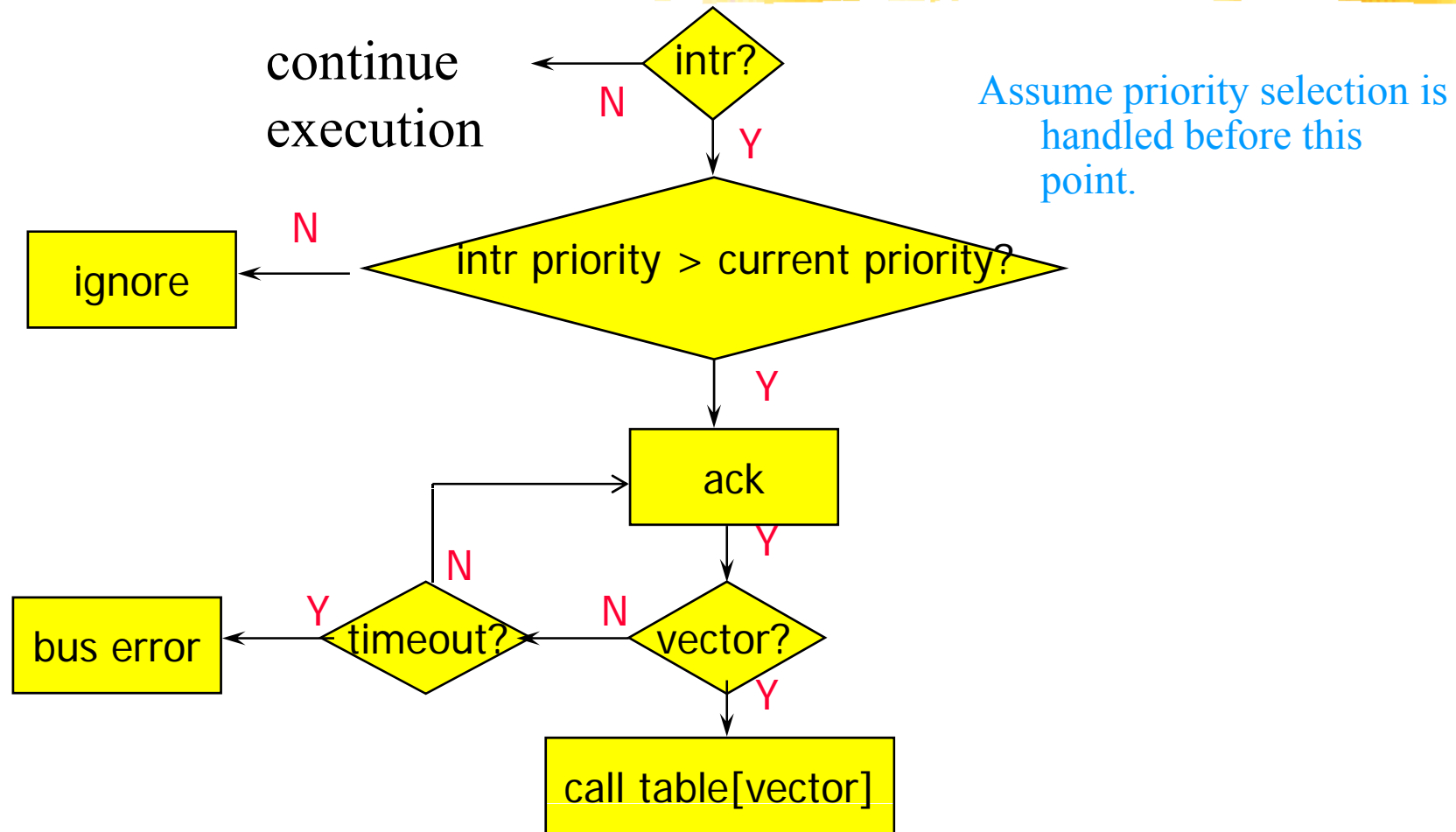
- ⌘ Allow different devices to be handled by different code.
- ⌘ Interrupt vector table:



Interrupt vector acquisition



Generic interrupt mechanism



Interrupt sequence



- ⌘ CPU acknowledges request.
- ⌘ Device sends vector.
- ⌘ CPU calls handler.
- ⌘ Handler 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.

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.
- ☑ Change the processor mode in new CPSR
- ☑ Interrupts (FIQ or IRQ) are disabled
- ☑ Force PC to vector.

ARM interrupt procedure

⌘ Handler :

- ☑ Save context
- ☑ Identifies the external interrupt source and executes the appropriate ISR
- ☑ Reset the interrupt
- ☑ Restore context

⌘ Return form handler

- ☑ Restore CPSR from SPSR
- ☑ interrupt disable flags.
- ☑ $pc = lr - 4$

IRQ interrupt procedure

- ⌘ an IRQ interrupt is raised when the processor is in user mode.
 - ☒ CPSR=nzcvqjift_usr : both IRQ and FIQ are enabled
- ⌘ User mode CPSR is saved into SPSR. Set new CPSR
 - ☒ new CPSR = nzcvqj|ft_irq
 - ☒ SPSR_irq = nzcvqjift_usr
 - ☒ r14_irq=pc
 - ☒ pc= 0x18

Link register offsets

- ⌘ Reset: lr is not defined on a reset
- ⌘ Data abort : $(lr - 8)$ points to the instruction that caused the abort
- ⌘ FIQ, IRQ: $(lr - 4)$ points to address from the handler
- ⌘ Prefetch abort: $(lr - 4)$ points to the instruction that caused the abort
- ⌘ SWI, Undefined Instruction: lr points to the next instruction after the SWI or undefined instruction

Return from IRQ or FIQ handler

handler

<handler codes>

...

SUBS pc, r14, #4 ; pc=r14 -4

- ⌘ Because there S at the end of the instruction and pc is the destination register, cpsr is automatically restored from spsr.

handler

SUB r14, r14, #4 ; r14 -= 4

...

<handler codes>

...

MOVS pc, r14 ; return

Return from IRQ or FIQ handler

handler

```
SUB          r14, r14, #4          ; r14 -= 4
STMFD       r13!, {r0-r3, r14}    ; store context
...
<handler codes>
...
LDMFD       r13!, {r0-r3, pc}^     ; restore context and return
```

⌘ ^ symbol in the instruction forces cpsr to be restored from spsr.

ARM interrupt latency

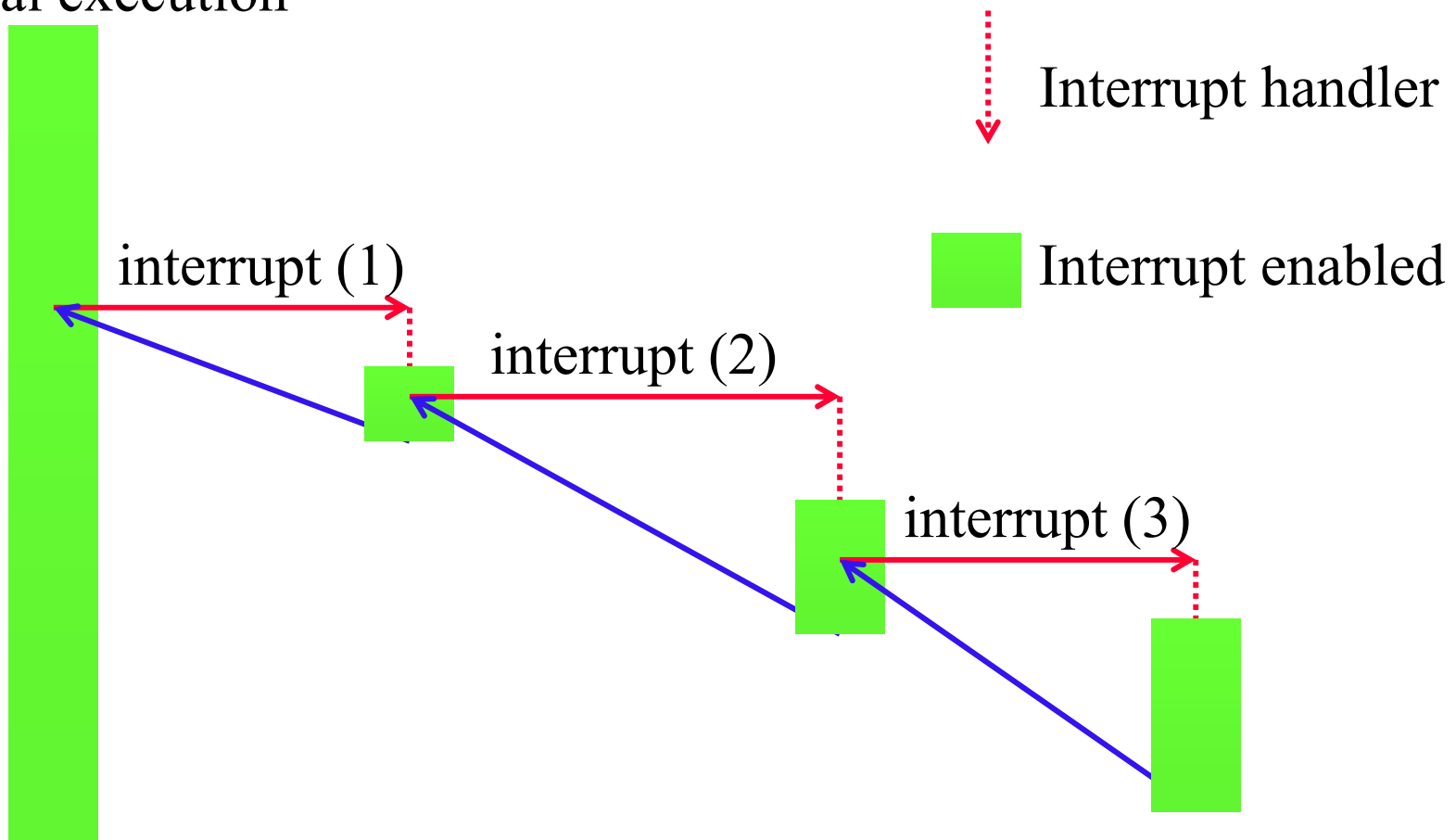


⌘ Worst-case latency to respond to interrupt is 27 cycles:

- ☑ 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.

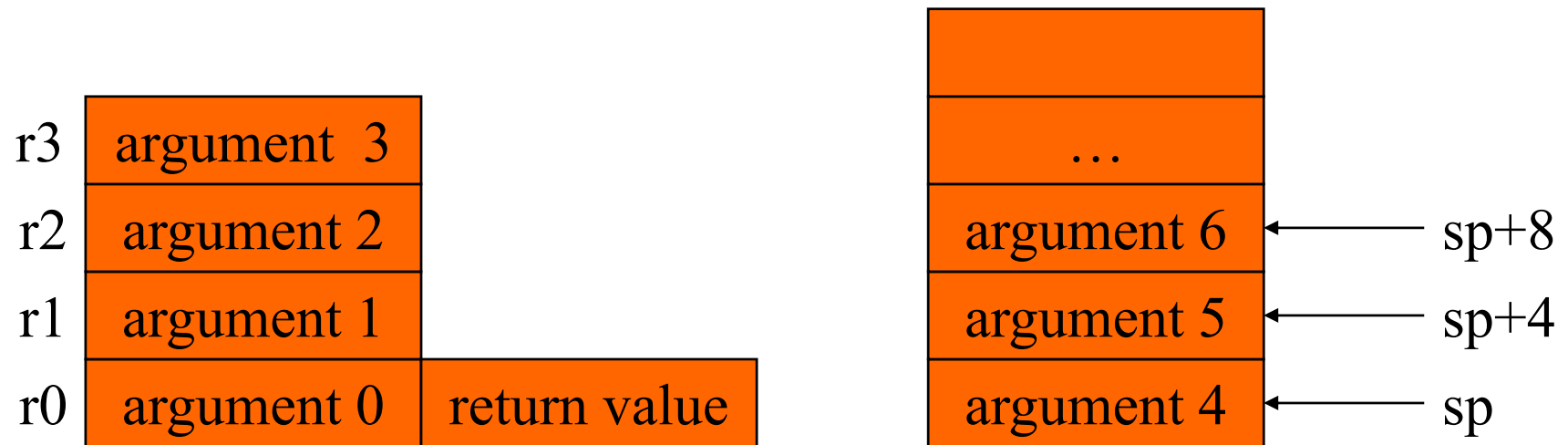
A three-level nested interrupt

Normal execution



ARM-Thumb procedure call standard (ATPCS)

- ⌘ The first four integer arguments are passed in the four ARM registers: r0, r1, r2, r3
- ⌘ Subsequent integer arguments are placed in the FD stack, ascending in memory.
- ⌘ Function return value is passed in r0



ARM procedure call standard

- ⌘ For functions with 4 or more arguments, both the caller and the callee must access the stack for some arguments.
- ⌘ Note that for C++ the first argument to an object is the **this** pointer. This argument is implicit and additional to the explicit arguments.
- ⌘ If a C function needs more than four arguments, or a C++ function more than three explicit arguments, then it is more efficient to use a structure as a grouped arguments and pass a structure pointer.

