Interrupts (Example)

Prof. John McLeod

ECE9047/9407, Winter 2022

Microcontrollers often have to send and receive data from a wide variety of peripherals and communications networks. Each of these operates at its own speed, which is often quite a bit slower than the microcontroller CPU. As the number of peripherals and communications networks grows, sequentially check each for completion becomes unmanageable. This is where interrupts come in: an interrupt enabled processor can "mind its own business" until a piece of hardware tells it that it is ready.

Example of Interrupt Programming

A worked example of programming interrupts on the A9 is given. A further example of using interrupts on the Nios II is also given; this second case is just for comparison. The Nios II has a much simpler interrupt interface than ARM processors — most non-ARM microcontrollers have simpler interrupt interfaces.

This program performs the following:

- Uses an interrupt-enabled timer to count $0.5\,\mathrm{s}$ intervals.
- Displays this count in binary on the LED panel.
- Uses an interrupt-enabled push button to reset the count.

When reading through this program, remember the cool figure I previously presented that shows the program flow for enabling interrupts in an ARMv7-type CPU.

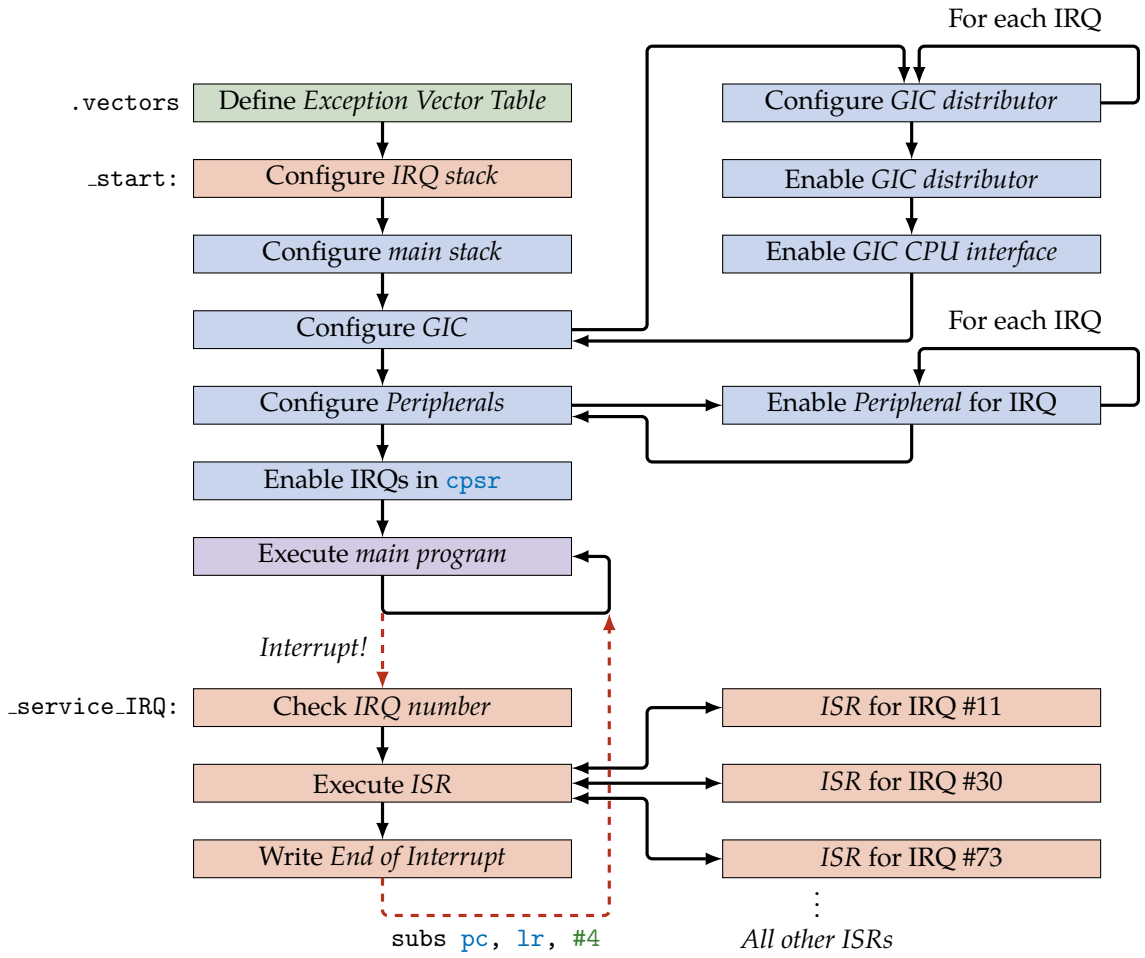

Figure 1: Program flow for enabling interrupts in an ARMv7-type CPU. Everything in blue is done in *supervisor mode*, everything in red is done in *IRQ mode*. Code in green are compiler-only instructions (i.e., to set the *exception vector table*), and the main program (in purple) could be executed in *supervisor mode* or *user mode*, as desired. All solid arrows indicate sequential flow or a branch to a normal subroutine. The red dashed arrows indicate a hardware-triggered branch according to the *exception vector table*, and an automatic change in the operating mode (to or from *IRQ mode*, as appropriate).

