

# ARM subroutine linkage

## ⌘ Branch and link instruction:

```
BL foo
```

☑ Copies current PC to r14.

## ⌘ To return from subroutine:

```
MOV r15,r14
```

# Nested subroutine calls

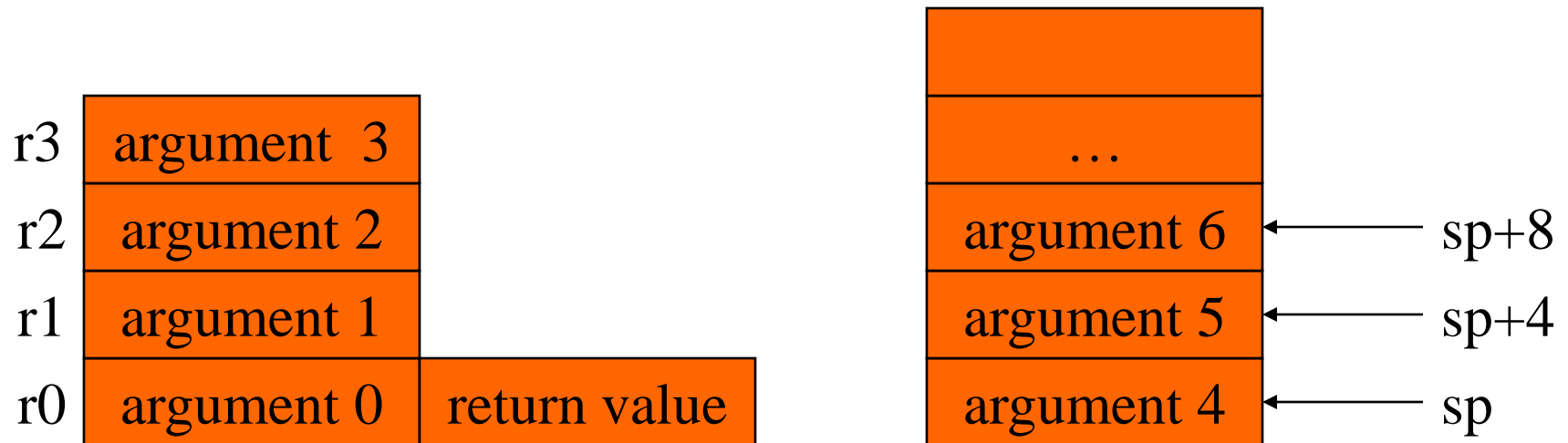
## ⌘ Nesting/recursion requires coding convention:

```
f1      LDR r0,[sp]          ; read arg  from stack
        ; call f2()
        STR lr,[sp, #-4]! ; push f1's return adrs
        STR r0,[sp, #-4]! ; push arg to f2 on stack
        BL f2 ; branch and link to f2
        ; return from f1()
        ADD sp,#4          ; pop f2's arg off stack
        LDR pc,[sp,#4]!    ; pop return address
```

# ARM-Thumb procedure call standard (ATPCS)



- ⌘ The first four integer arguments are passed in the four ARM register: 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.

# Example: STM--LDM pair

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

```
mid    r13=0x0000900c
        r1=0x00000001
        r2=0x00000002
        r3=0x00000003
```

```
mem32[0x0000900c]=0x00000007
mem32[0x00009008]=0x00000008
mem32[0x00009004]=0x00000009
mem32[0x00009000]=0x0000000A
```

```
pre    r13=0x00009000
        r1=0x00000009
        r2=0x00000008
        r3=0x00000007
```

```
mem32[0x0000900c]=0x0000000D
mem32[0x00009008]=0x0000000C
mem32[0x00009004]=0x0000000B
mem32[0x00009000]=0x0000000A
```

```
post   r13=0x00009000
        r1=0x00000009
        r2=0x00000008
        r3=0x00000007
```

# Stack operations

⌘ (pop,push) for each addressing mode

⊞ Full ascending:

⊗ ( pop, push): (LDMFA, STMFA)=(LDMDA, STMIB)

⊞ Full descending:

⊗ (pop, push) : (LDMFD, STMFD)=(LDMIA, STMDB)

# Load-store multiple instructions

$\langle \text{LDM|STM} \rangle \{ \langle \text{cond} \rangle \} \{ \text{addressing\_mode} \} \{ S \} R_n \{ ! \}, \langle \text{registers} \rangle \{ ^ \}$

⌘ Addressing mode (N is the number of addresses in the register list)

(start address, end address,  $R_n$ !)

☒ 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:  $R_n-4, R_n-4N, R_n-4N$

# Nested subroutine calls

```
label0    BL      SUB1
          ..

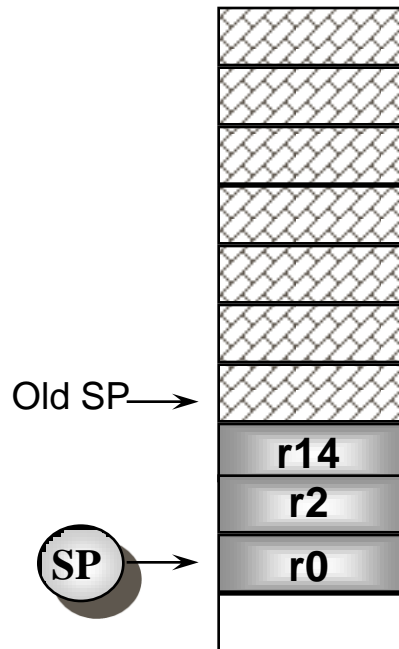
SUB1      STMFD  sp!,{r0-r2,r14} ; save work & link register
          BL      SUB2
          ..
          LDMFD  sp!, {r0-r2,pc} ; restore work regs & link
          ..

SUB2      ..
          MOV    pc, r14          ; return
```

