IN4073
Embedded Real-Time Systems

X32 Microcontroller Soft core

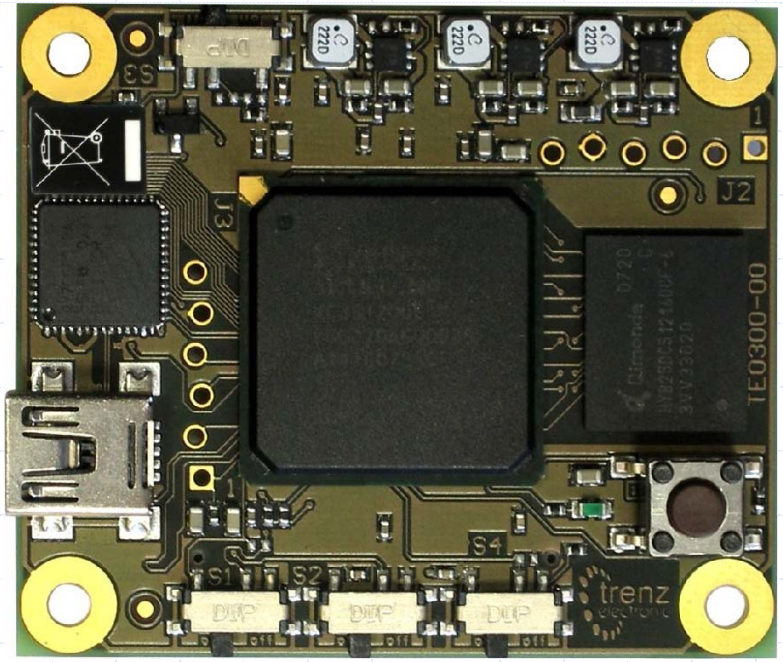
FGPAs vs Standard Microcontrollers

- 8051, ARM, .., core architectures
- AD, Atmel, Dalas, Intel, MC (PIC), NS, ST, TI, Zilog
- ASICs with core, e.g., Cypress
- Can be expensive, peripheral set is “hard-wired”

- FPGAs are a flexible, low-cost “bread-board” to experiment with HW using SW (VHDL) instead of ICs and wires
- More expensive FPGA also have hard cores: often best of both worlds
- TE0300 Board: $100 (1600k gates, no hard core)

TE0300 FPGA + Memory

- Create your own core and set of peripherals that communicate via FPGA I/O lines (UARTS, LEDS)
  - UART (PC Link, chars)
  - UART (QR Link, frames)
  - LEDS (8)
  - Timer
  - Clock
  - Filters
  - ...

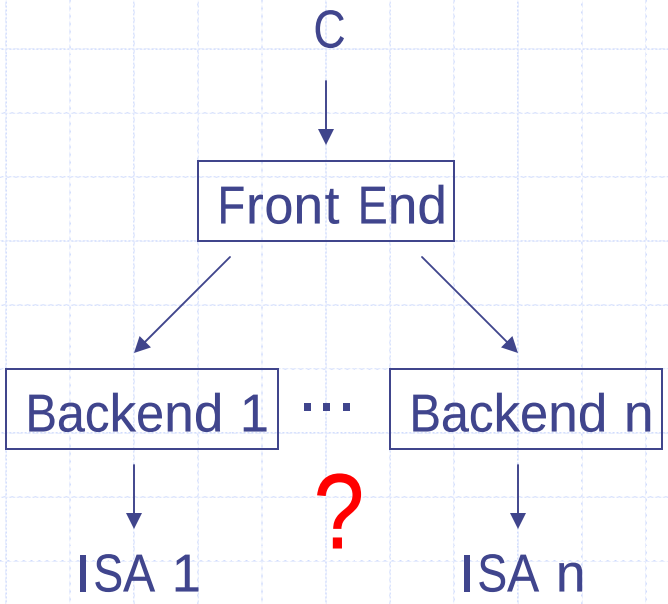


Turn FPGA into a Microcontroller

- high-speed designs in HW (e.g., VHDL)
  - XS3E-1600 clock 100 MHz, 1600k gates
- low-speed designs in SW (e.g., C)
  - Need processor core that communicates with VHDL devices
- no hard core (only in more expensive FPGAs)
- so need soft core to execute programs
- academia: **zero**-cost public domain VHDL soft core
- experience with free 8-bit soft core (6502) + tool chain (CC65): not so good
- so we built our own (well, Sijmen Woutersen did)

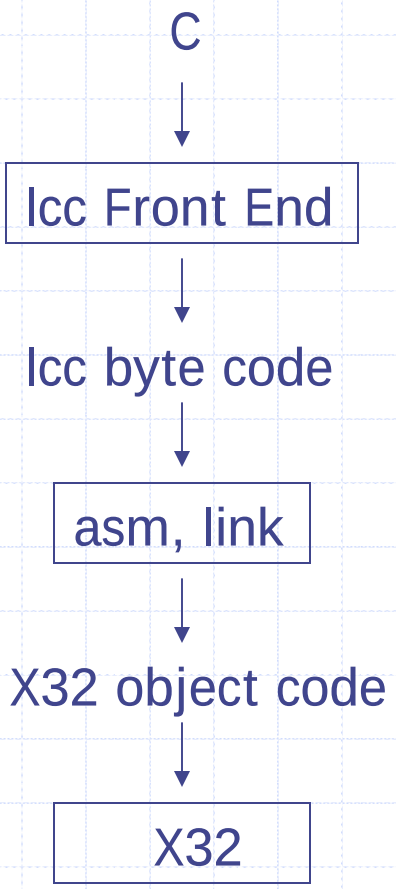
X32 Approach (1)

- no binary compatibility issues so choice of instruction set architecture (ISA) basically free
- but we don't want to build (i.e., retarget) an ANSI C compiler to some ISA that we first need to architect given the specific FPGA properties
- we just want simplicity, no big-time performance optimization project