Setting up the Exception Vector Table

The very first few lines of code need to set up the exception vector table (EVT). In an ARM, this is always 8 words that contain the instructions for what the processor should do upon reset, aborted data fetch, interrupts (IRQ), fast interrupts (FIQ), etc.

```
@ first set up exception vector table
.section .vectors, "ax"
  b _start
  b evt_undef
  b evt_undef
  b evt_undef
  b evt_undef
  .word 0
  b service_irq
  b evt_undef
```

In this example, the ARM should go to `_start:` after being reset, and should run `service_irq:` if an IRQ is triggered. I have no intention of programming anything for the other exceptions, so they all go to `evt_undef:`. That is a dead loop.

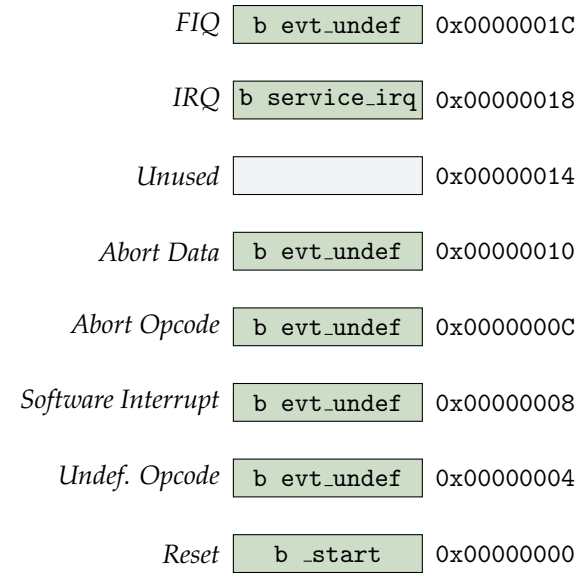

Figure 2: The ARM **exception vector table**. Each row contains an instruction, typically (as shown here) a branch to some subroutine. These instructions are executed whenever that exception is triggered.

Initializing the Program

The second step is to initialize the main program. Because the ARM has banked registers, one of the most important things to do is set the stack pointers. Otherwise the IRQ and SVC stack pointers will overwrite each other's data.

```
_start:
  @ change to IRQ mode
  mov r1, #0b11010010
  msr cpsr_c, r1
  @ set up stack pointer for IRQ mode
  ldr sp, =0xFFFFFFFC

  @ change back to supervisor mode
  mov r1, #0b11010011
  msr cpsr, r1
  @ set stack pointer for supervisor mode
  ldr sp, =0x3FFFFFFC

  @ call routine to configure the GIC
  bl config_gic
```

Here the code branches to the subroutine `config_gic:`, which handles the messy business of configuring the ARM generic interrupt controller (GIC). Fortunately, the code for `config_gic:` is provided by ARM (or Intel or Altera) so we can just copy that with some minor modifications.