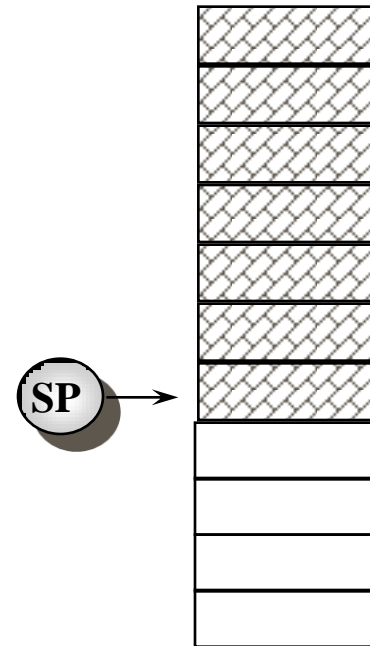


# Stack Examples

STMFD sp!, {r0, r2, r14}



LDMFD sp!, {r0, r2, pc}



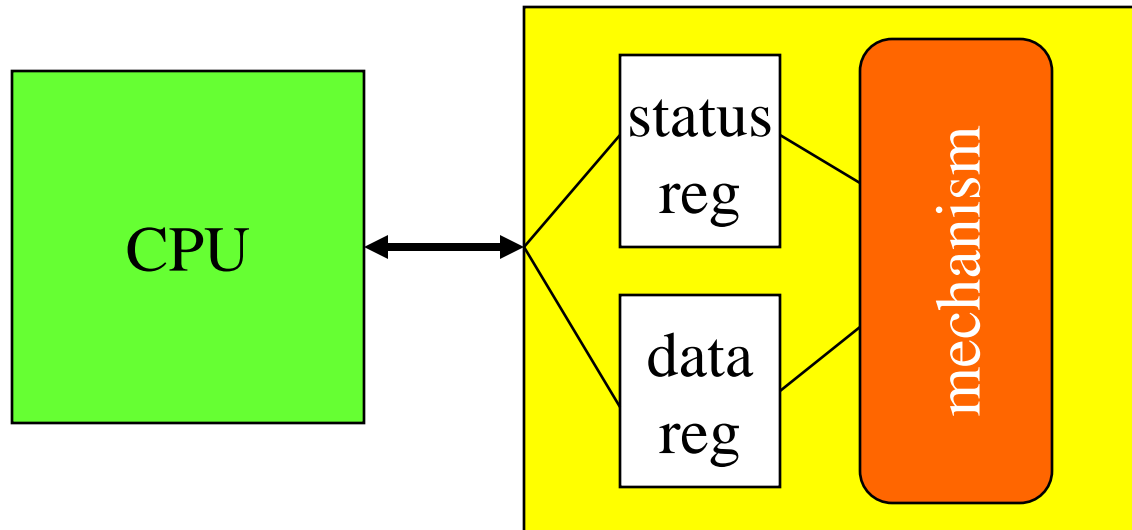
# CPUs



- ⌘ 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. (ADC and DAC 필요)
- ⌘ Typical digital interface to CPU:



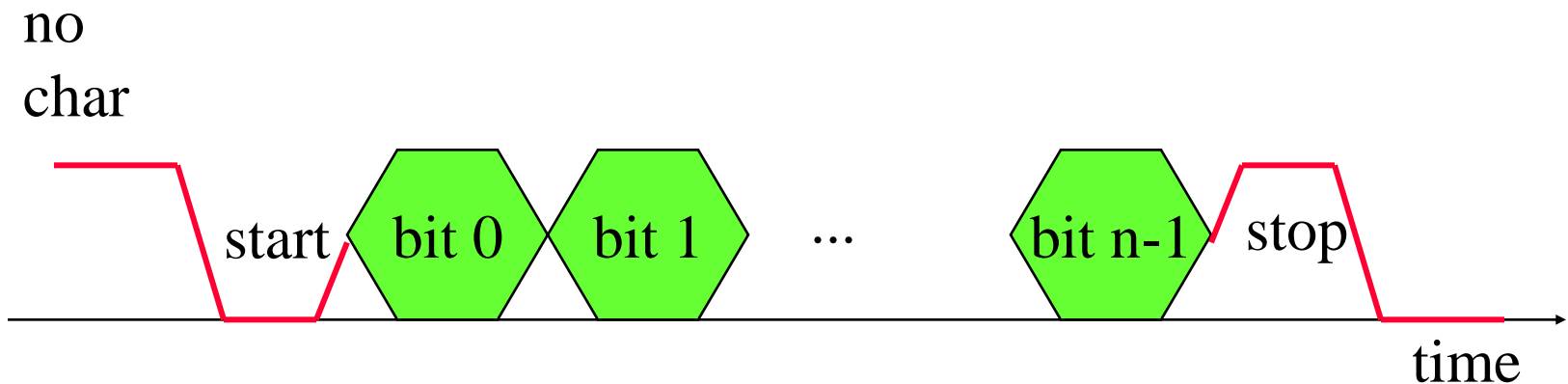
# Application: 8251 UART



- ⌘ 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

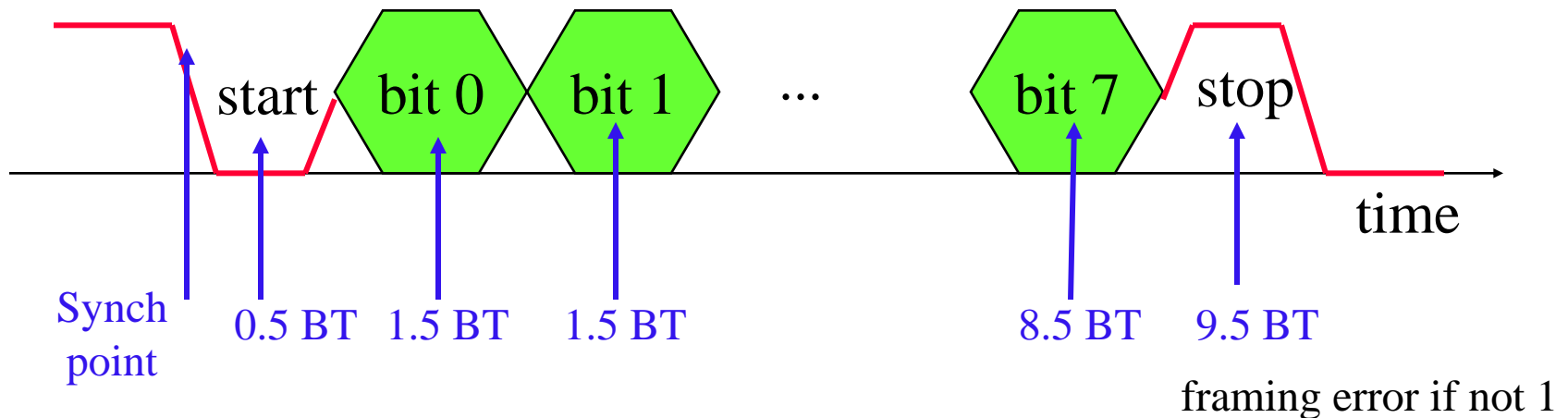


- ⌘ Baud (bit) rate.
- ⌘ Number of bits per character (5-8).
- ⌘ Parity/no parity.
- ⌘ Even/odd parity.
- ⌘ Length of stop bit (1, 1.5, 2 bits).