<LDM|STM> {<cond>} {addressing_mode} {S} Rn{!}, <registers> {^}

Load-store multiple instructions

⌘ Addressing mode

- ☒ IA: increment after: R_n , R_n+4N-4 , R_n+4N
- ☒ IB: increment before: R_n+4 , R_n+4N , R_n+4N
- ☒ DA: decrement after: R_n , R_n-4N+4 , R_n-4N
- ☒ DB: decrement before: R_n-4 , R_n-4N , R_n-4N

⌘ Load-store multiple pairs when base update is used

- ☒ STMIA – LDMDB
- ☒ STMIB – LMDA
- ☒ STMDA – LDMIB
- ☒ STMDB – LDMIA

STM--LDM pair

⌘ Nesting/recursion requires coding convention:

```
; pre
STMIB r0!, {r1-r3}
MOV r1, #1
MOV r2, #2
MOV r3, #3
;mid
LDMDA r0!, {r1-r3}
;post
```

pre	r0=0x00009000	mid	r0=0x0000900c	post	r0=0x00009000
	r1=0x00000009		r1=0x00000001		r1=0x00000009
	r2=0x00000008		r2=0x00000002		r2=0x00000008
	r3=0x00000007		r3=0x00000003		r3=0x00000007

Stack operations

⌘ (pop,push) for each addressing mode

- ☒ Full ascending: (LDMFA, STMFA) = (LMDDA, STMIB)
- ☒ Full descending: (LDMFD, STMFD) = (LMDIA, STMDB)
- ☒ Empty ascending: (LDMEA, STMEA) = (LMDDDB, STMIA)
- ☒ Empty descending: (LDMED, STMED) = (LMDIB, STMDA)

Nested subroutine calls

```
    BL    SUB1
    ..
SUB1   STMFD r13!,{r0-r2,r14}           ; save work & link
register
    BL    SUB2
    ..
    LDMFD r13!, {r0-r2,pc}           ; restore work regs & link
SUB2   ..
    MOV   pc, r14                     ; return
```


SWI exception



⌘ Only a branch to the SWI handler is at x08.

SWI_handler

```
    STMFD    sp!, {r0-r12,r14}    ; save context
    LDR      r10, [r14,#-4]       ; load SWI instruction
    BIC      r10, r10, #0xff000000 ; mask off the MSB 8 bits
    MOV      r1,r13              ; copy SVC stack to r1
    BL       swi_jumtable        ; branch to swi_jumtable
    LDMFD    r13!, {r0-r12, pc} ^ ; restore context and return
```

swi_jumtable

```
    MOV      r0,r10              ; mov SWI number to r0
    B        eventSWIhandler     ; branch to SWI handler
```

ARM supervisor mode



⌘ Use SWI instruction to enter supervisor mode, similar to subroutine:

SWI CODE_1

⌘ Actions of SWI

☑ Save the address of the next instruction to r14_svc

☑ Save CPSR in SPSR

☑ Set CPSR

☒ Enter supervisor mode: CPSR[4:0]=10011

☒ disable IRQ: CPSR[7]=1.

☑ Set PC=x08

⌘ 24-bit argument to SWI is passed to supervisor mode code.

Exception



- ⌘ **Exception**: internally detected error.
- ⌘ Exceptions are synchronous with instructions but unpredictable.
- ⌘ Build exception mechanism on top of interrupt mechanism.
- ⌘ Exceptions are usually prioritized and vectorized.

Trap



⌘ **Trap (software interrupt)**: an exception generated by an instruction.

☑ Call supervisor mode.

⌘ ARM uses SWI instruction for traps.

⌘ SHARC offers three levels of software interrupts.

☑ Called by setting bits in IRPTL register.

Stack design



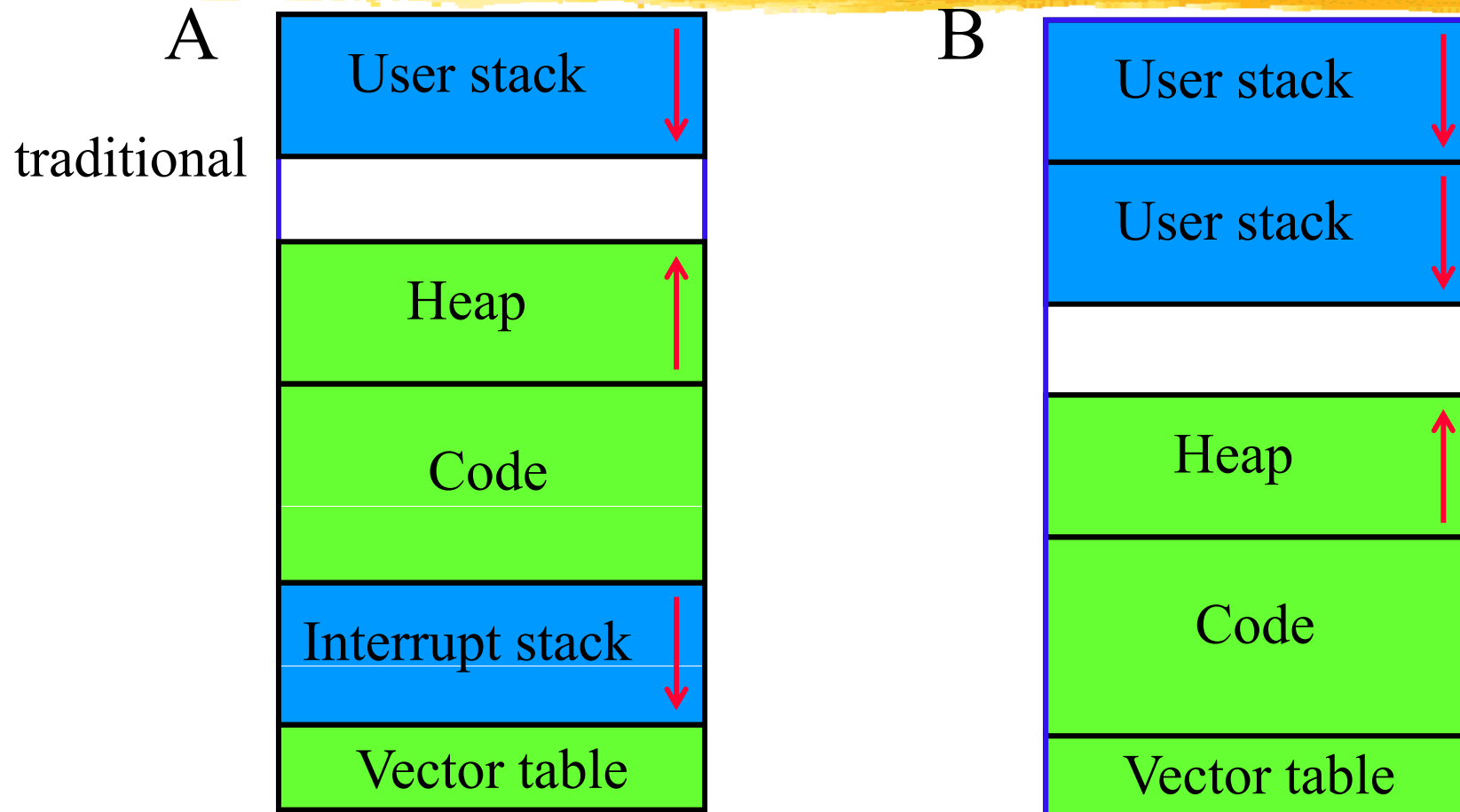
⌘ Depends on

- ☑ OS requirement for stack design
- ☑ Target HW provides a physical limit to the size and positioning of the stack memory.

⌘ ARM-based system : stack grow downward with top of the stack at a high memory address

⌘ Stack overflow must be avoided

Typical memory layouts



Layout B does not corrupt the vector table when a stack overflow occurs

Co-processor



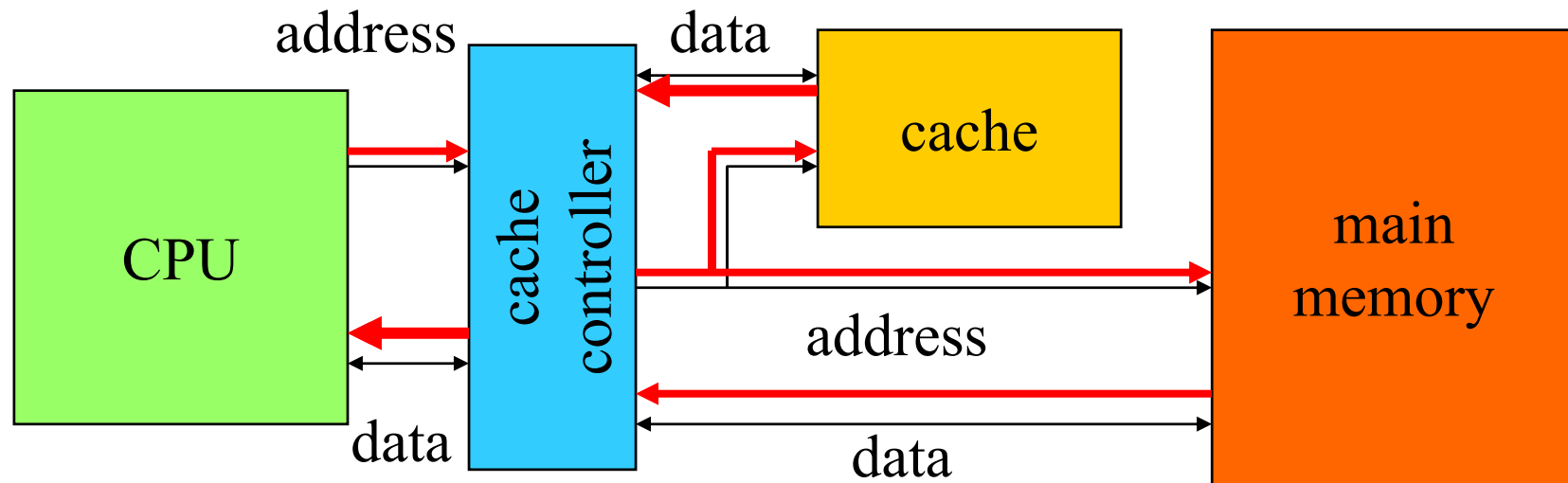
⌘ **Co-processor**: added function unit that is called by instruction.

☑ Floating-point units are often structured as co-processors.

⌘ ARM allows up to 16 designer-selected co-processors.

☑ Floating-point co-processor uses units 1, 2.

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



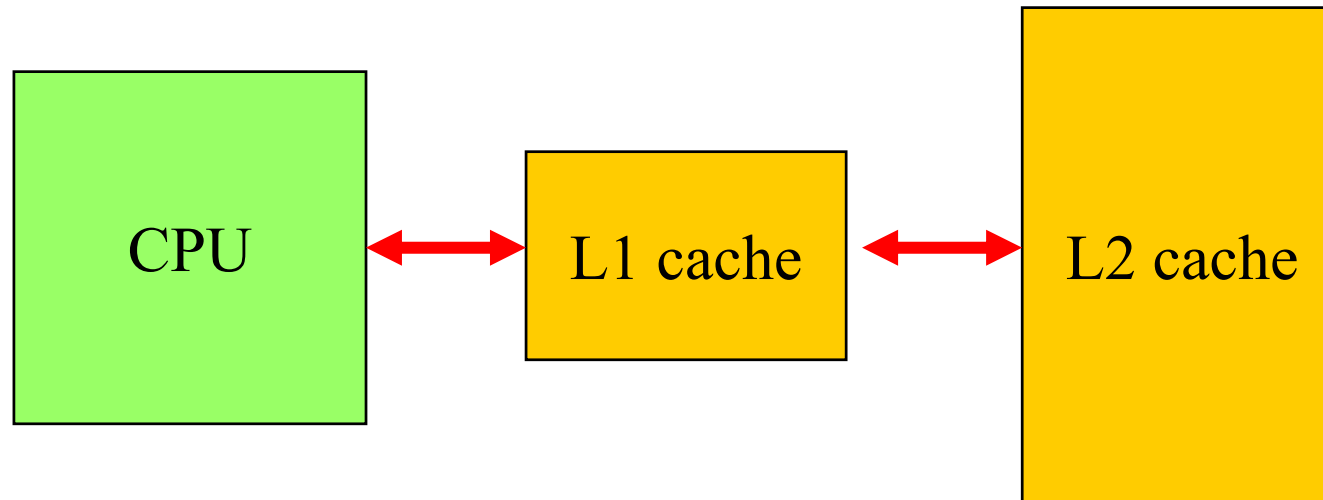
⌘ h = cache hit rate.