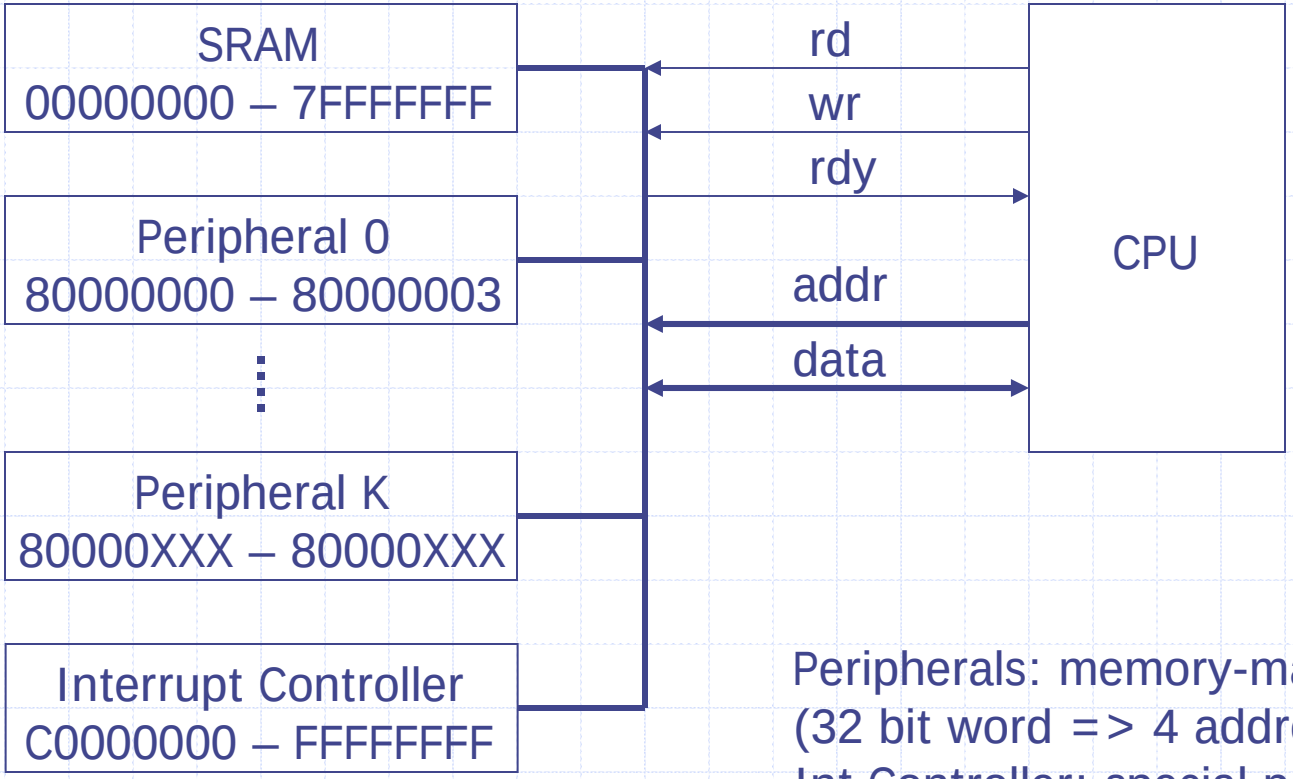


X32 Approach (2)

- So we took **`lcc`** (ANSI C)
- But we simply took the intermediate representation (byte code) as target ISA
- So compiler is already done
- Soft core = just writing an ISA interpreter in VHDL
- Experimental 32 bit arch: “X32”
- Top-down approach: X32 + compiler took 1 MSc project!
- Simple, but only 2 MIPS! (mem bound stack ISA!)