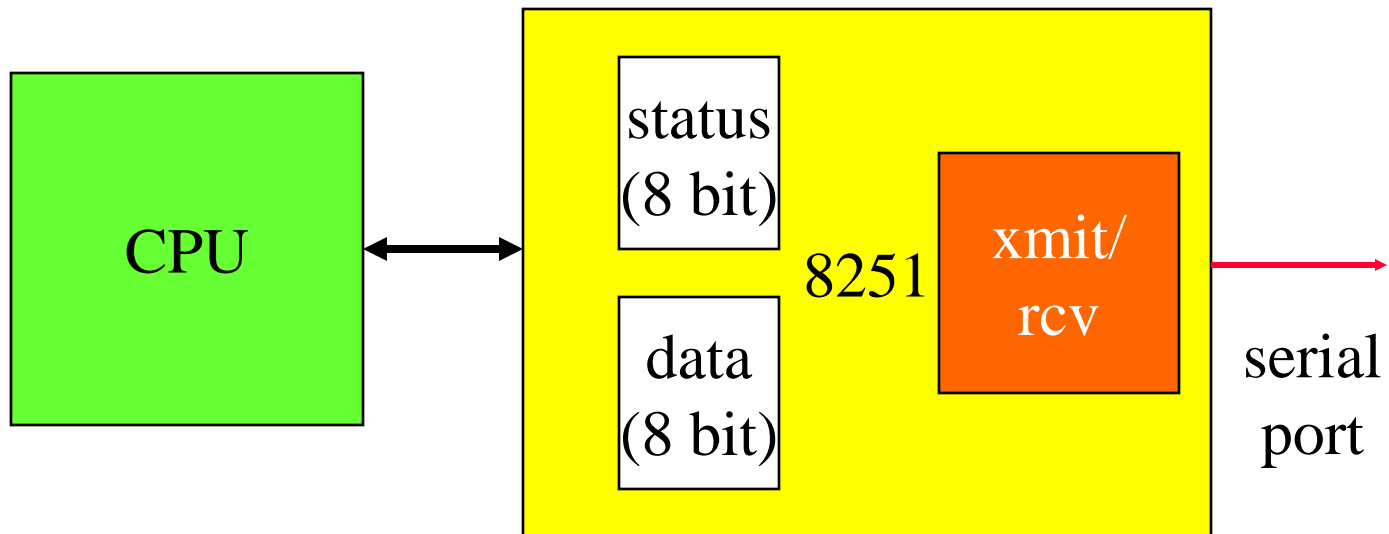
# Sample points

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

no  
char



# 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) { /* read */  
    return *location; }
```

```
void poke(char *location, char newval) { /* write */  
    (*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 or busy-wait */
    current_char++;
}
/* busy-wait function checks the device status until it changes
to 0 */
```

# Simultaneous busy/wait input and output

```
#define IN_DATA      0x1000
#define IN_STATUS   0x0001
#define OUT_DATA    0x1100
#define OUT_STATUS  0x1101

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

# 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



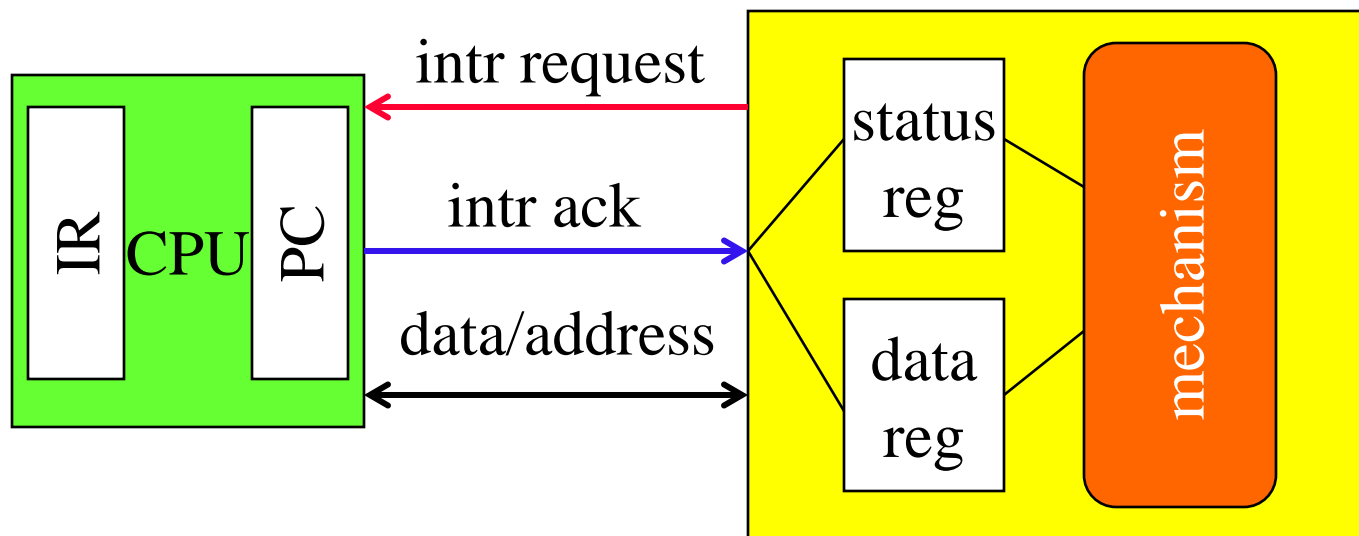
- ⌘ Based on subroutine call mechanism.
- ⌘ Interrupt forces next instruction to be a subroutine call to a predetermined location.
  - ☑ Return address is saved to **later** resume executing **foreground program**.