⌘ t_{cache} = cache access time, t_{main} = main memory access time.

⌘ Average memory access time:

$$\boxed{\wedge} t_{\text{av}} = ht_{\text{cache}} + (1-h)t_{\text{main}}$$

Multiple levels of cache



Multi-level cache access time

⌘ h_1 = cache hit rate.

⌘ h_2 = hit rate on L2 and miss on L1.

⌘ Average memory access time:

$$\begin{aligned} \square t_{av} &= h_1 t_{L1} + (1-h_1)h_2 t_{L2} + (1-h_2)(1-h_1)t_{main} \\ &= h_1 t_{L1} + h_2^* t_{L2} + (1-h_1-h_2^*)t_{main} \end{aligned}$$

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



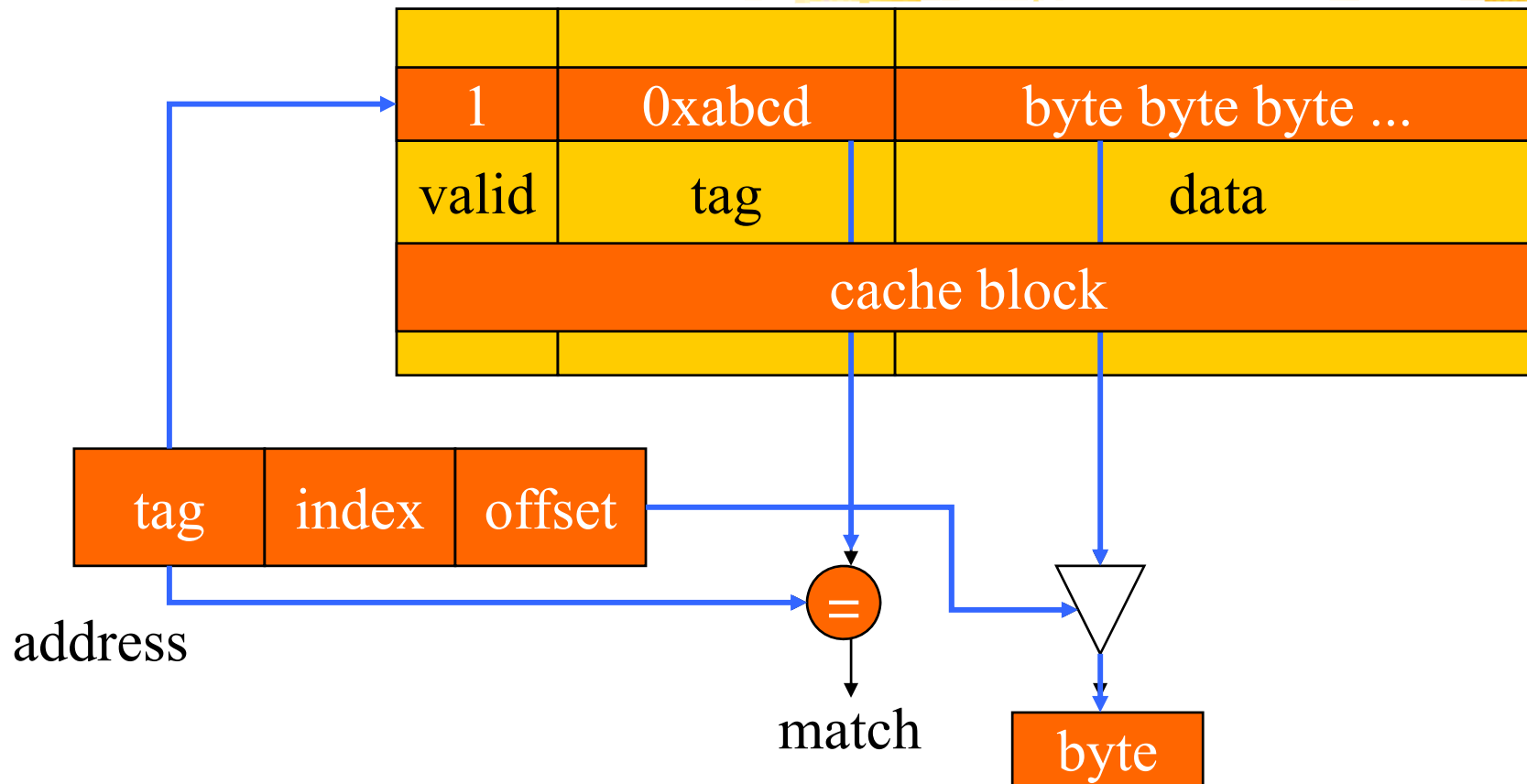
- ⌘ **Fully-associative**: any memory location can be stored anywhere in the cache (almost never implemented).
- ⌘ **Direct-mapped**: each memory location maps onto exactly one cache entry.
- ⌘ **N-way set-associative**: each memory location can go into one of n sets.

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.
 - ☑ **Spatial locality**

Direct-mapped cache



hit=match and valid

Write operations



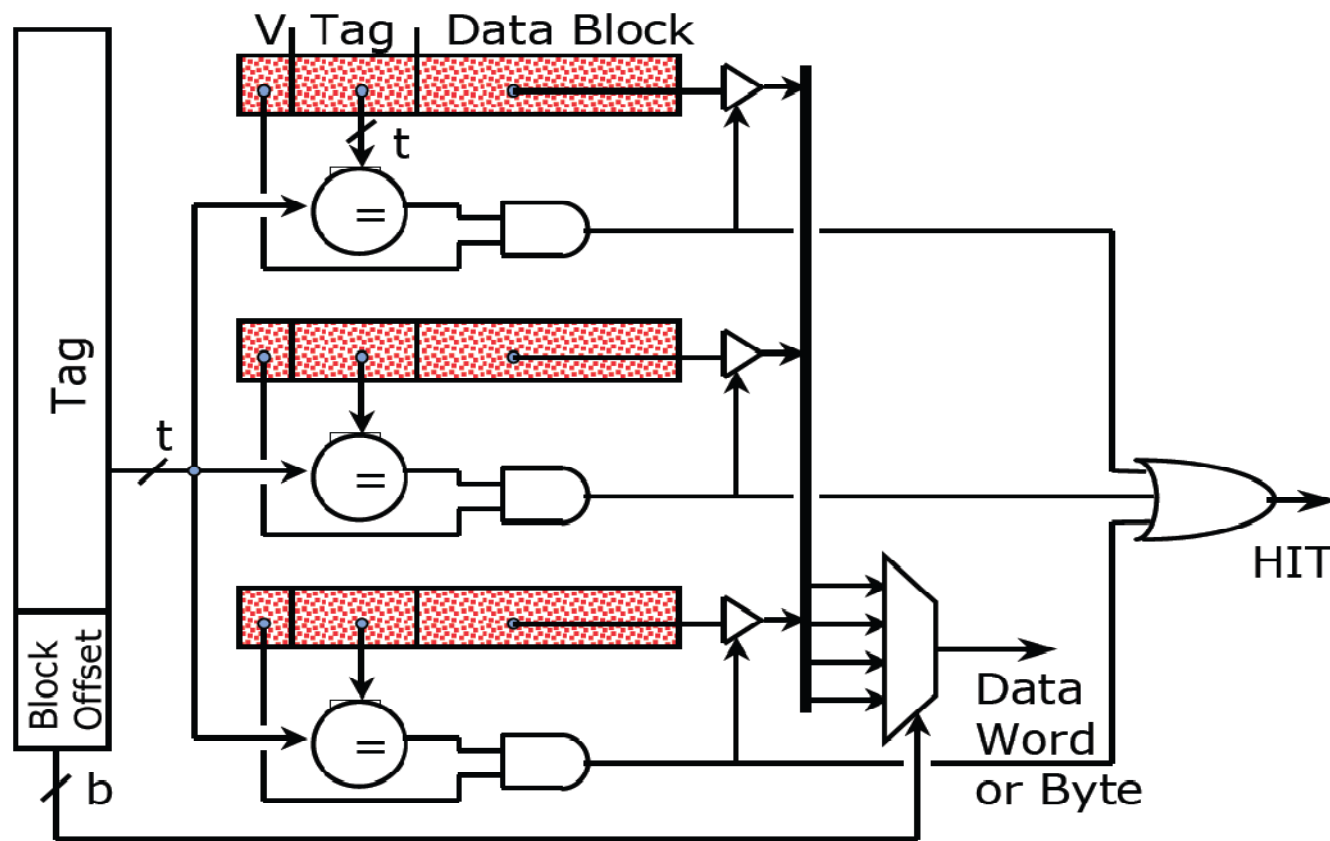
- ⌘ **Write-through**: immediately copy write to main memory.
- ⌘ **Write-back**: write to main memory only when location is removed from cache.

Direct-mapped cache locations



- ⌘ Many locations map onto the same cache block.
- ⌘ Conflict misses are easy to generate:
 - ☒ Array $a[]$ uses locations $0, 1, 2, \dots$
 - ☒ Array $b[]$ uses locations $1024, 1025, 1026, \dots$
 - ☒ Operation $a[i] + b[i]$ generates conflict misses.

Fully Associative Cache



Example caches



⌘ StrongARM:

- ☑ 16 Kbyte, 32-way, 32-byte block instruction cache.
- ☑ 16 Kbyte, 32-way, 32-byte block data cache (write-back).

⌘ C55x:

- ☑ Various models have 16KB, 24KB cache.
- ☑ Can be used as scratch pad memory.

Scratch pad memories



- ⌘ Alternative to cache:

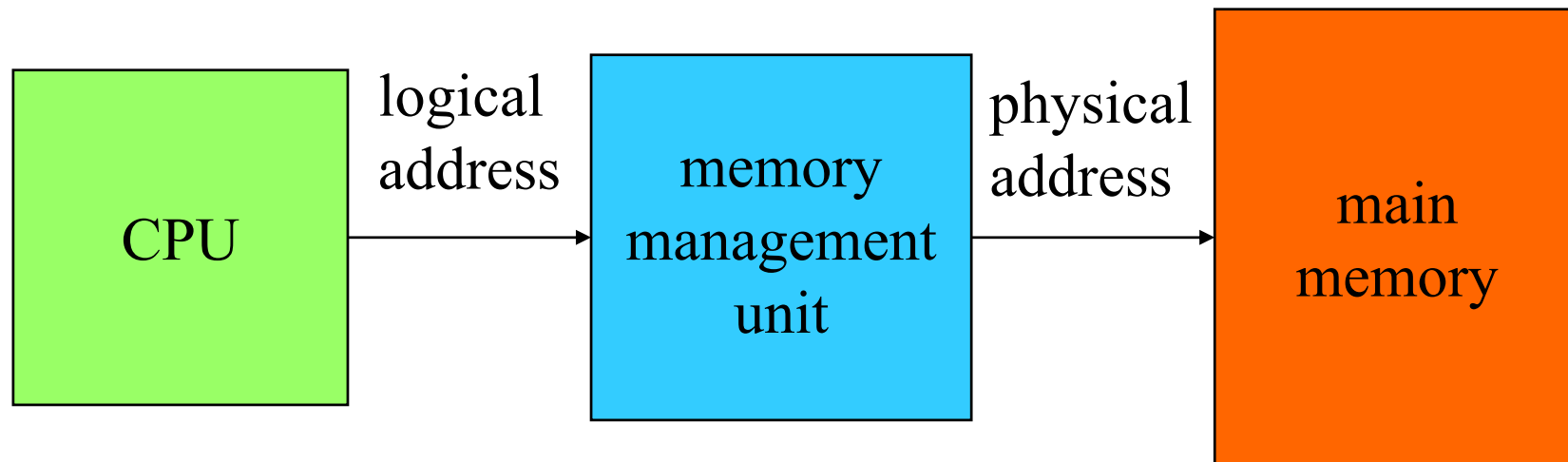
- ☑ Software determines what is stored in scratch pad.

- ⌘ Provides predictable behavior at the cost of software control.