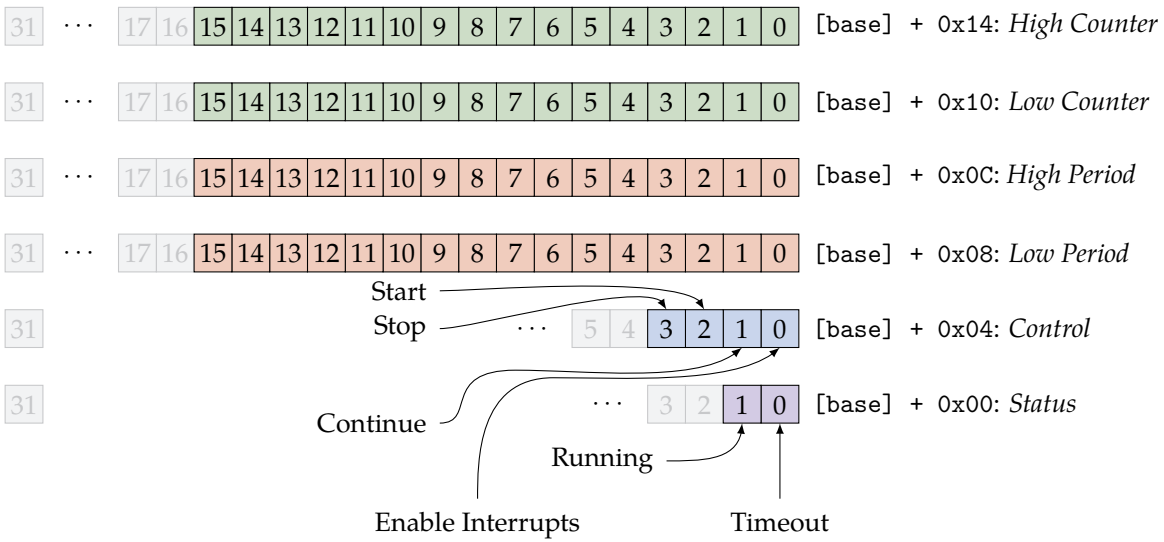

Figure 3: Structure of the interval timers on the DE1-SoC board. The base addresses are `0xFF202000` and `0xFF202020`. Note that although the timer is structured in words (32 bits), for legacy reasons at most only half-words (16 bits) are used for each part.

Proceeding with the code in order, the next thing to do is configure the timer for a $0.5\,\mathrm{s}$ continuous countdown with interrupts enabled.

- Remember the awkward structure of splitting a 32-bit count value into two 16-bit parts to write to the low and high period registers.

```
@ initialize timer, 0.5s count with interrupts
ldr r0, =0xFF202000
@ turn off timer
mov r1, #8
str r1, [r0, #4]
@ clear time outs
str r1, [r0]
@ set interval
```

```
ldr r1, =50000000
@ write low period
str r1, [r0, #8]
@ shift bits right by 16
lsr r1, #16
@ write high period
str r1, [r0, #12]
@ bit battern for count-down repeat with
@ interrupts for control register
mov r1, #7
@ start timer
str r1, [r0, #4]
```

Here I explicitly turn off and clear time outs while initializing the timer. I added this code while debugging the program — it didn't work the first time (surprise!) and I got frustrated with the timer continually running in the background. This code isn't strictly necessary.

Now we need to configure the push buttons for interrupts. This was not previously documented in my notes, but there is an **interrupt control register** two words (8 bytes) above the base address for the push buttons. Setting bit n to 1 in this register will enable interrupts on button n.

```
@ turn on button interrupts
ldr r0, =0xFF200050
@ I just want button(0) to have interrupts
mov r1, #1
str r1, [r0, #8]
```

As noted in the comments of the code above, I am only turning on interrupts for button 0. Use `mov r1, #15` rather than `mov r1, #1` to turn on interrupts for all buttons (as $15_{10} = 0b1111$).

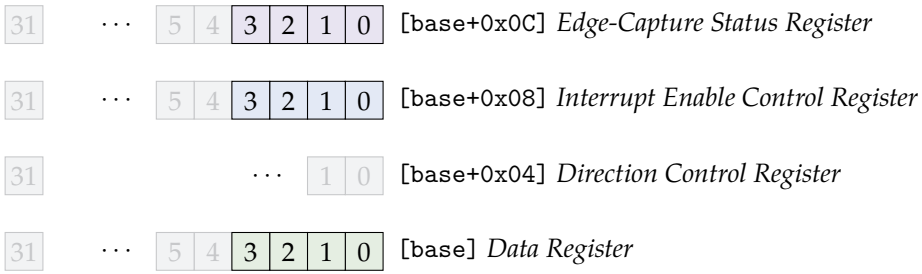

Figure 4: Full structure of the 4 push buttons at 0xFF200050 on the DE1-SoC. This structure is common to all GPIO ports, but the push button *Direction Control Register* is disabled as the buttons are always input.

Finally, we need to enable IRQ interrupts in the processor by clearing the appropriate bit in `cpsr`.

```
@ enable IRQ interrupts in the processor
mov r0, #0b01010011
msr cpsr_c, r0
```

The Main Program

The "main program" is just a dead loop — everything is handled by interrupts.

```
@ main program is a dead loop
idle:
  b idle
```

This is one of the main benefits of interrupt-driven programming: once you've gone through the hassle of setting up all the interrupts, there often isn't much left to do!

Exception Subroutines

Now for the subroutines required. The first is the dead loop for any undefined exceptions.

```
evt_undef:
  b evt_undef
```

Now I need to write the service routine for IRQs. This subroutine is automatically called any time a hardware interrupt is triggered. This subroutine determines *which* piece of hardware triggers the interrupt by checking IRQ # available in the ICCIAR register.

```
service_irq:
  push {r0 - r7, lr}

  @ Read the ICCIAR from the CPU Interface
  ldr r4, =0xFFFEC100
  ldr r5, [r4, #12]

  @ see if timer (IRQ #72)
  @ called interrupt
  cmp r5, #72
  beq timer_isr

  @ see if buttons (IRQ #73)
  @ called interrupt
  cmp r5, #73
  beq button_isr
```

```
exit_irq:
  @ clear end-of-interrupt register (ICCEOIR)
  str r5, [r4, #16]

  pop {r0 - r7, lr}
  @ return from IRQ mode
  subs pc, lr, #4
```