# Interrupt 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 (IRQ, FIQ)



# Interrupt interface



# Character I/O handlers

Example 3.4: use interrupts as a basic replacement for busy-wait

```
void input_handler() { /* interrupt handler */
    achar = peek(IN_DATA);
    gotchar = TRUE;
    poke(IN_STATUS,0);
}
```

```
void output_handler() { /* react for a char sent */
/* do nothing */ }
```

# Interrupt-driven main program

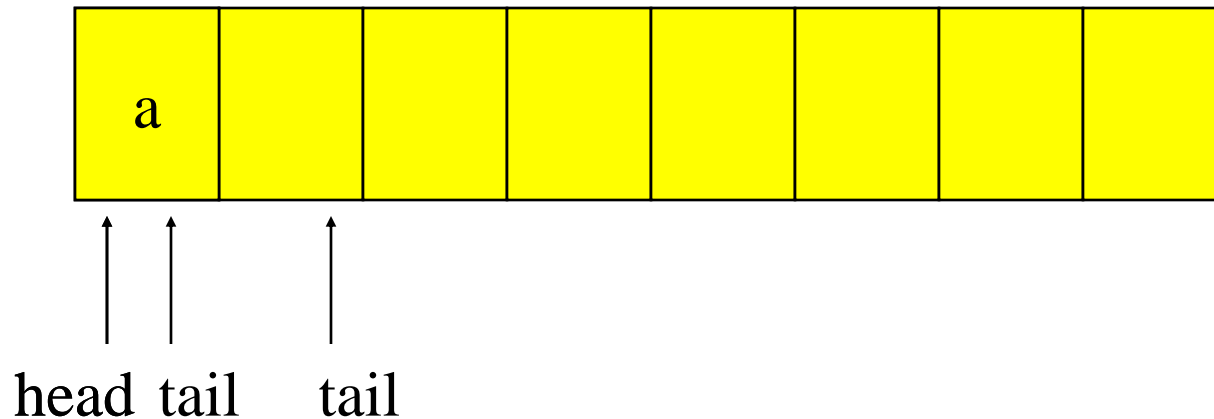
```
main() {  
    while (TRUE) {  
        if (gotchar) { /* check input status */  
            poke(OUT_DATA, achar);  
            poke(OUT_STATUS, 1);  
            gotchar = FALSE;  
        }  
    }  
}  
/* still the foreground program does not do useful work */
```

# Interrupt I/O with buffers

- ⌘ Use a queue between read and write routines to make them run independently.
- ⌘ a queue
  - ⊞ io\_buf: a character string
  - ⊞ buf\_start, buf\_end: head, tail
  - ⊞ status: set to 0 if io\_buf overflows (error)

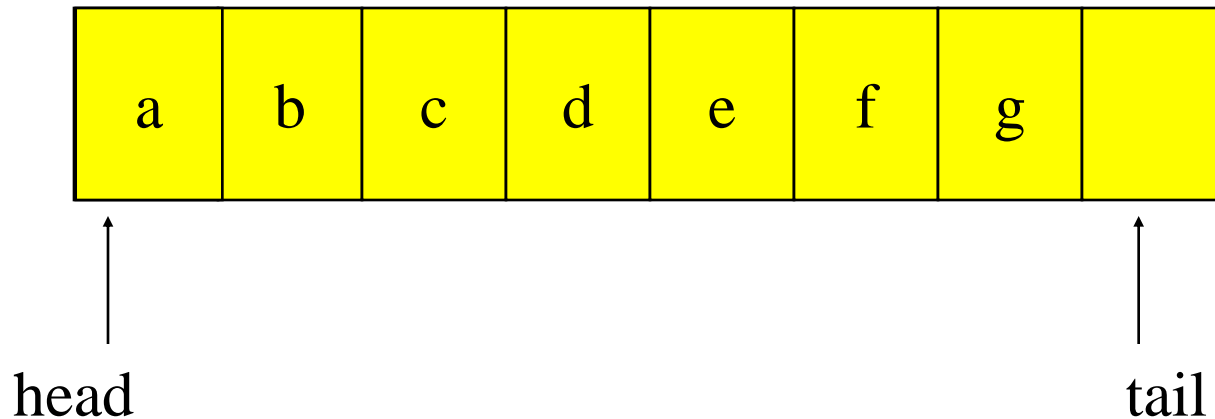
# Interrupt I/O with buffers

⌘ Queue for characters:



# Interrupt I/O with buffers

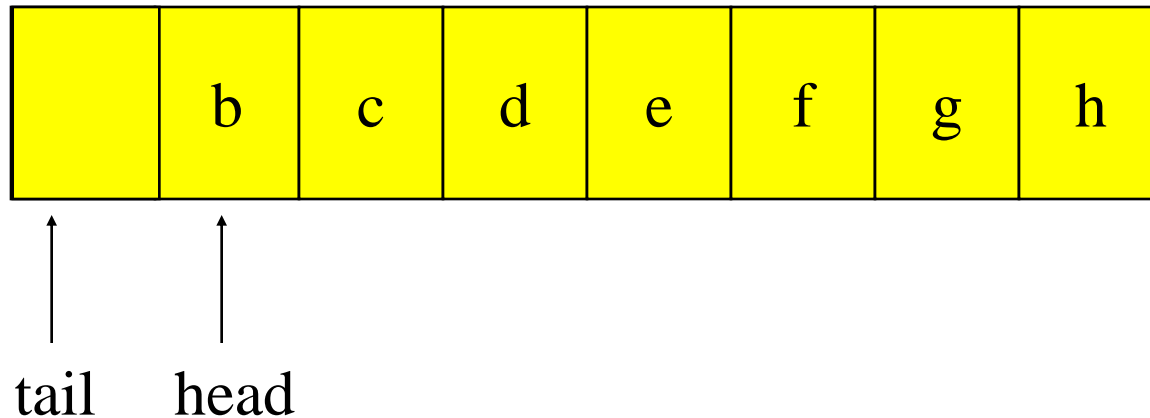
⌘ Queue for characters: full



$$\text{full} = ((\text{tail}+1) == \text{head})$$

# Interrupt I/O with buffers

⌘ Queue for characters:



# Buffer-based input handler

```
#define BUF_SIZE 8
char io_buf[BUF_SIZE];
int buf_head=0, buf_tail=0;
int error =0;
void empty_buffer() {
    buf_head == buf_tail;
}
void full_buffer() {
    (buf_tail+1) % BUF_SIZE == buf_head;
}
```



# Buffer-based input handler

```
int nchars () { /* return # of chars in the buffer */
    if (buf_tail >= buf_head) return buf_tail - buf_head;
    else return BUF_SIZE + buf_tail - buf_head;
}

void add_char(char achar) { /* add a char into the buffer */
    io_buf[buf_tail++] == achar;
    if (buf_tail == BUFF_SIZE) buf_tail = 0;
}

char remove_char() { /* take a char from the buffer */
    char achar;
    achar = io_buffer[buf_head++];
    if (buf_head == BUF_SIZE) buf_head = 0;
}
```

# Buffer-based input handler

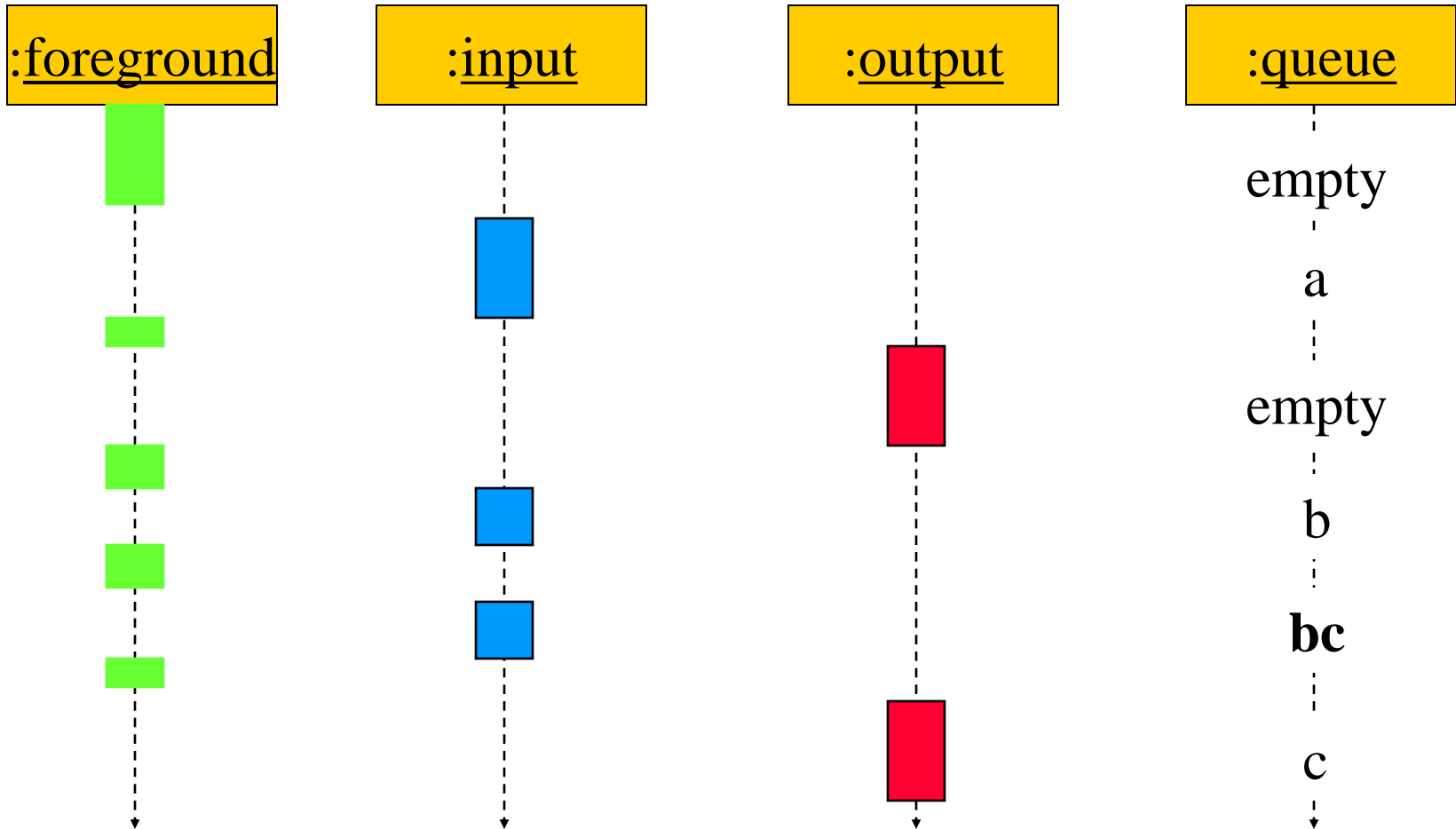
```
#define IN_DATA          0x1000
#define IN_STATUS       0x1001
void input_handler() {
    char achar;
    if (full_buffer()) error = 1;
    else { achar = peek(IN_DATA); add_char(achar); }
    poke(IN_STATUS,0);
    if (nchars == 1) {
        /* the buffer was empty, start a new output action by itself */
        poke(OUT_DATA,remove_char());
        poke(OUT_STATUS,1);
    }
}
```

# Buffer-based output handler

```
#define OUT_DATA      0x1100
#define OUT_STATUS   0x1101
void output_handler() {
    if (!empty_buffer()) {
        poke(OUT_DATA,remove_char());
        poke(OUT_STATUS,1); }
}

main() {
/* free to do useful work */
/* do nothing for io operations */
}
```

# I/O sequence diagram



# Interrupts



- ⌘ Interrupts allow a lot of concurrency, which can make efficient use of the CPU.
- ⌘ An interrupt can occur at any time
- ⌘ What if you forget to change registers?
  - ☑ Foreground program can exhibit mysterious bugs.
  - ☑ Bugs will be hard to repeat---depend on interrupt timing.