- ⌘ C55x cache can be configured as scratch pad.

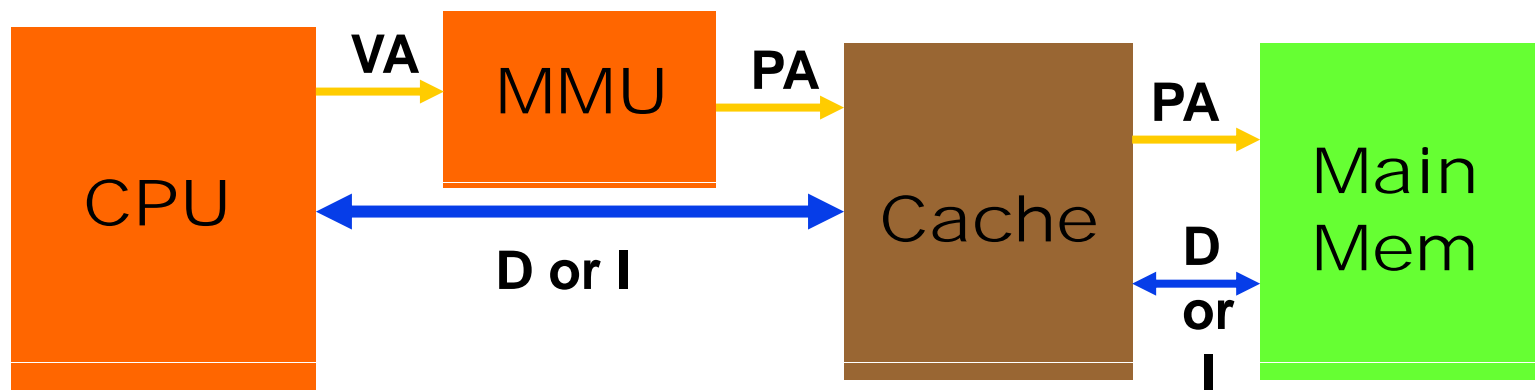
Memory management units

- ⌘ Memory management unit (MMU) translates addresses:



MMU

- ⌘ Responsible for
VIRTUAL → PHYSICAL
address mapping
- ⌘ Sits between CPU and cache



Access time comparison



Media	Read	Write	Erase
DRAM	60ns (2B) 2.56us (512B)	60ns (2B) 2.56us (512B)	N/A
NOR flash	150ns (2B) 14.4us (512B)	211us (2B) 3.53ms (512B)	1.2s (128KB)
NAND flash	10.2us (2B) 35.9us (512B)	201us (2B) 226us (512B)	2ms (16KB, 128K)
Disk	12.5ms (512B) (Average seek)	14.5ms (512B) (Average seek)	N/A

⌘ Price

⏏ HDD << NAND < DRAM < NOR

MMU - operation



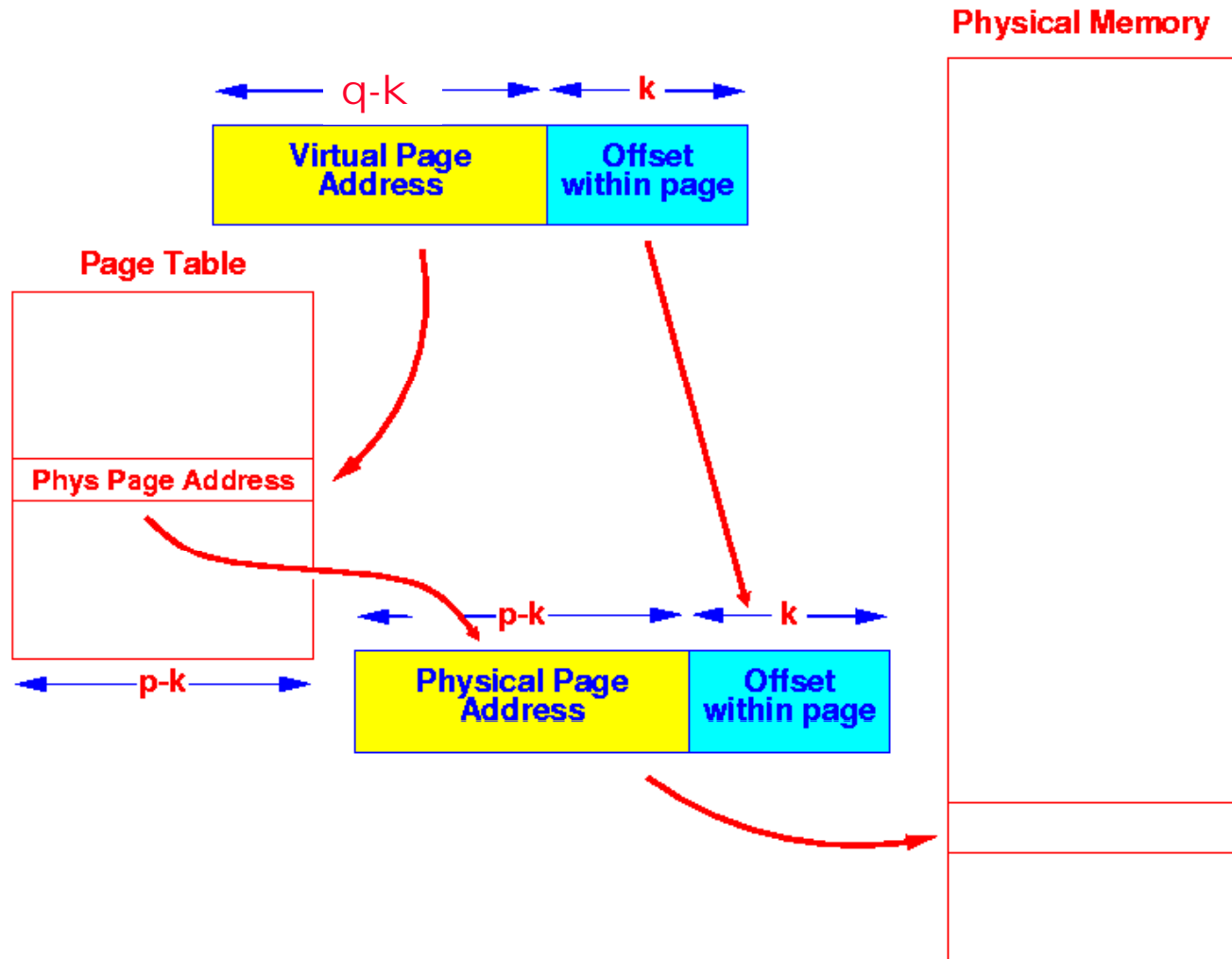
- ⌘ Operating System allocates pages of physical memory to users
- ⌘ OS constructs **page tables** - *one for each user*
- ⌘ Page address from memory address selects a page table entry
- ⌘ Page table entry contains physical page address

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.

MMU – address translation



MMU - Virtual memory space

⌘ Page Table Entries can also point to disc blocks

- ☑ If Valid bit is **set**, page in memory (address is physical page address); **cleared**, page “swapped out” (address is disc block address)

- ☑ MMU hardware generates **page fault** when swapped out page is requested

⌘ Allows virtual memory space to be larger than physical memory

- ☑ Only “**working set**” is in physical memory

- ☑ Remainder on paging disc

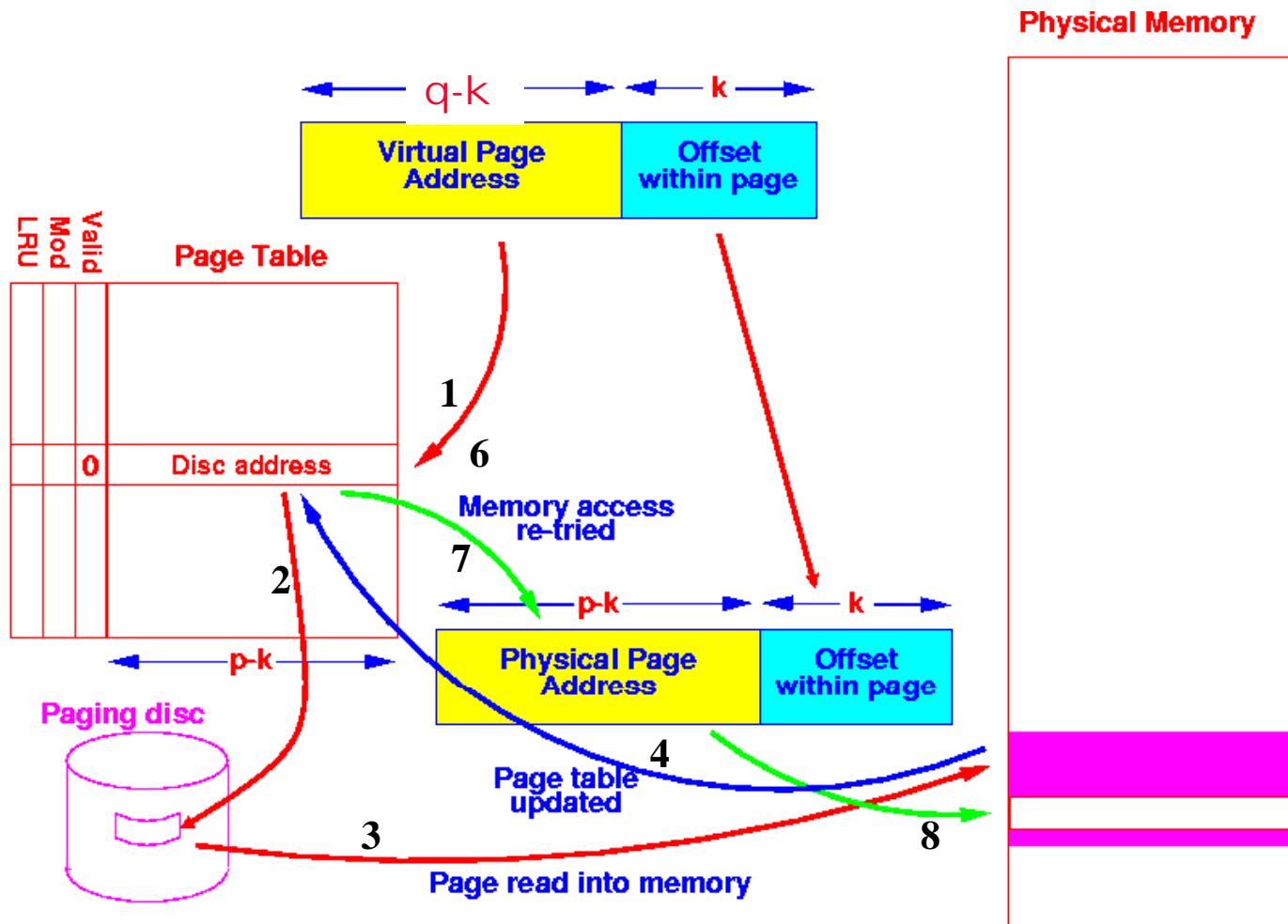
MMU - Page Faults



⌘ Page Fault Handler

- ☑ Part of OS kernel
- ☑ Finds usable physical page
 - ☒ LRU algorithm
- ☑ Writes it back to disc if modified
- ☑ Reads requested page from paging disc
- ☑ Adjusts page table entries
- ☑ Memory access re-tried

Page Fault



MMU - Page Faults



⌘ Page Fault Handler

- ☑ Part of OS kernel
- ☑ Finds usable physical page
 - ☒ LRU algorithm
- ☑ Writes it back to disc if modified
- ☑ Reads requested page from paging disc
- ☑ Adjusts page table entries
- ☑ Memory access re-tried

⌘ Can be an expensive process!

- ☑ Usual to allow page tables to be swapped out too!
- ⇒ Page fault can be generated on the page tables!

MMU - practicalities

⌘ Page size

☒ 8 kbyte pages $\Rightarrow k = 13$

☒ $q = 32$, $q - k = 19$

☒ So page table size

☒ $2^{19} \approx 0.5 \times 10^6$ entries

☒ Each entry 4 bytes

$\Rightarrow 0.5 \times 10^6 \times 4 \approx 2$ Mbytes!

⌘ Page tables can take a lot of memory!

☒ Larger page sizes reduce page table size
but can waste space (fragmentation)

MMU - practicalities



- ⌘ Page tables are stored in main memory
 - ☑ They're too large to be in smaller memories!
 - ☑ MMU needs to read page table for address translation
- ∴ Address translation can require additional memory accesses!

MMU - Protection



⌘ Page table entries

- ☑ Extra bits are added to specify access rights

 - ☑ Set by OS (software)

but

 - ☑ Checked by MMU hardware!