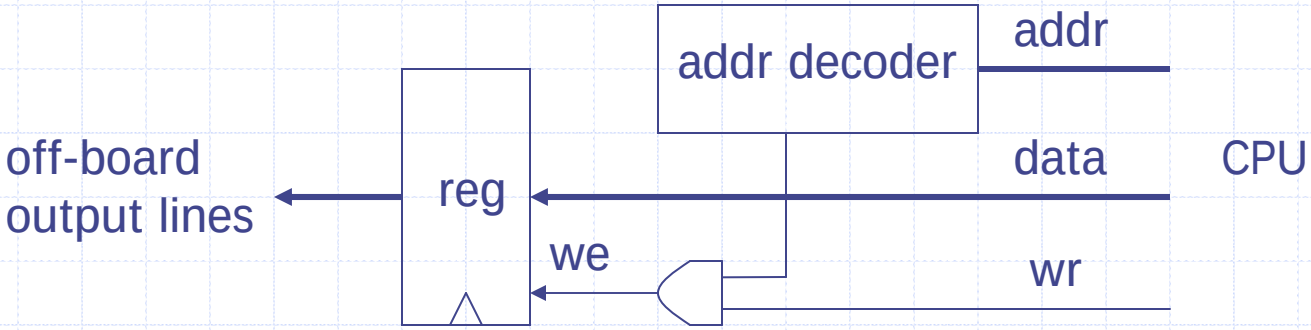


X32: Memory Map



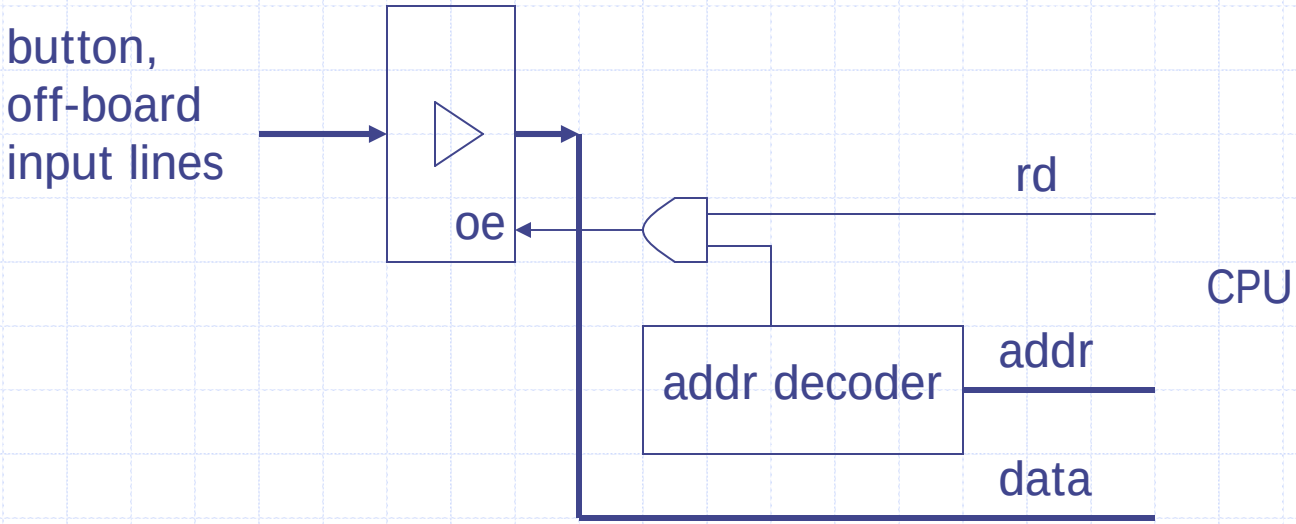
Peripherals: memory-mapped I/O
(32 bit word => 4 addresses)
Int.Controller: special peripheral
(occupies larger address space)