# CPU checks Interrupts



- ⌘ At the beginning of execution of every instruction to be able to response quickly to service requests form the devices.
- ⌘ If an interrupt exists, CPU does not fetch the instruction pointed to by PC.
- ⌘ Instead CPU set PC to a predefined location, which is the beginning of the interrupt handling routine.
- ⌘ The starting address of the interrupt handler is usually given as a pointer rather than defining a fixed location for the handler.

# Calling convention using stack



- ⌘ The interrupt handler must return to the foreground program without disturbing its machine state.
- ⌘ The subroutine call mechanism is typically based on utilizing the stack.

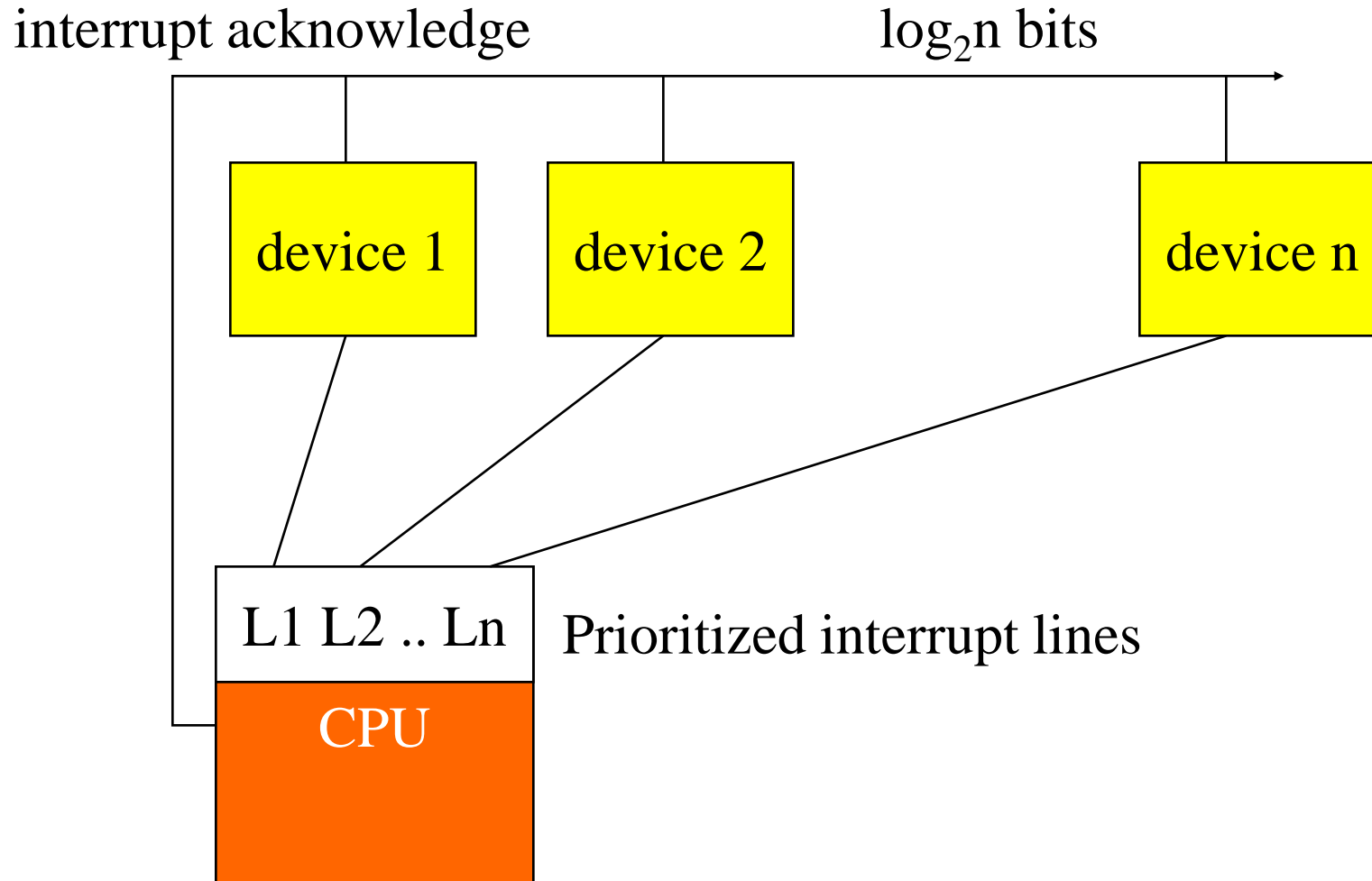
# Priorities and vectors



- ⌘ 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



# Vectored Interrupt flow

1. An interrupt occurs.
2. The ARM processor branches to the IRQ interrupt vector.
3. Read the VICADDRESS Register and branch to the interrupt service routine. This can be done using an LDR PC instruction. Reading the VICADDRESS Register updates the hardware priority register of the interrupt controller.
4. Stack the workspace so that IRQ interrupts can be re-enabled.
5. Enable the IRQ interrupts on the processor so that a higher priority can be serviced.