- ☑ Access control bits

 - ☑ Read

 - ☑ Write

 - ☑ Read/Write

 - ☑ Execute only

MMU - Alternative Page Table Styles

⌘ Inverted Page tables

☑ One page table entry (PTE) / **page of physical memory**

☑ MMU has to search for correct VA entry

∴ PowerPC **hashes** VA → PTE address

- PTE address = $h(VA)$

- h – hash function

☒ Hashing ⇒ collisions

⌘ Hash functions in hardware

☑ “hash” of n bits to produce m bits (Usually $m < n$)

☑ Fewer bits reduces information content

MMU - Alternative Page Table Styles

⌘ Hash functions in hardware

- ⊡ “Fewer bits reduces information content
 - ⊗ There are only 2^m distinct values now!
 - ⊗ Some n -bit patterns will reduce to the same m -bit patterns

⊡ Trivial example

- ⊗ 2-bits \rightarrow 1-bit with xor
- ⊗ $h(x_1 x_0) = x_1 \text{ xor } x_0$

y	$h(y)$
00	0
01	1
10	1
11	0

Collisions

MMU - Alternative Page Table Styles

⌘ Inverted Page tables

- ☒ One page table entry (PTE) / page of physical memory
- ☒ MMU has to search for correct VA entry
 - ∴ PowerPC **hashes** VA → PTE address
 - PTE address = $h(VA)$
 - h – hash function
 - ☒ Hashing ⇒ collisions
 - ☒ PTEs are linked together
 - PTE contains **tags** (like cache) and link bits
 - ☒ MMU searches linked list to find correct entry
- ☒ Smaller Page Tables / Longer searches

Address Translation - Speeding it up

⌘ Two+ memory accesses for each datum?

☑ Page table 1 - 3 (single - 3 level tables)

☑ Actual data 1

☑ *system can be slowed down*

⌘ Translation Look-Aside Buffer

- Acronym: TLB or TLAB
- Small **cache** of recently-used **page table entries**
- Usually fully-associative
- Can be quite small!

Address Translation - Speeding it up

⌘ TLB sizes

☒ MIPS R10000 1996 64 entries

☒ Pentium 4 (Prescott) 2006 64 entries

- One page table entry / **page** of data
- *Locality of reference*
 - Programs spend a lot of time in same memory region

⇒ TLB hit rates tend to be very high

- 98%

⇒ Compensate for cost of a miss

(many memory accesses –
but for only 2% of references to memory!)

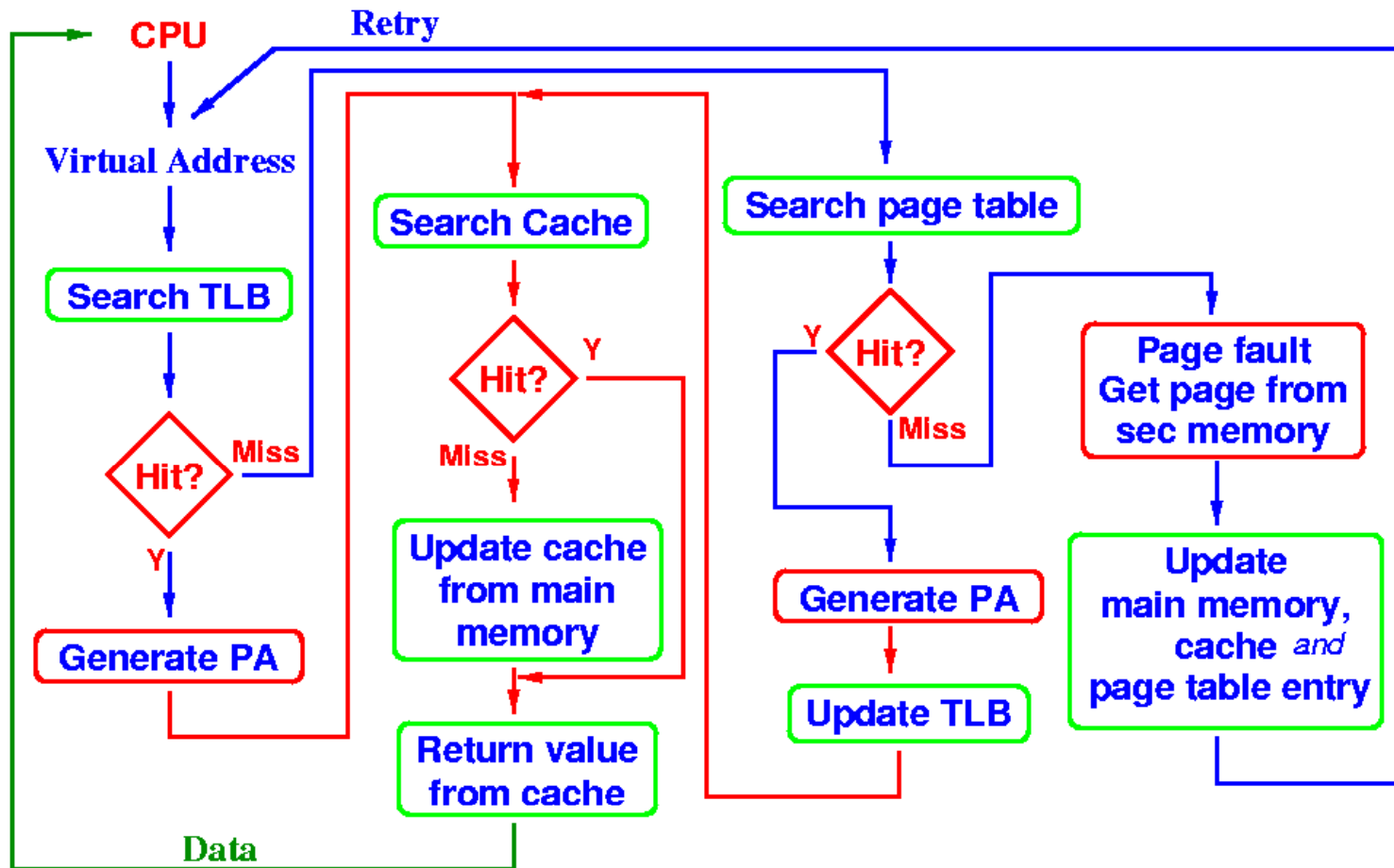
TLB – Sequential access



- ⌘ Luckily, sequential access is fine!
- ⌘ Example: large (several MByte) matrix of doubles (8 bytes floating point values)
 - ☒ 8kbyte pages => 1024 doubles/page
- ⌘ Sequential access, *eg* sum all values:

```
for (j=0; j<n; j++)  
    sum = sum + x[j]
```

Memory Hierarchy - Operation

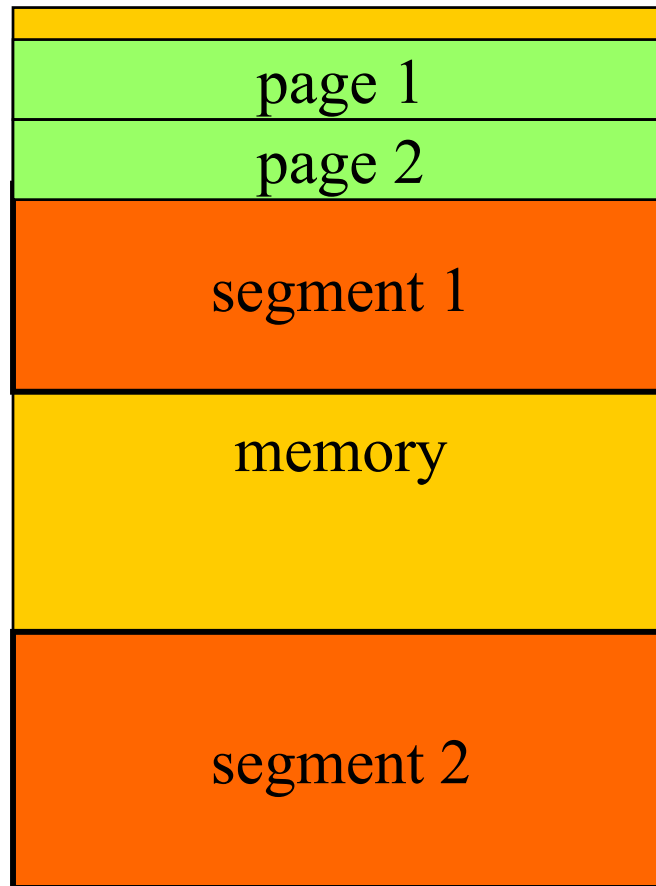


Address translation

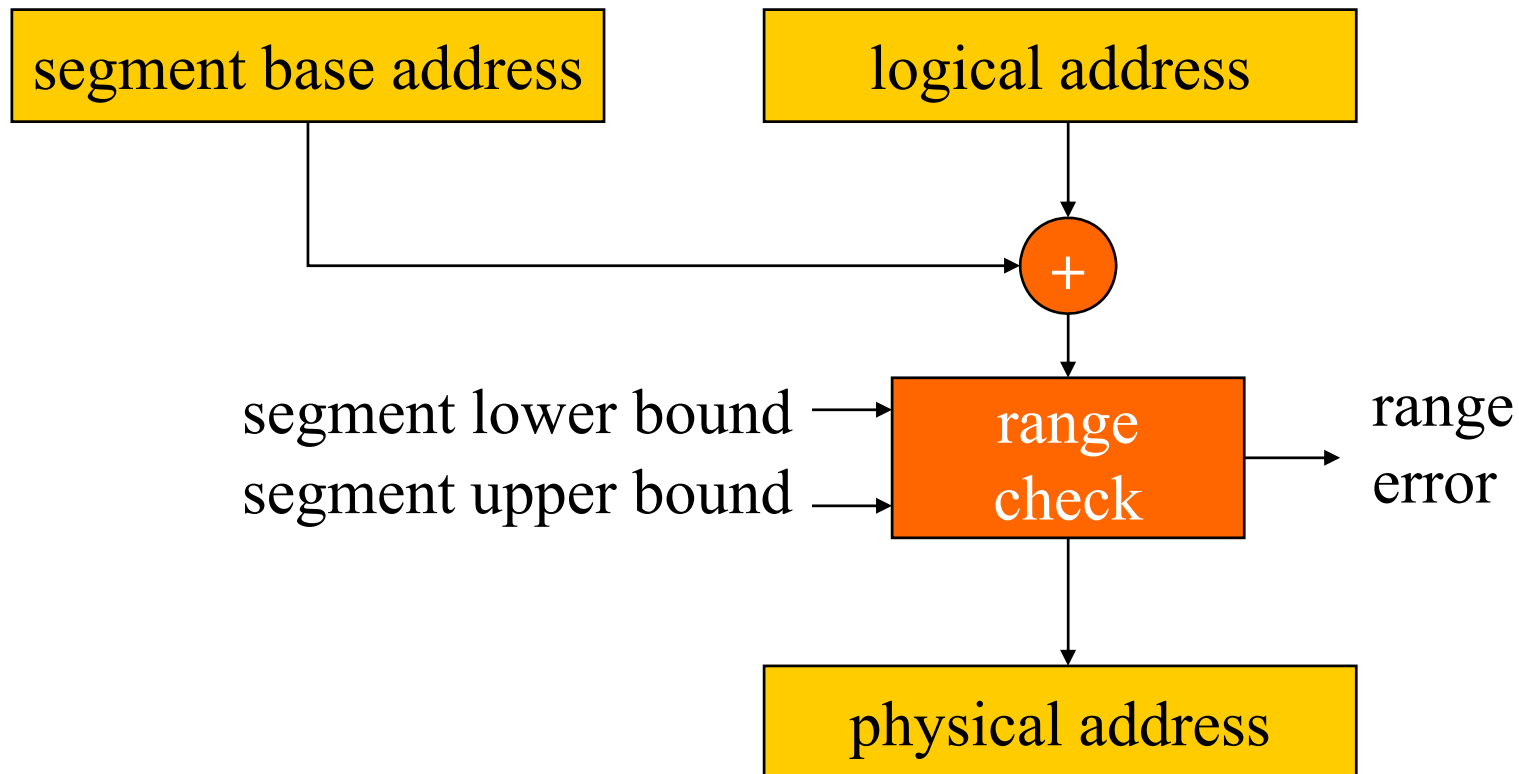


- ⌘ Requires some sort of register/table to allow arbitrary mappings of logical to physical addresses.
- ⌘ Two basic schemes:
 - ☒ segmented;
 - ☒ paged.
- ⌘ Segmentation and paging can be combined (x86).