X32 Peripheral: Simple Output



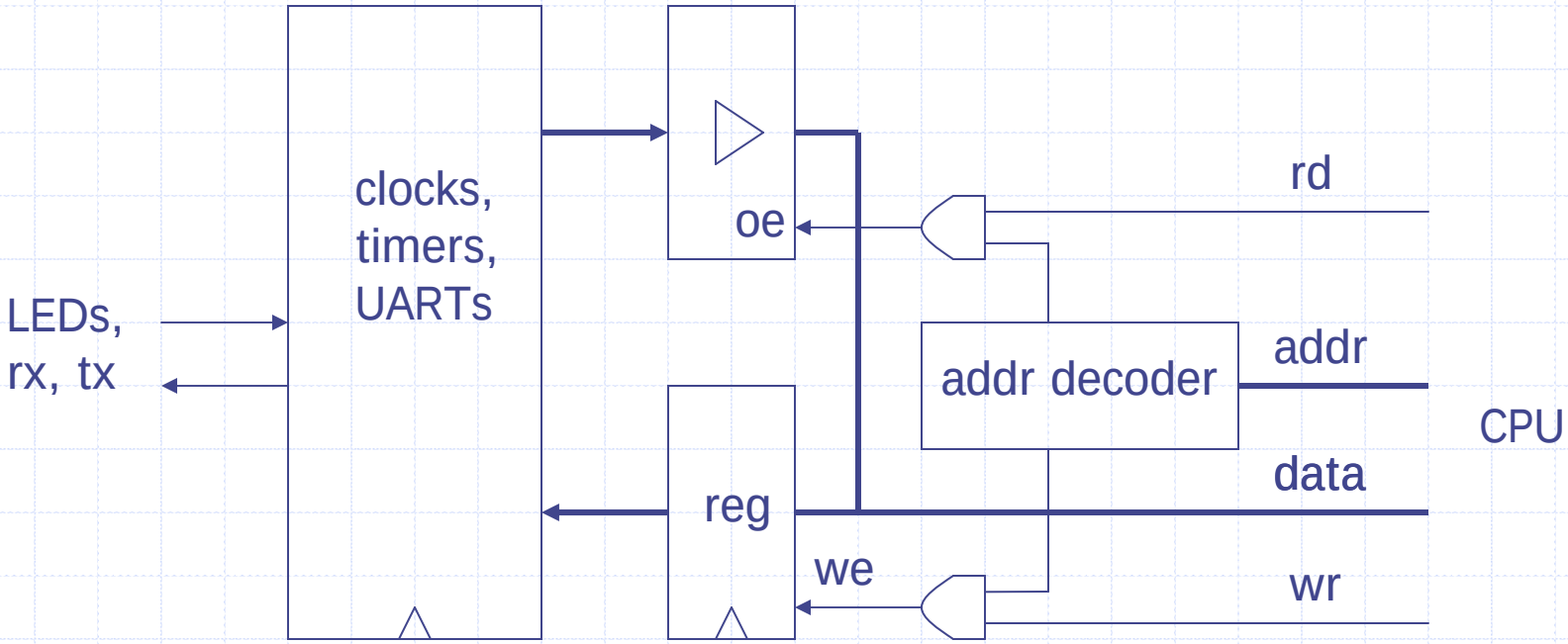
NOTE: all circuits are VHDL

X32 Peripheral: Simple Input



NOTE: all circuits are VHDL

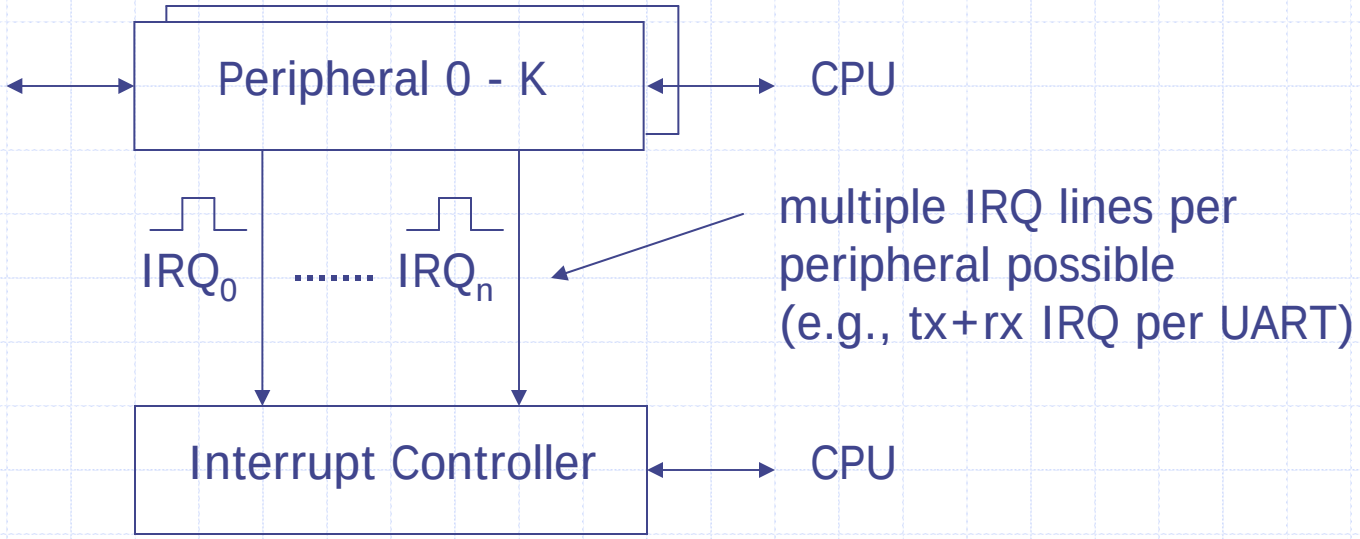
X32 Peripheral: Complex Devices



NOTE: all circuits are VHDL

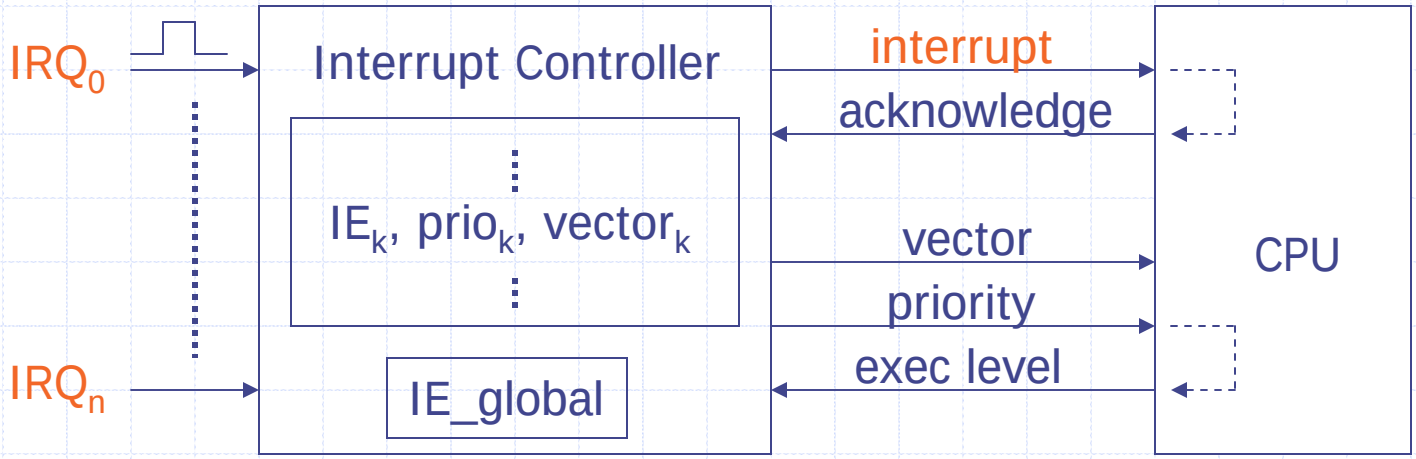
X32 Peripheral Interrupts

buttons,
LEDs,
switches,
timers,
RS232,
off-board
I/O ports

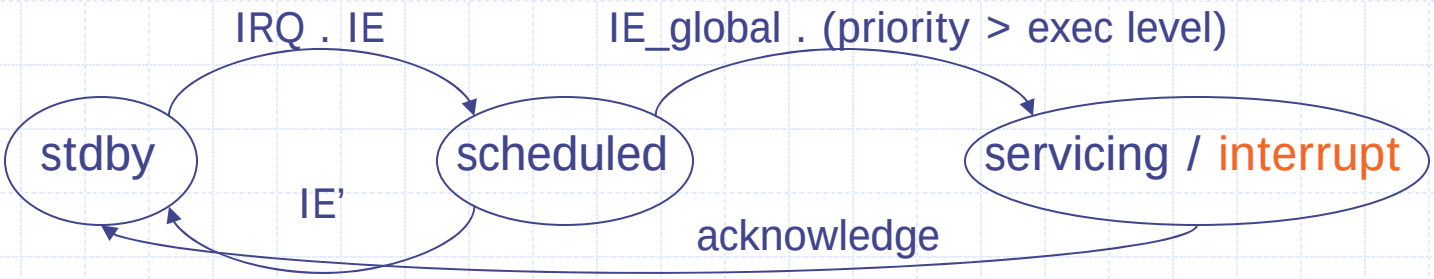


peripherals used at the lab:
buttons, LEDs, UART (RS232 interfaces to PC and to QR)

X32: Interrupt Controller



FSM for each IRQ:



X32: Interrupt Sources

- CPU: divide-by-0, overflow (disable unless needed!)
- Buttons, switches, I/O ports
  - positive AND negative edge-triggered
- Timers: counter value > threshold reg
- UART: rx buffer char received / tx buffer empty
- Maxon motor decode error (in2305)
- QR link data frame (7 readings) received (in4073)