# Vectored Interrupt flow

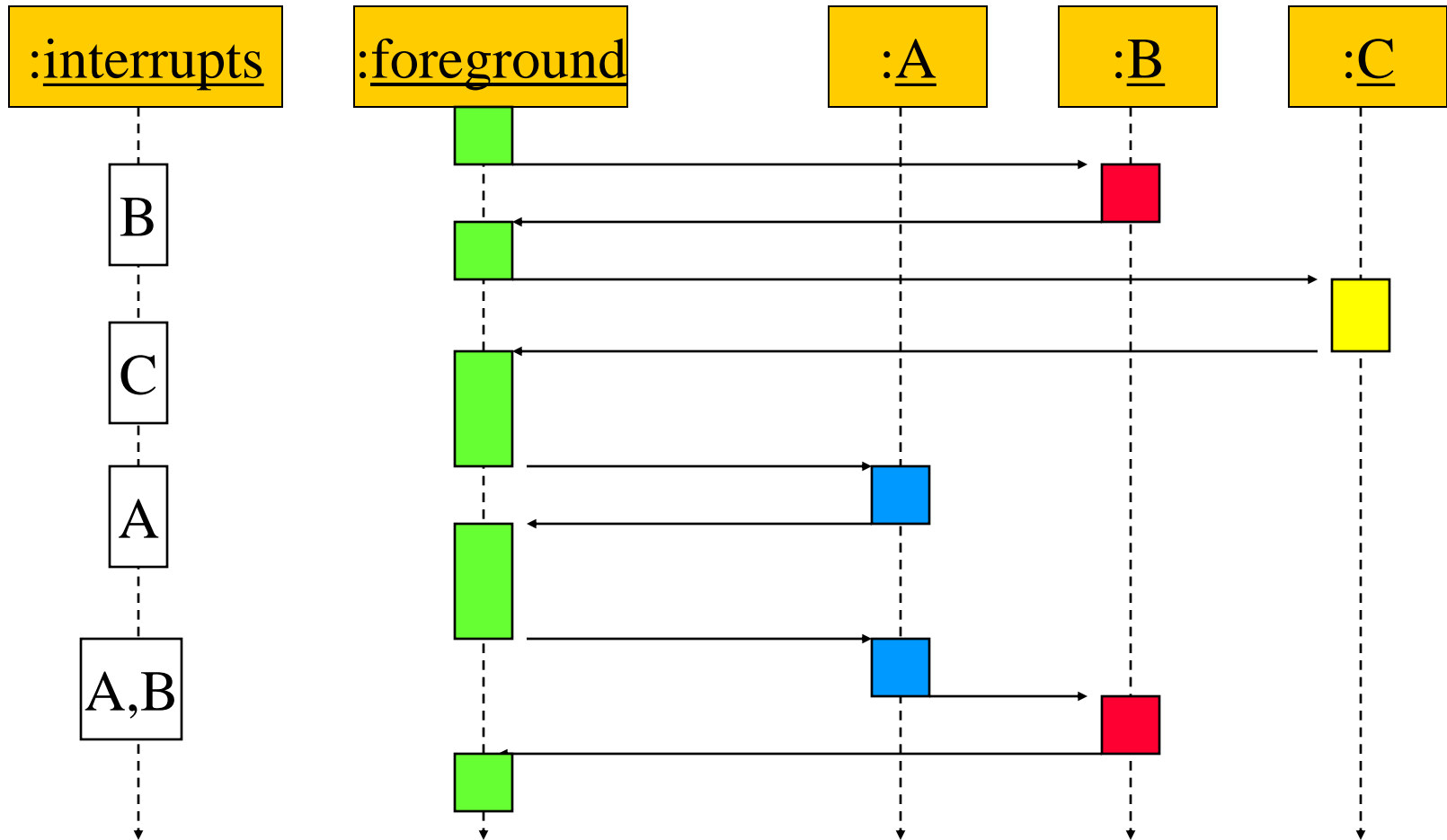


6. Execute the *Interrupt Service Routine* (ISR).
7. Clear the requesting interrupt in the peripheral, or write to the VICSOFTINTCLEAR Register if the request was generated by a software interrupt.
8. Disable the interrupts on the processor and restore the workspace.
9. Write to the VICADDRESS Register. This clears the respective interrupt in the internal interrupt priority hardware.
10. Return from the interrupt. This re-enables the interrupts.

# Interrupt prioritization

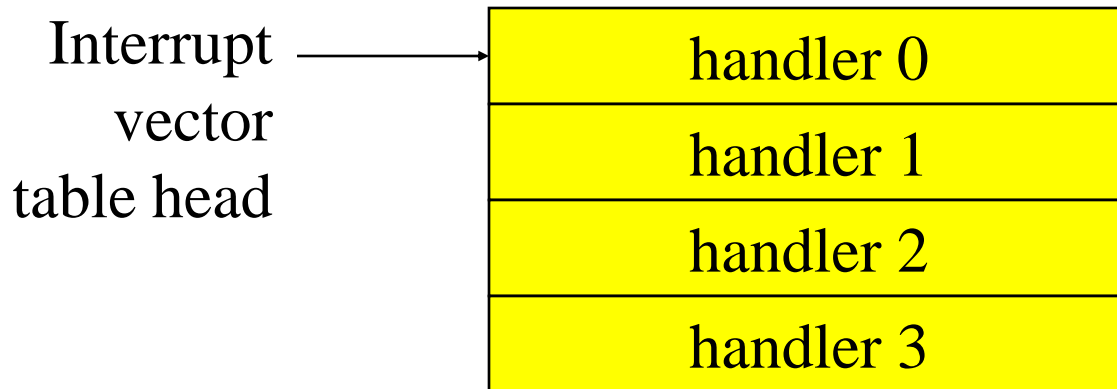
- ⌘ **Masking**: interrupt with priority lower than current priority is not recognized until pending interrupt is complete.
- ⌘ **Non-maskable interrupt (NMI)**: highest-priority, never masked.
  - ☑ Often used for power-down.