Segments and pages



Segment address translation



ARM memory management



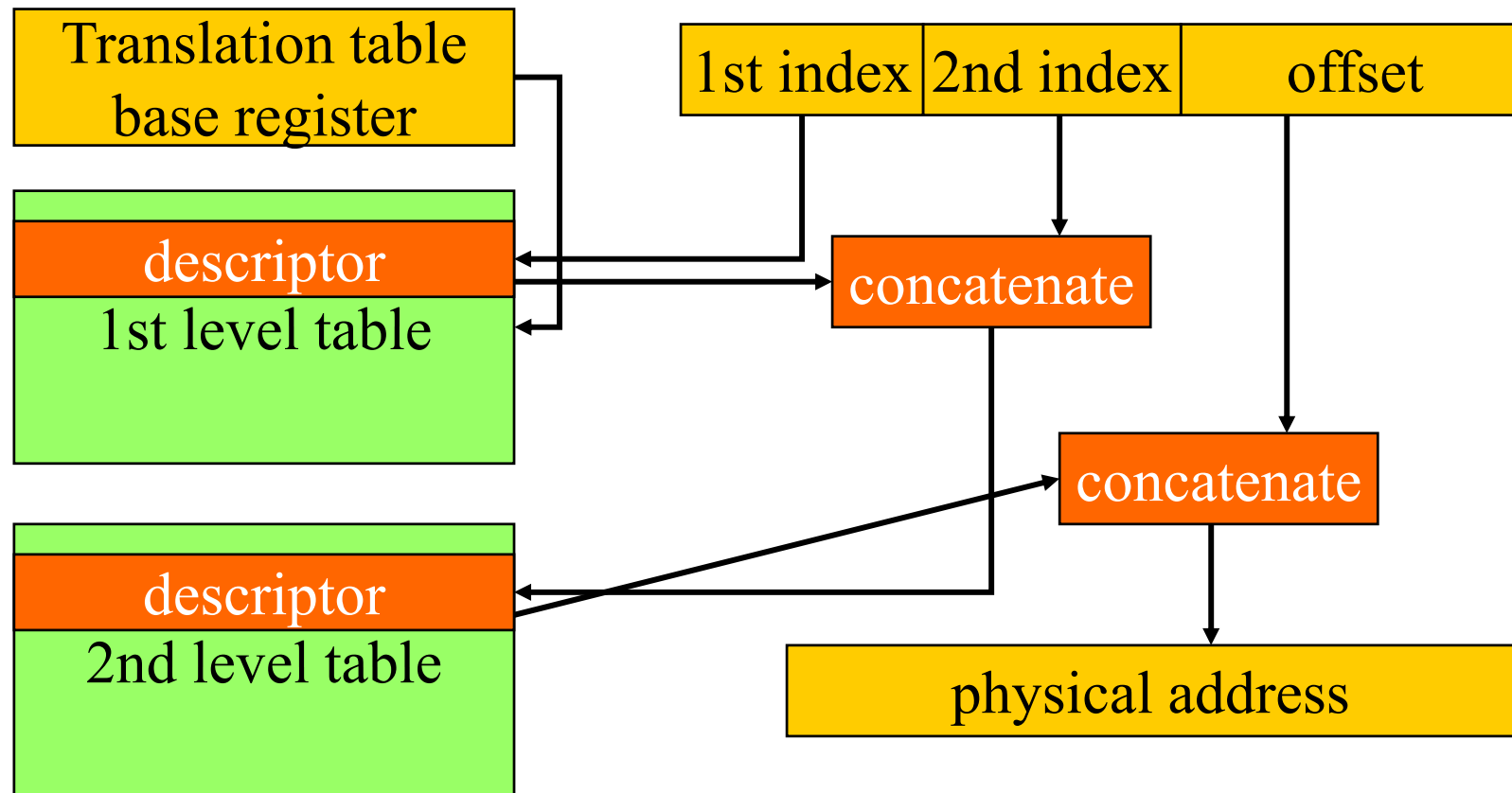
⌘ Memory region types:

- ☑ section: 1 Mbyte block;
- ☑ large page: 64 kbytes;
- ☑ small page: 4 kbytes.

⌘ An address is marked as section-mapped or page-mapped.

⌘ Two-level translation scheme.

ARM address translation



Elements of CPU performance



- ⌘ Cycle time.
- ⌘ CPU pipeline.
- ⌘ Memory system.

Pipelining



- ⌘ Several instructions are executed simultaneously at different stages of completion.
- ⌘ Various conditions can cause **pipeline bubbles** that reduce utilization:
 - ☑ branches;
 - ☑ memory system delays;
 - ☑ etc.

Performance measures



- ⌘ **Latency**: time it takes for an instruction to get through the pipeline.
- ⌘ **Throughput**: number of instructions executed per time period.
- ⌘ Pipelining increases throughput without reducing latency.

ARM7 pipeline



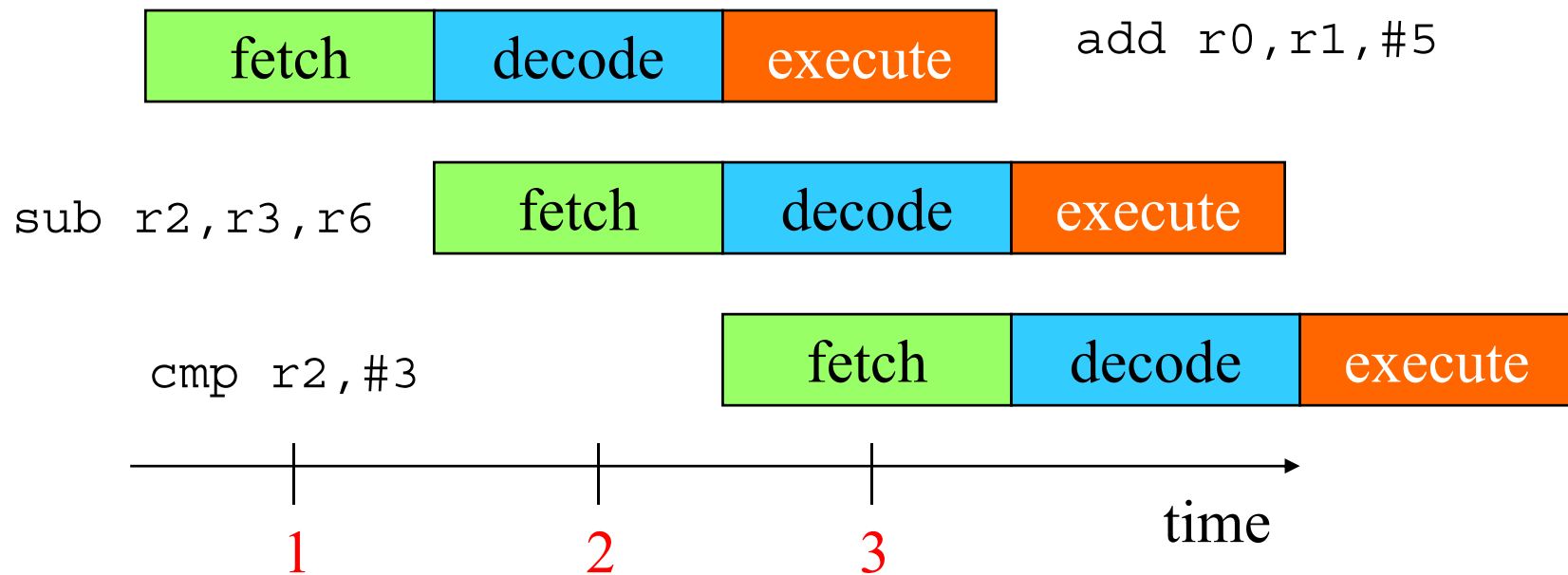
⌘ ARM 7 has 3-stage pipe:

☒ **fetch** instruction from memory;

☒ **decode** opcode and operands;

☒ **execute**.

ARM pipeline execution

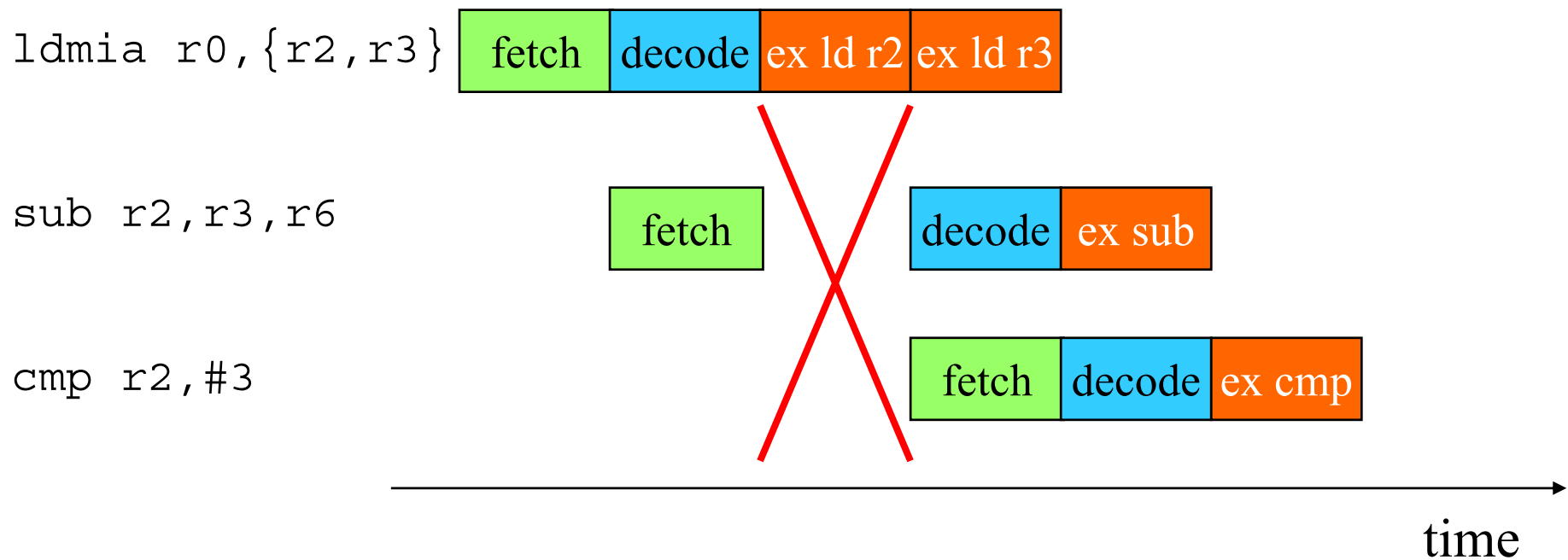


Pipeline stalls



- ⌘ If every step cannot be completed in the same amount of time, pipeline stalls.
- ⌘ Bubbles introduced by stall increase latency, reduce throughput.

ARM multi-cycle LDMIA instruction

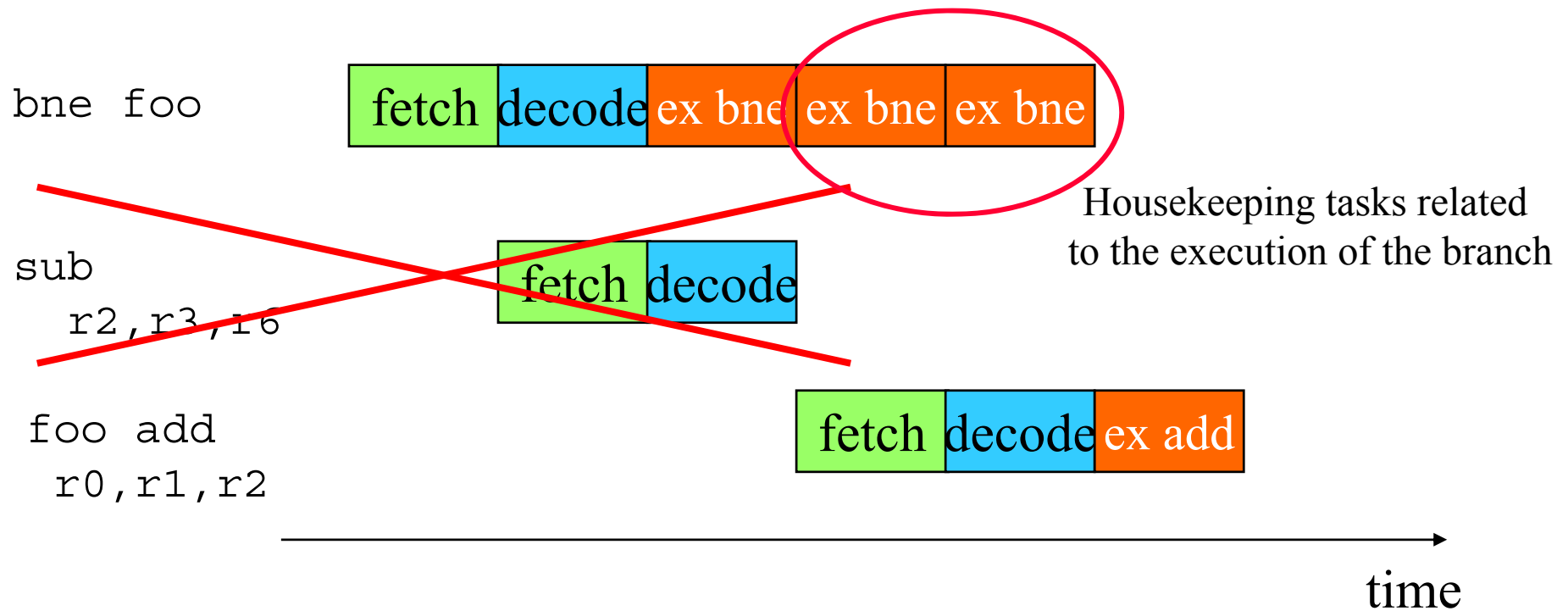


Control stalls



- ⌘ Branches often introduce stalls (branch penalty).
 - ☑ Stall time may depend on whether branch is taken.
- ⌘ May have to squash instructions that already started executing.
- ⌘ Don't know what to fetch until condition is evaluated.