X32: Interrupt Features

- Each peripheral can have multiple IRQ lines
- Local interrupt enable/disable
- IRQ latch within IC *cleared* on local interrupt disable
- Each IRQ can have different priority and ISR vector
- ISRs are serviced in order of priority
- Interrupts NOT automatically disabled: ISR *preemption*!
- Only higher-priority IRQs preempt current priority IRQ
- Global interrupt enable/disable
- Pending interrupt by IC *NOT* cleared on global disable

X32: Software

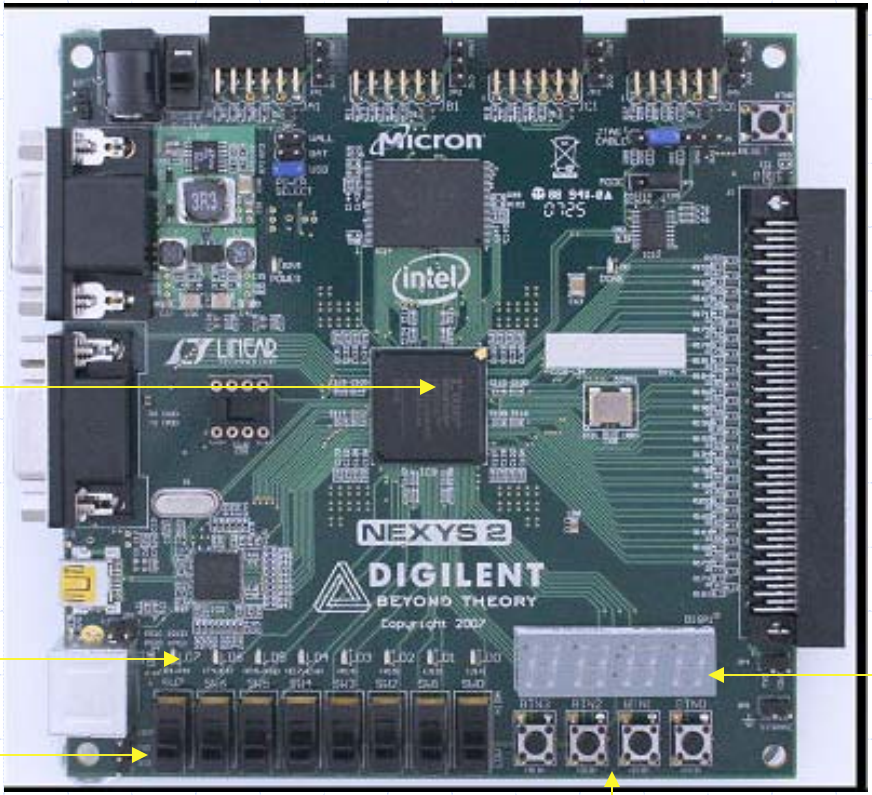
- ISA: lcc + X32-specifics:
  - {**LOAD**|**STORE**} **X** met **X** ∈ {**SP**, **FP**, **AP**, **EL**, ..}
- header file **`x32.h`**
- library functions
  - console I/O (**`getchar`**/**`putchar`**)
  - strings
  - **`printf`**
  - **`setjmp`**/**`longjmp`** (for compatibility reasons)
  - ..
- **No** HW floating-point support (SW lib)

X32: Memory-mapped I/O in C

```
#define ADDR_PERIPHERALS 0x800000000
int *peripherals = (int *) ADDR_PERIPHERALS;

#define PERIPHERAL_LEDS 0x07 // LEDs are 8th register

int    main(void)
{
       // write to 0x80000000 + offset of 7 ints
       // is 0x80000000 + 0x1c = 0x8000001c:

       peripherals[PERIPHERAL_LEDS] = 0x77;
       return(0);
}
```

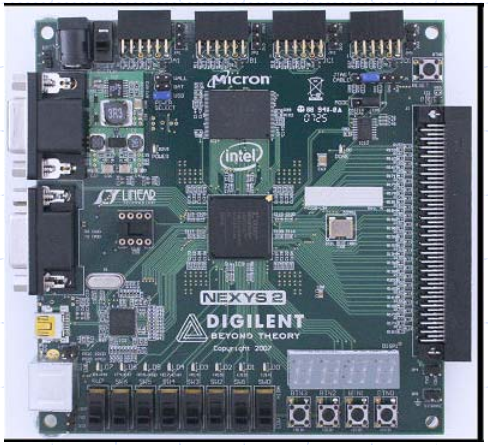
X32: Sample Project

X32 Demo Board:



X32: Sample Project

```
#include <x32.h>
#define X32_leds = peripherals[PERIPHERAL_LEDS]
#define X32_buttons = ...
#define X32_display = ..
#define X32_clock =

int    main(void)
{
       printf("Hello World!\r\n");
       while (1) {
              X32_display = X32_clock;
              X32_leds = X32_buttons;
              if (X32_buttons == 0x09)
                     break;
       }
}
```

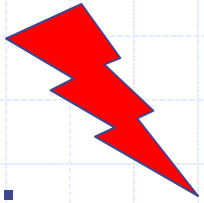


X32 Site (Free Downloads)