# Example: Prioritized I/O

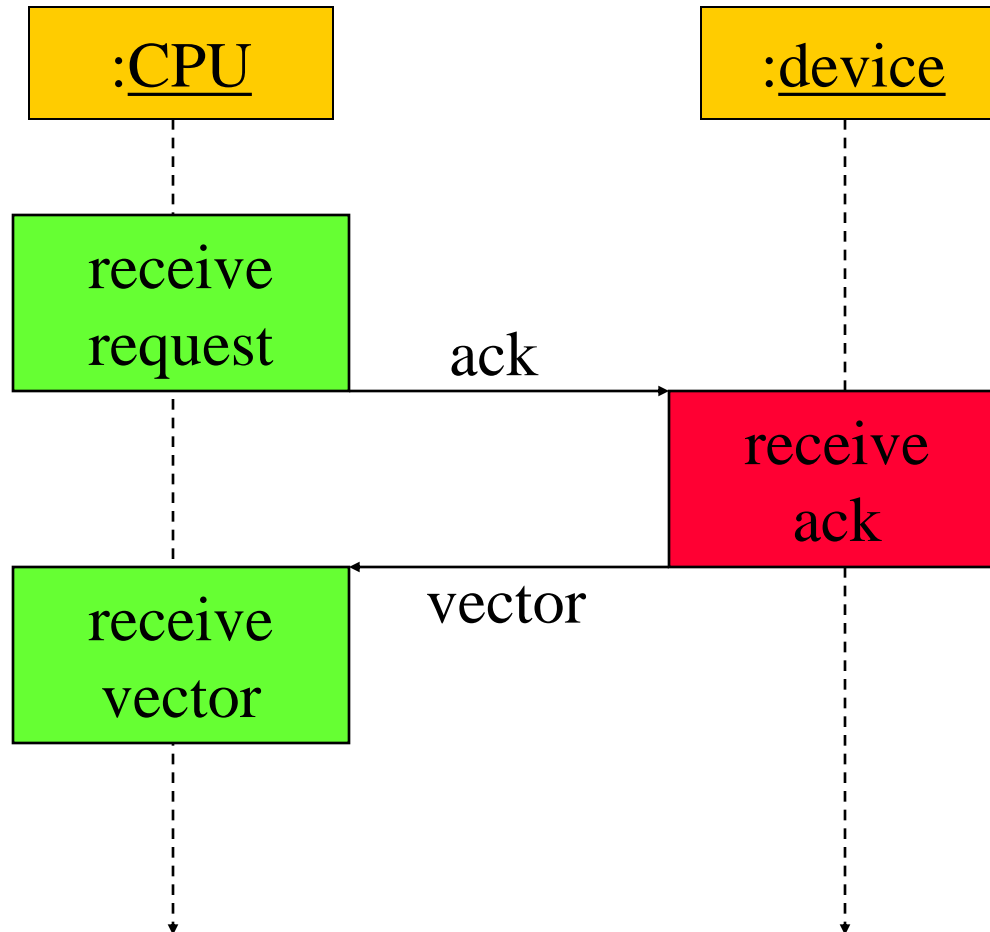


# Interrupt vectors

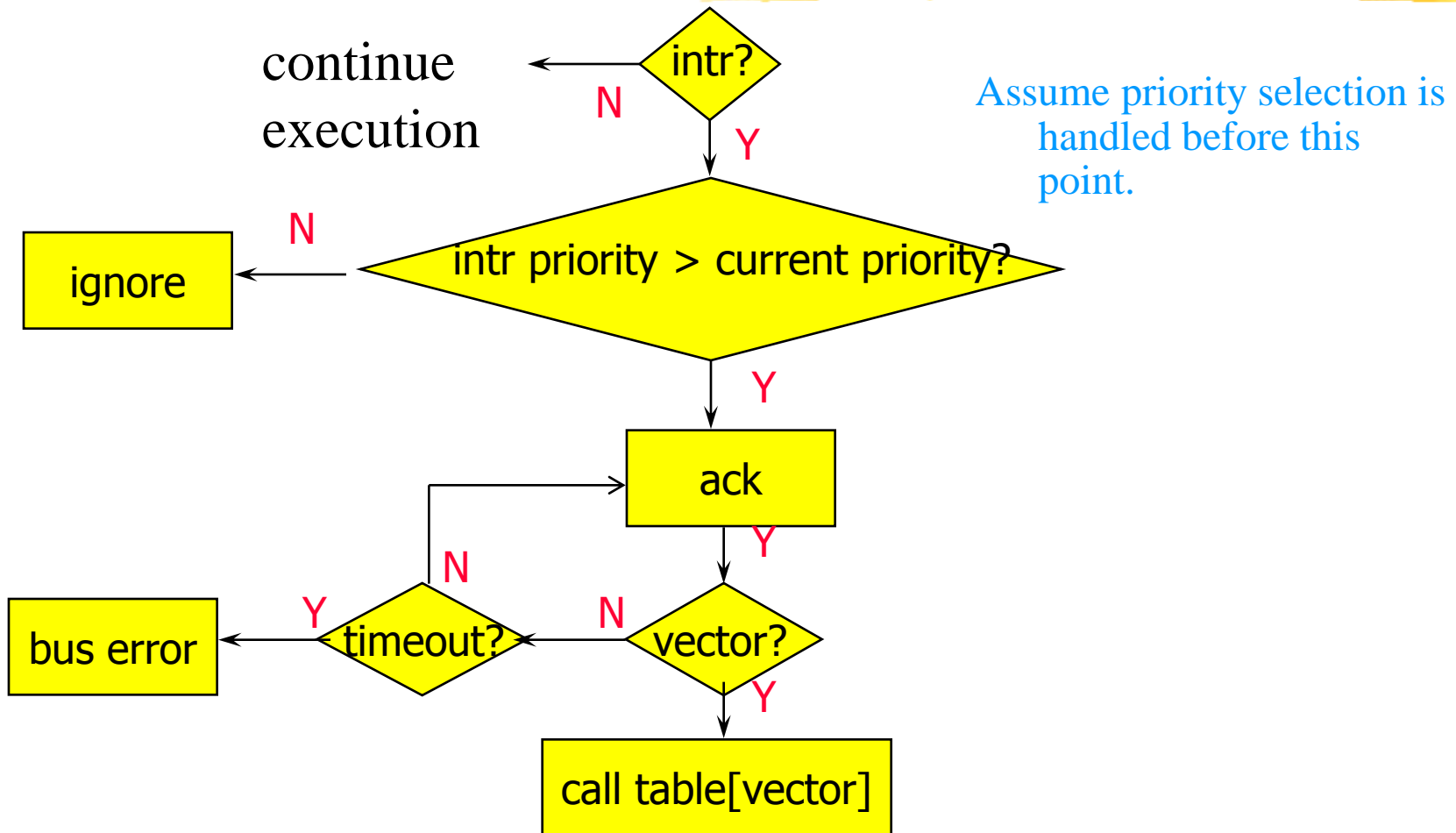
- ⌘ Allow different devices to be handled by different code.
- ⌘ Interrupt vector table:



# 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 = nzcvqjIft\_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 is 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 form 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}<sup>^</sup>                ; restore context and return

⌘ <sup>^</sup> 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