ARM pipelined branch



Delayed branch



- ⌘ To increase pipeline efficiency, delayed branch mechanism requires n instructions after branch always executed whether branch is executed or not.

Example: ARM execution time

⌘ Determine execution time of FIR filter:

```
for (i=0; i<N; i++)
```

```
    f = f + c[i]*x[i];
```

⌘ Only branch in loop test may take more than one cycle.

⊠ **BLT loop** takes 1 cycle best case, 3 worst case.

FIR filter ARM code

```

; loop initiation code
MOV r0,#0 ; use r0 for i, set to 0
MOV r8,#0 ; use an index for arrays
ADR r2,N ; get address for N
LDR r1,[r2] ; get value of N
MOV r2,#0 ; use r2 for f, set to 0
ADR r3,c ; load r3 with C base
ADR r5,x ; load r5 with x base

; loop body
Loop LDR r4,[r3,r8] ; get value of c[i]
LDR r6,[r5,r8] ; get value of x[i]
MUL r4,r4,r6 ; compute c[i]*x[i]
ADD r2,r2,r4 ; add into running sum
; update loop counter and array index
ADD r8,r8,#4 ; add one to array index
ADD r0,r0,#1 ; add 1 to i
; test for exit
CMP r0,r1
BLT loop ; if i < N, continue loop
loopend ...
```

FIR filter performance by block

Block	Variable	# instructions	# cycles
Initialization	t_{init}	7	7
Body	t_{body}	4	4
Update	t_{update}	2	2
Test	t_{test}	2	[2,4]

$$t_{\text{loop}} = t_{\text{init}} + N(t_{\text{body}} + t_{\text{update}}) + (N-1)t_{\text{test,worst}} + t_{\text{test,best}}$$

Loop test succeeds is worst case

Loop test fails is best case

Memory system performance



⌘ Caches introduce indeterminacy in execution time.

☑ Depends on order of execution.

⌘ **Cache miss penalty**: added time due to a cache miss.

CPU power consumption



- ⌘ Most modern CPUs are designed with power consumption in mind to some degree.
- ⌘ Power vs. energy:
 - ☑ heat depends on power consumption;
 - ☑ battery life depends on energy consumption.

CMOS power consumption



- ⌘ **Voltage drops**: power consumption proportional to V^2 .
- ⌘ **Toggling**: more activity means more power.
- ⌘ **Leakage**: basic circuit characteristics; can be eliminated by disconnecting power.

CPU power-saving strategies



- ⌘ Reduce power supply voltage.
- ⌘ Run at lower clock frequency.
- ⌘ Disable function units with control signals when not in use.
- ⌘ Disconnect parts from power supply when not in use.

C55x low power features



- ⌘ Parallel execution units---longer idle shutdown times.
- ⌘ Multiple data widths:
 - ☑ 16-bit ALU vs. 40-bit ALU.
- ⌘ Instruction caches minimizes main memory accesses.
- ⌘ Power management:
 - ☑ Function unit idle detection.
 - ☑ Memory idle detection.
 - ☑ User-configurable IDLE domains allow programmer control of what hardware is shut down.

Power management styles



- ⌘ **Static power management**: does not depend on CPU activity.
 - ☑ Example: user-activated power-down mode.
- ⌘ **Dynamic power management**: based on CPU activity.
 - ☑ Example: disabling off function units.

Application: PowerPC 603 energy features



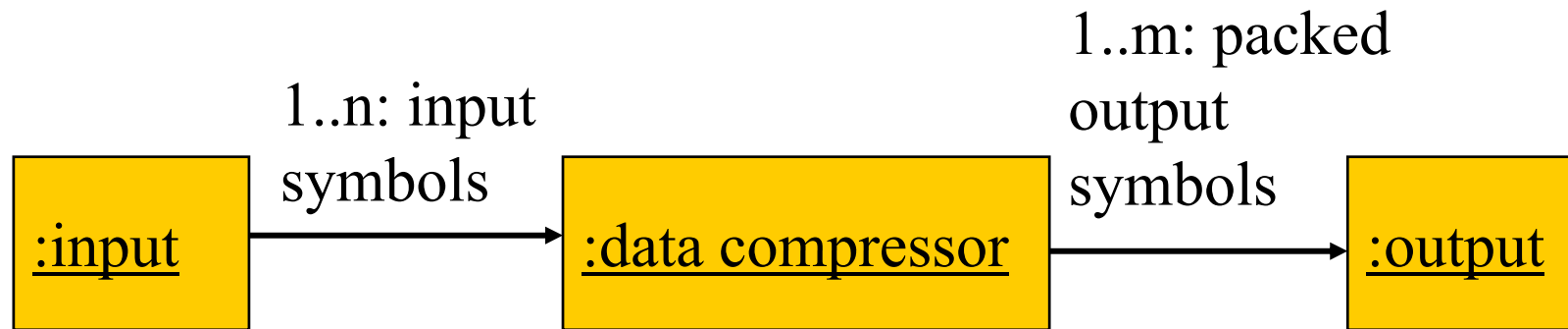
- ⌘ Provides doze, nap, sleep modes.
- ⌘ Dynamic power management features:
 - ☑ Uses static logic.
 - ☑ Can shut down unused execution units.
 - ☑ Cache organized into subarrays to minimize amount of active circuitry.

Goals



- ⌘ Compress data transmitted over serial line.
 - ☑ Receives byte-size input symbols.
 - ☑ Produces output symbols packed into bytes.
- ⌘ Will build software module only here.

Collaboration diagram for compressor

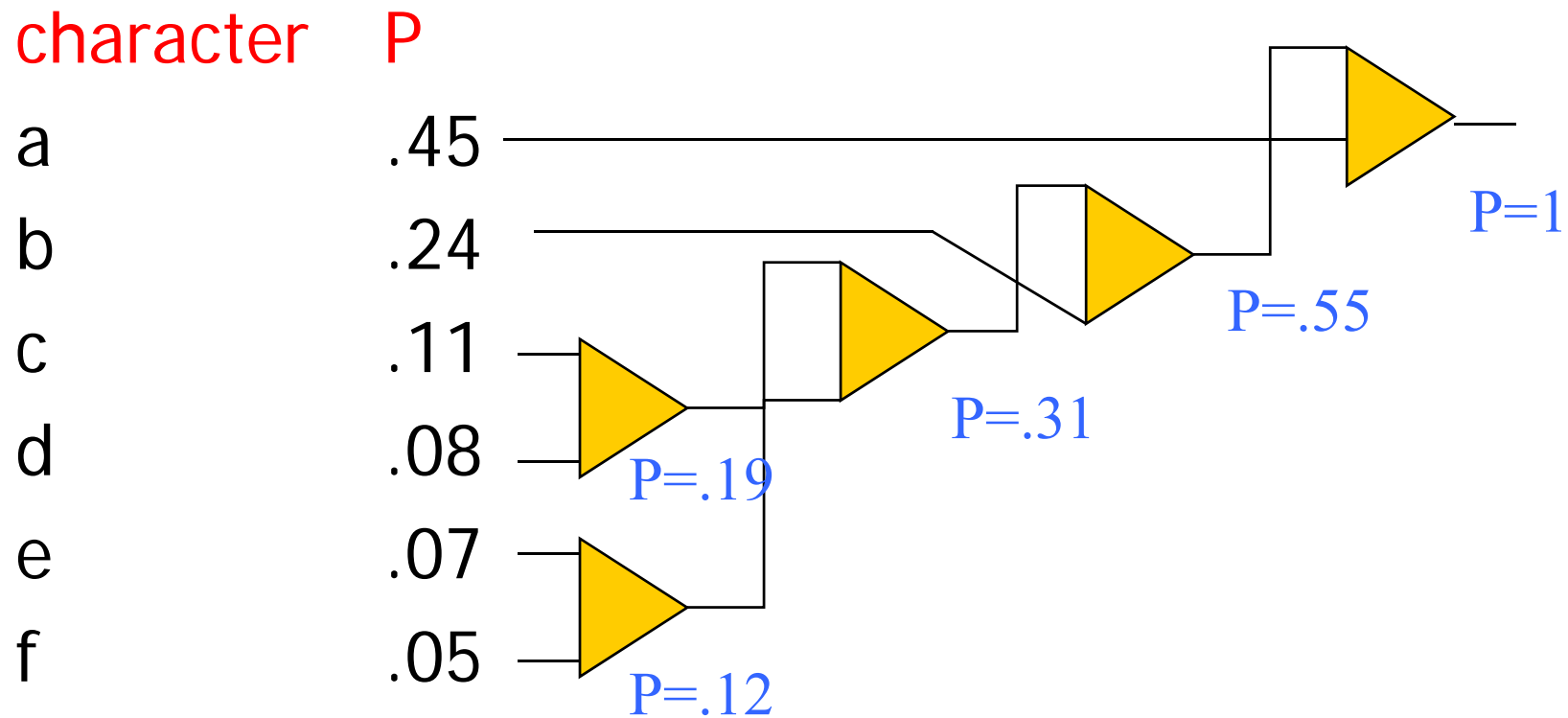


Huffman coding



- ⌘ Early statistical text compression algorithm.
- ⌘ Select non-uniform size codes.
 - ☑ Use shorter codes for more common symbols.
 - ☑ Use longer codes for less common symbols.
- ⌘ To allow decoding, codes must have unique prefixes.
 - ☑ No code can be a prefix of a longer valid code.

Huffman example



Example Huffman code



⌘ Read code from root to leaves:

a	1
b	01
c	0000
d	0001
e	0010
f	0011

Huffman coder requirements table



name	data compression module
purpose	code module for Huffman compression
inputs	encoding table, uncoded byte-size inputs
outputs	packed compression output symbols
functions	Huffman coding
performance	fast
manufacturing cost	N/A
power	N/A
physical size/weight	N/A

Building a specification



- ⌘ Collaboration diagram shows only steady-state input/output.
- ⌘ A real system must:
 - ☑ Accept an encoding table.
 - ☑ Allow a system reset that flushes the compression buffer.

data-compressor class

data-compressor

buffer: data-buffer
table: symbol-table
current-bit: integer

encode(): boolean,
 data-buffer
flush()
new-symbol-table()

data-compressor behaviors



- ⌘ **encode**: Takes one-byte input, generates packed encoded symbols and a Boolean indicating whether the buffer is full.
- ⌘ **new-symbol-table**: installs new symbol table in object, throws away old table.
- ⌘ **flush**: returns current state of buffer, including number of valid bits in buffer.