http://x32.ewi.tudelft.nl/

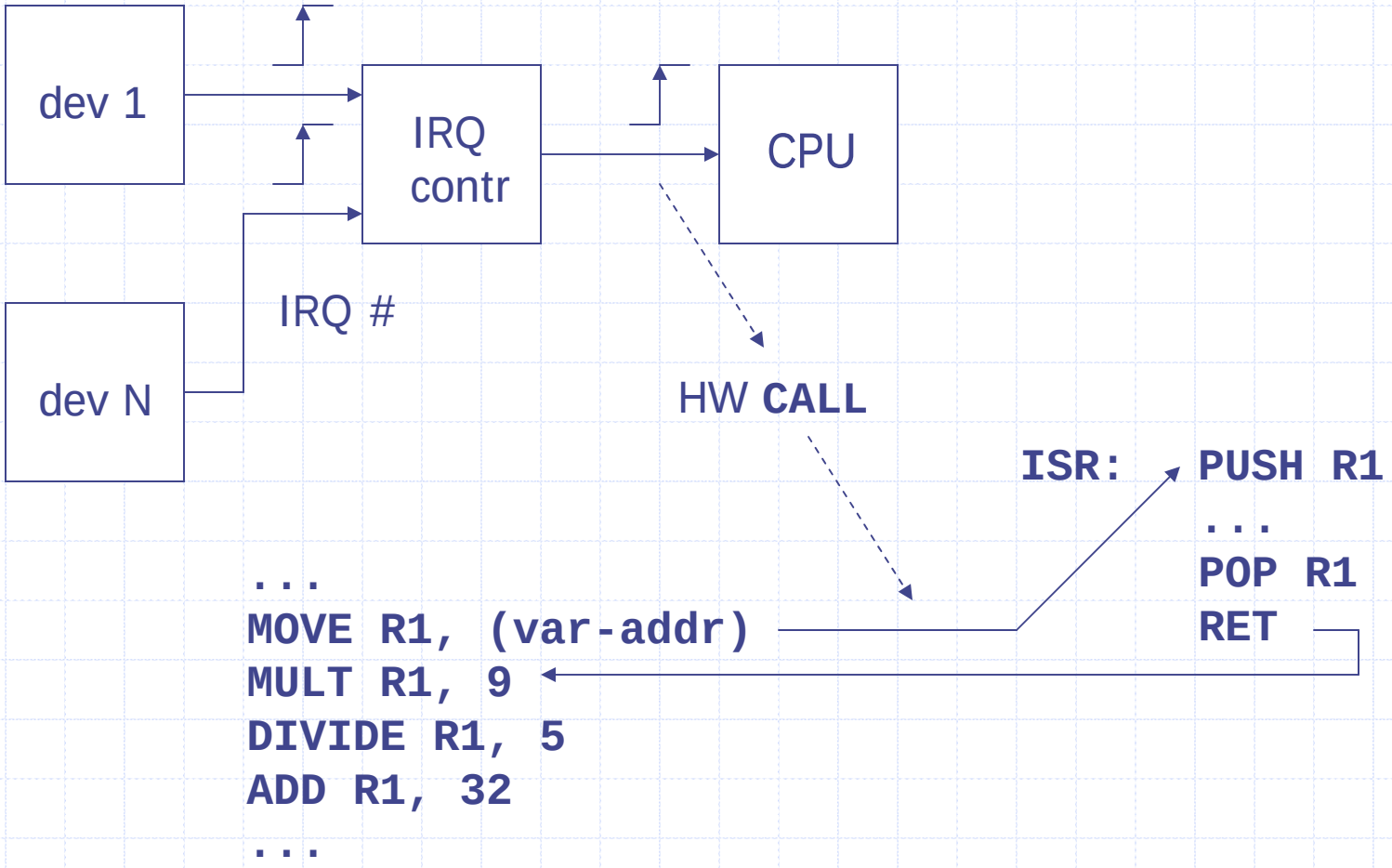
X32: Demo

Demo ..

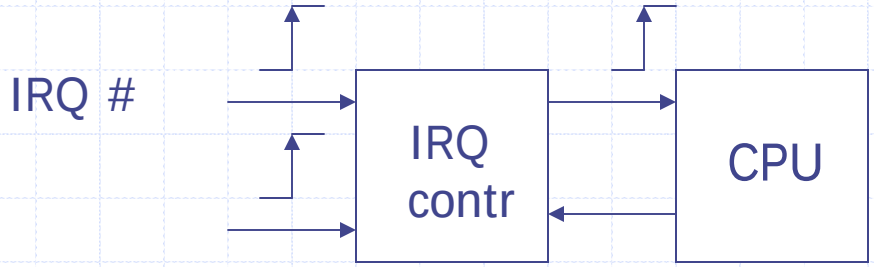


(x32_projects.tgz: leds.c)

Interrupts: Principle



X32 Interrupts



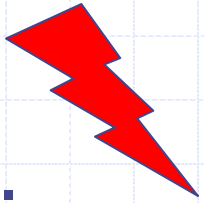
- IRQ controller preprocesses multiple IRQ's
- Each device: (IRQ #, priority)
- Vectored IRQ
- Interrupts NOT disabled
- Automatic ISR preemption if prio IRQ > prio current ISR
- Normal call saves context -> no **`interrupt`** keyword

Example: ISR version of hello.c

```
#define IRQ_BUTTONS = INTERRUPT_BUTTONS

void   isr_buttons(void) {
       X32_leds = X32_buttons;
       if (X32_buttons == 0x09) done = 1;
}

void   main(void) {
       SET_INTERRUPT_VECTOR(IRQ_BUTTONS),&isr_buttons);
       SET_INTERRUPT_PRIORITY(IRQ_BUTTONS),10);
       ENABLE_INTERRUPT(IRQ_BUTTONS);
       ENABLE_INTERRUPT(INTERRUPT_GLOBAL);
       printf("Hello World!\r\n");
       while (! done) X32_disp = X32_clock;
       DISABLE_INTERRUPTS(INTERRUPT_GLOBAL);
}
```

X32: Demo

Demo ..

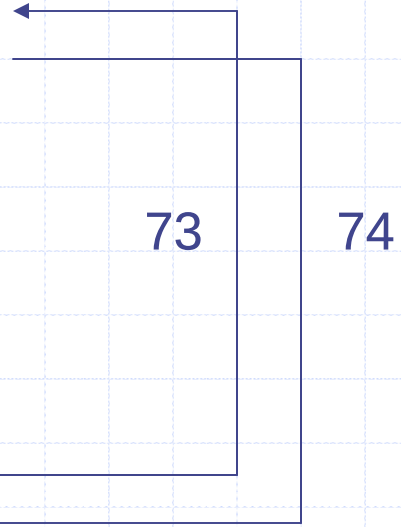


(x32_projects.tgz, console.c)

Interrupts: Data Sharing Problem

```
void   isr_read_temps(void)
{
       iTemp[0] = peripherals[..];
       iTemp[1] = peripherals[..];
}

void   main(void)
{
       ...
       while (TRUE) {
            tmp0 = iTemp[0];
            tmp1 = iTemp[1];
            if (tmp0 != tmp1)
                  panic();
       }
}
```

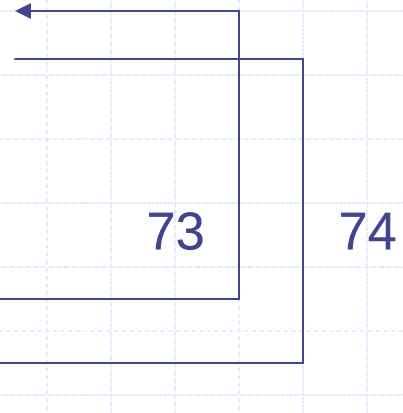
NOT ATOMIC!



Interrupts: Data Sharing Problem

- single C expressions usually aren't atomic either ..:

```
void   isr_read_temps(void)
{
       iTemp[0] = peripherals[..];
       iTemp[1] = peripherals[..];
}

void   main(void)
{
       ...
       while (TRUE) {
NOT ATOMIC!    if (iTemp[0] != iTemp[1])
                      panic();

       }
}
```



Solutions (1)

- Disable interrupts that trigger those ISRs that share the data

```
...
while (TRUE) {
        !! DISABLE INT
        tmp0 = iTemp[0];
        tmp1 = iTemp[1];
        !! ENABLE INT
        if (tmp0 != tmp1)
                panic();
}
```

The critical section is now atomic

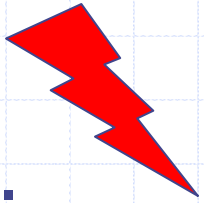
- Use semaphores to protect critical section (discussed in a later lecture)

Solutions (2)

- Don’t disable interrupts but write ingenious code, e.g., involving alternating data buffers or even queues such that ISR and (main) code never access the same data
- Why? code becomes error-prone and (too) hard to read
- Rule: keep it simple, just disable interrupts, as long as you adhere to:
  - keep the critical sections SHORT
  - keep the ISRs SHORT (to minimize latency, see later)

X32: Demo

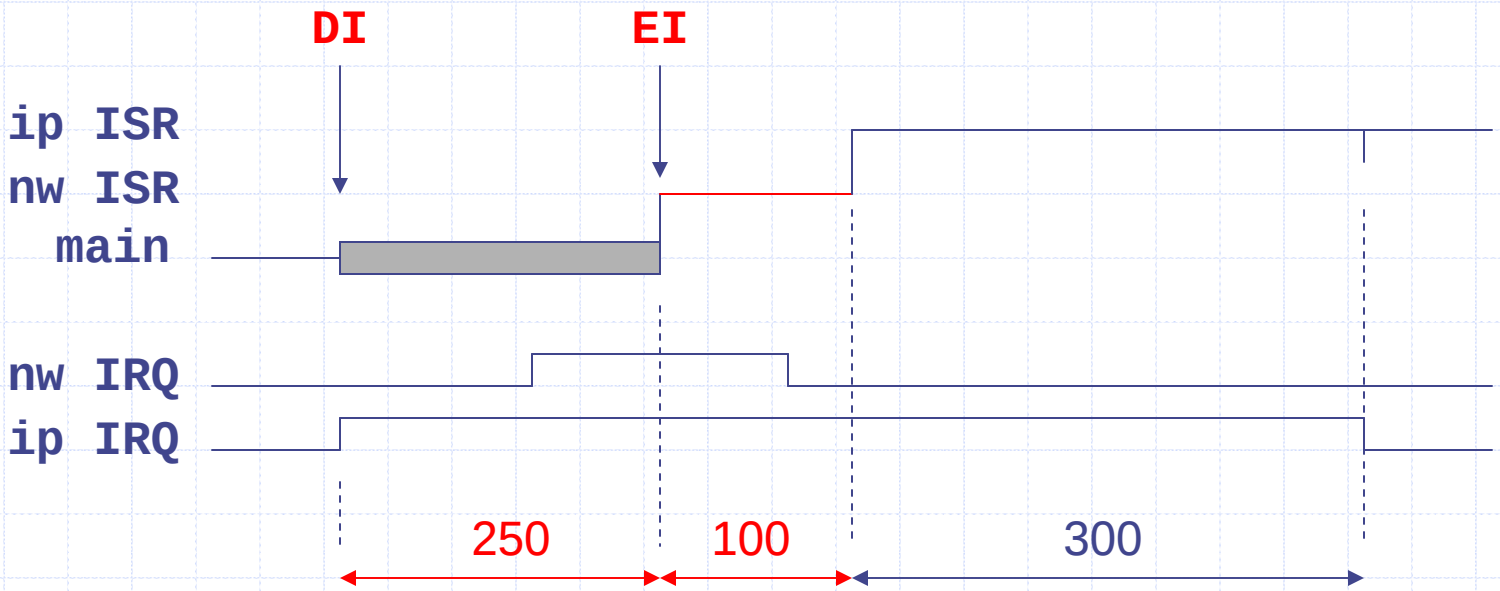
Demo ..