Auxiliary classes

data-buffer

databuf[databuflen] :
 character
len : integer

insert()
length() : integer

symbol-table

symbols[nsymbols] :
 data-buffer
len : integer

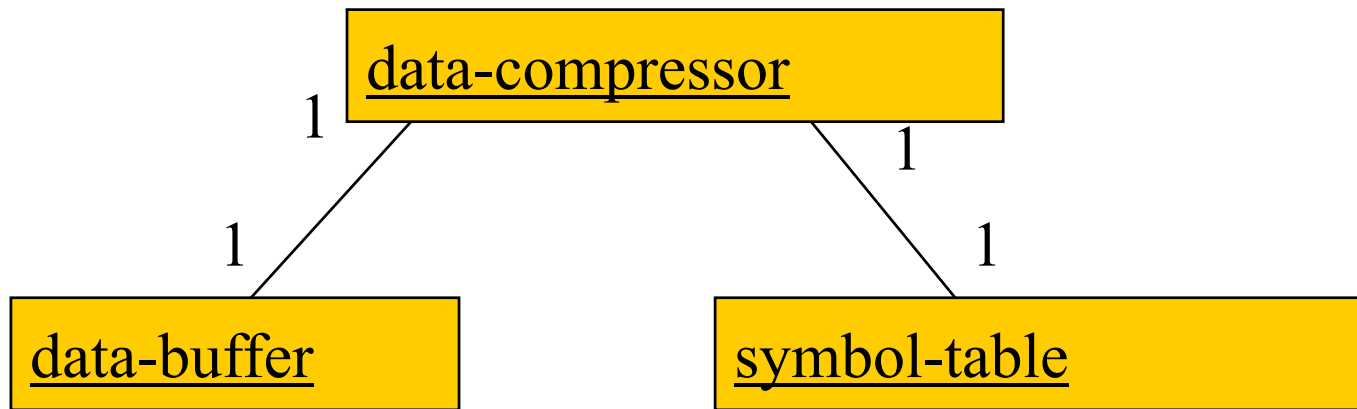
value() : symbol
load()

Auxiliary class roles

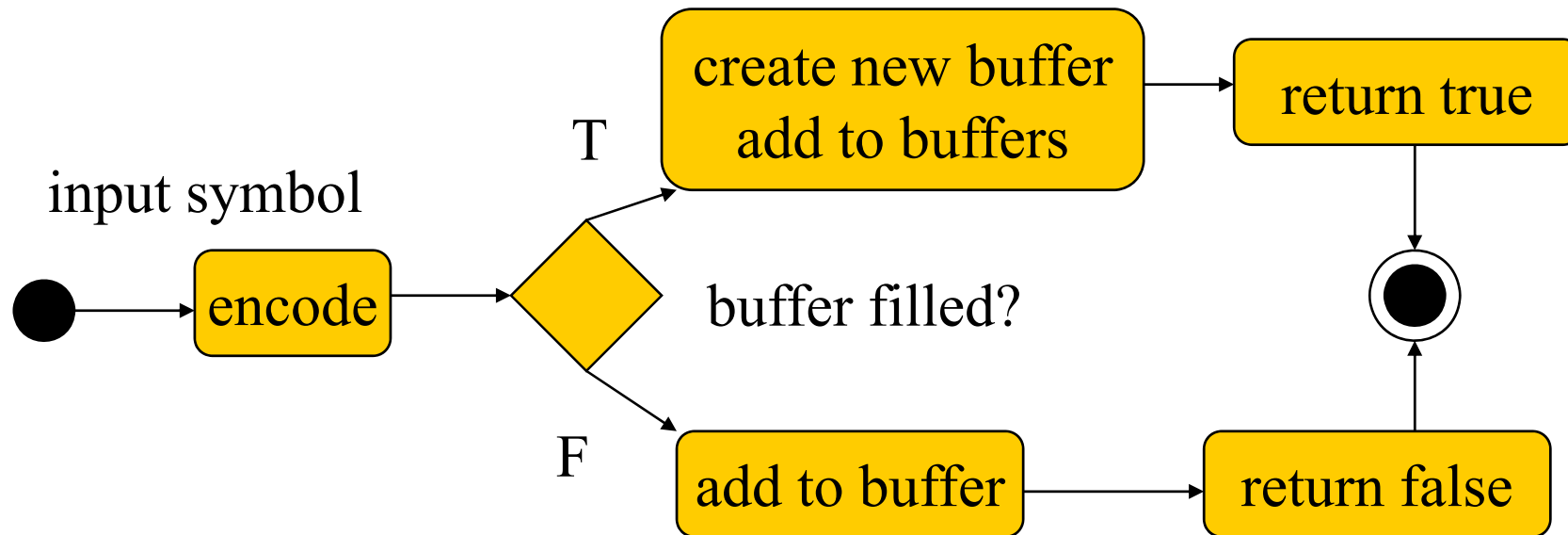


- ⌘ data-buffer holds both packed and unpacked symbols.
 - ☑ Longest Huffman code for 8-bit inputs is 256 bits.
- ⌘ symbol-table indexes encoded version of each symbol.
 - ☑ load() puts data in a new symbol table.

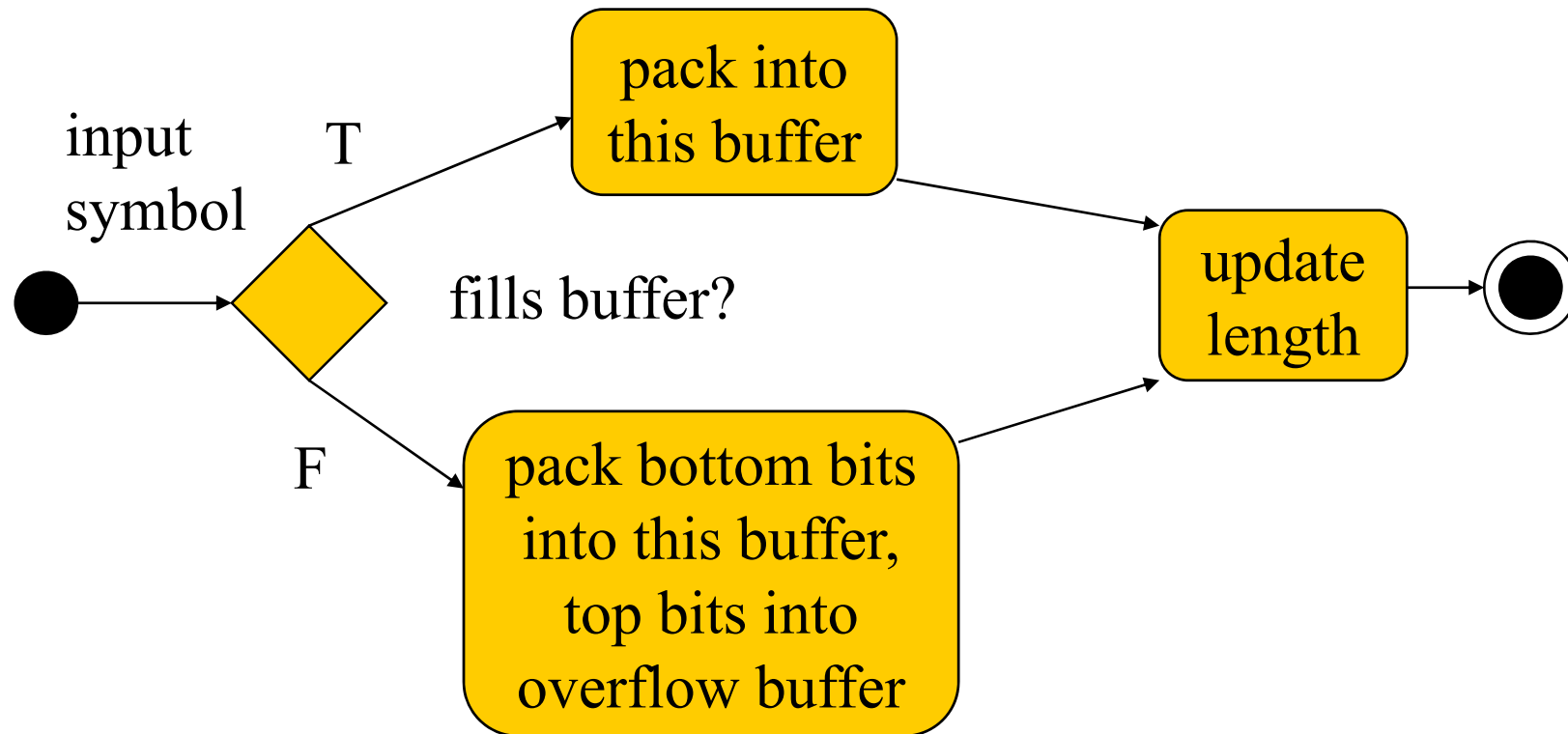
Class relationships



Encode behavior



Insert behavior



Program design



- ⌘ In an object-oriented language, we can reflect the UML specification in the code more directly.
- ⌘ In a non-object-oriented language, we must either:
 - ☑ add code to provide object-oriented features;
 - ☑ diverge from the specification structure.

C++ classes



```
Class data_buffer {  
    char databuf[databuflen];  
    int len;  
    int length_in_chars() { return len/bitsperbyte; }  
public:  
    void insert(data_buffer,data_buffer&);  
    int length() { return len; }  
    int length_in_bytes() { return (int)ceil(len/8.0); }  
    int initialize();  
    ...  
};
```

C++ classes, cont'd.



```
class data_compressor {
    data_buffer buffer;
    int current_bit;
    symbol_table table;
public:
    boolean encode(char, data_buffer&);
    void new_symbol_table(symbol_table);
    int flush(data_buffer&);
    data_compressor();
    ~data_compressor();
}
```

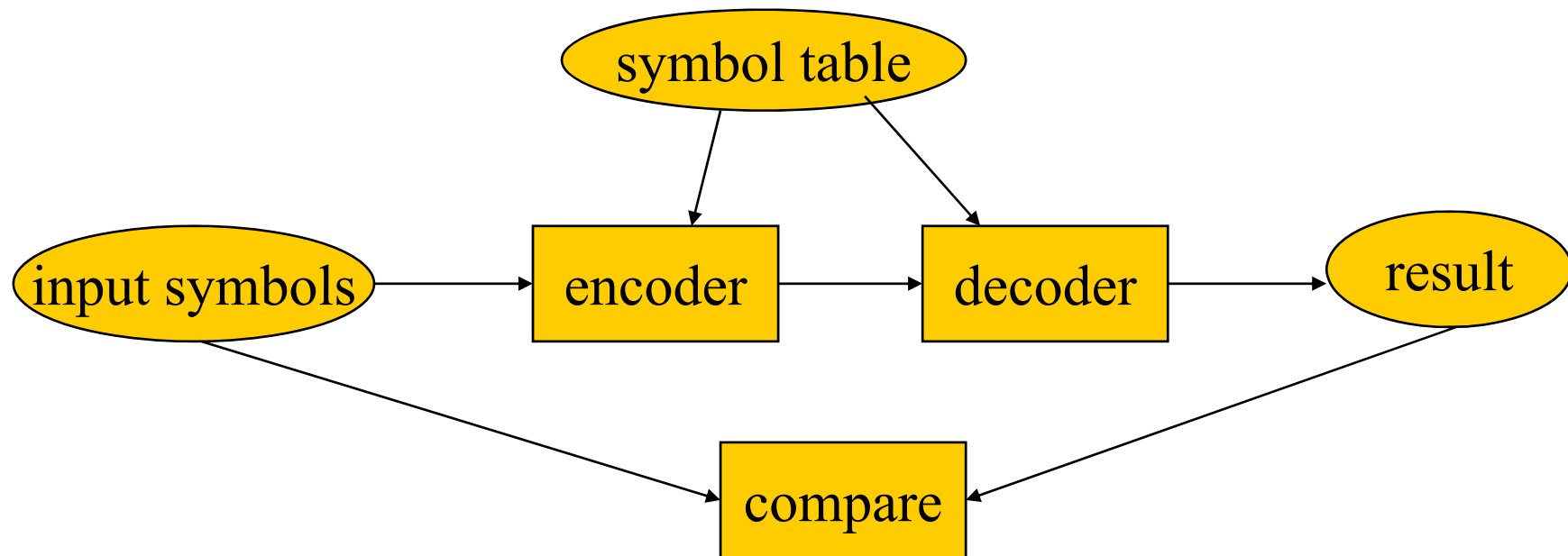
C code



```
struct data_compressor_struct {
    data_buffer buffer;
    int current_bit;
    sym_table table;
}
typedef struct data_compressor_struct data_compressor,
    *data_compressor_ptr;
boolean data_compressor_encode(data_compressor_ptr
    mycmptrs, char isymbol, data_buffer *fullbuf) ...
```

Testing

⌘ Test by encoding, then decoding:



Code inspection tests



⌘ Look at the code for potential problems:

- ☑ Can we run past end of symbol table?
- ☑ What happens when the next symbol does not fill the buffer? Does fill it?
- ☑ Do very long encoded symbols work properly? Very short symbols?
- ☑ Does flush() work properly?