(x32_projects.tgz, critical.c)

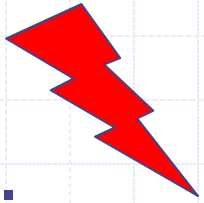
Interrupt Latency

- Quick response to IRQ may be needed
- Depends on previous rules: worst-case latency = t_disabled + t_higher prio ISRs + t_myISR



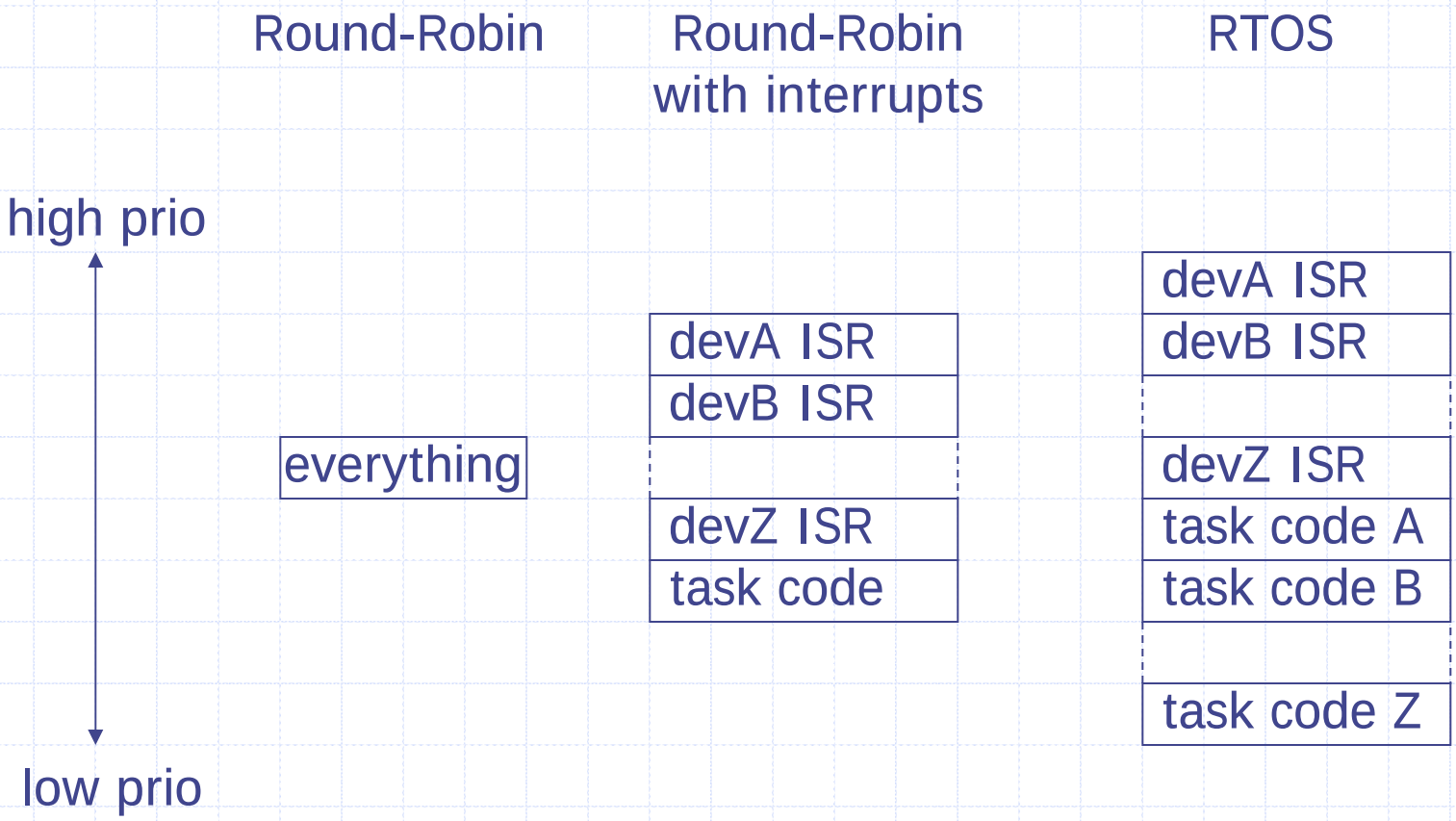
X32: Demo

Demo ..



(x32_projects.tgz: timing.c)

Embedded SW Architectures



RTOS: Primary Motivation

- Task switching with *priority preemption*
- Additional services (semaphores, timers, queues, ..)
- Better code!
  - Having interrupt facilities, one doesn't always need to throw a full-fledged RTOS at a problem
  - However, in vast majority of the cases the code becomes (1) cleaner, (2) much more readable by another programmer, (3) less buggy, (4) more efficient
- The price: small run-time overhead and small footprint

Task Switching

- Task switching = switching from current context (PC, stack, registers) to another context
- Context = thread identity, hence aka multithreading
- Need two constructs:
  - initialize a context
  - switch to a context
- Often used: **`setjmp`**/**`longjmp`**
- We use X32 **`init_stack`**/**`context_switch`**

Simple Example (X32)

```
void   **thread_main; **thread_a;
void   *stack_a[1024];

int    main(void)
{
       thread_a = init_stack(stack_a, task_a);
       printf("now in thread_main\n");
       context_switch(thread_a,&thread_main);
       printf("back in main_thread\n");
}

void   task_a(void)
{
       print("now in thread_a\n");
       context_switch(thread_main,&thread_a);
}
```

Time Slicing Example (1)

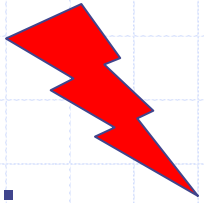
```
void   **thread_main; **thread_a;
void   *stack_a[1024];
int    thread_id;

void   isr_timer(void)
{
       if (thread_id == 0) {
              thread_id = 1;
              context_switch(thread_a,&thread_main);
       }
       else {
              thread_id = 0;
              context_switch(thread_main,&thread_a);
       }
}
```

Time Slicing Example (2)

```
int     main(void)
{
        thread_a = init_stack(stack_a, task_a);
        thread_id = 0; // now in main

        !! set timer to interrupt every 5 ms

        while (TRUE)
                print(now in thread_main\n);
}

void    task_a(void)
{
        while (TRUE)
                print("now in thread_a\n");
}
```

X32: Demo

Demo ..



(x32_projects.tgz: slicing.c)