As written here, the interrupt service routine for the timer isn't really a subroutine, since after it runs it jumps to `exit_irq:` instead of using the link register.

```
timer_isr:
  @ timer base address
  ldr r0, =0xFF202000
  @ clear timeout
  mov r1, #1
  str r1, [r0]

  @ LED address
  ldr r0, =0xFF200000
  @ get current LED pattern
  ldr r1, [r0]
  @ add 1 to pattern
  add r1, #1
  @ write new pattern to LED
  str r1, [r0]

  b exit_irq
```

Similarly, the interrupt service routine for the push buttons again isn't really a subroutine. These *could* both be written as subroutines (i.e. branching with links, preserving state, returning using the link register) if desired, but I don't see the benefit. It was not previously documented in my notes, but there is also a **edge-capture status register** three words (12 bytes) above the base address for the push buttons. When bit n is set to 1, an interrupt occurred on that button. These bits need to be cleared (by writing 1 to that register) when the interrupt is handled.

```
button_isr:
  @ base address of push buttons
  ldr r0, =0xFF200050
  @ clear interrupt
  mov r1, #1
  str r1, [r0, #12]

  @ base address of LEDs
  ldr r0, =0xFF200000
  @ clear LED display
  mov r1, #0
  str r1, [r0]

  b exit_irq
```

Configuring the GIC

Now we can write the subroutines to configure the GIC. These are largely copy+pasted from ARM's documentation, the only thing I need to change is which IRQ # to enable.

```
.global config_gic
config_gic:
  push {lr}
  @ to configure an interrupt
  @  1. set target cpu in ICDIPTRn
  @  2. enable interrupt in ICDISERn

  @ timer is IRQ #72
  mov r0, #72
  @ bit mask for cpu0
  mov r1, #1
  @ subroutine to configure GIC registers
  bl config_interrupt

  @ buttons are IRQ #73
  mov r0, #73
  @ bit mask for cpu0
  mov r1, #1
  @ subroutine to configure GIC registers
  bl config_interrupt

  @configure the GIC CPU Interface
  @ base address of CPU Interface
```

```
ldr r0, =0xFFFEC100
@ enable all priority levels
ldr r1, =0xFFFF
@ write to priority register (ICCPMR)
str r1, [r0, #4]
@ enable interface control register (ICCICR)
mov r1, #1
str r1, [r0]

@ enable distributor control register (ICDDCR)
ldr r0, =0xFFFED000
str r1, [r0]

pop {lr}
bx lr
```

The tricky part about configuring the GIC is calculating the appropriate address offset and bit or byte from the IRQ #. Fortunately ARM's documentation provides the `config_interrupt:` subroutine that does that for us; all we need to ensure is that the IRQ # is in `r0` and that the target CPU number is in `r1`. [1]

[1] And, unless you are writing some highly-efficient, multithreaded code, you probably don't care which CPU is used so you can leave the CPU target as `#1`.

The subroutine for configuring a specific IRQ # in the GIC is given below. It can probably be used without modification in any code you write.

```
config_interrupt:
  push {r4 - r5, lr}

  @ calculate register in ICDISERn
  lsr r4, r0, #3
  bic r4, r4, #3
  @ get ICDISERn base address
  ldr r2, =0xFFFED100
  @ get IRQ #-specific ICDISERn address
  add r4, r2, r4
  @ find bit
  and r2, r0, #0x1F
  @ get bit mask to enable
  mov r5, #1
  lsl r2, r5, r2

  @ now register address is in r4
  @ bit mask is in r2
  @ enable specific IRQ # in ICDISERn register
  ldr r3, [r4]
  orr r3, r3, r2
  str r3, [r4]

  @ now calculate register in ICDIPTRn
```

```
bic r4, r0, #3
@ get ICDIPTRn base address
ldr r2, =0xFFFED800
@ find IRQ #-specific ICDIPTRn address
add r4, r2, r4
@ find byte
and r2, r0, #3
@ combine word and byte address
add r4, r2, r4

@ now byte address is in r4
@ set CPU target in byte for specific IRQ #
@ in ICDIPTRn register
strb r1, [r4]

pop {r4 - r5, lr}
bx lr
```

If you cared about setting the interrupt priority level (using *ICDIPRn* or whether the interrupt is edge- or level-triggered (using *ICDICFRn*) you should do so here. This would require extra parameters — probably `r2` should contain the desired priority level, and `r3` contain `#2` if you want the interrupt to be edge-triggered, `#0` for level-triggered — so we in addition to adding the code to do that, we would also need to modify the code to avoid using these registers for general purposes.

As a reference, the structure of the GIC distributor interface is shown in Figure 5. There is one bit, byte, or pair of bits for each IRQ value. This figure was also shown in Lesson 7.

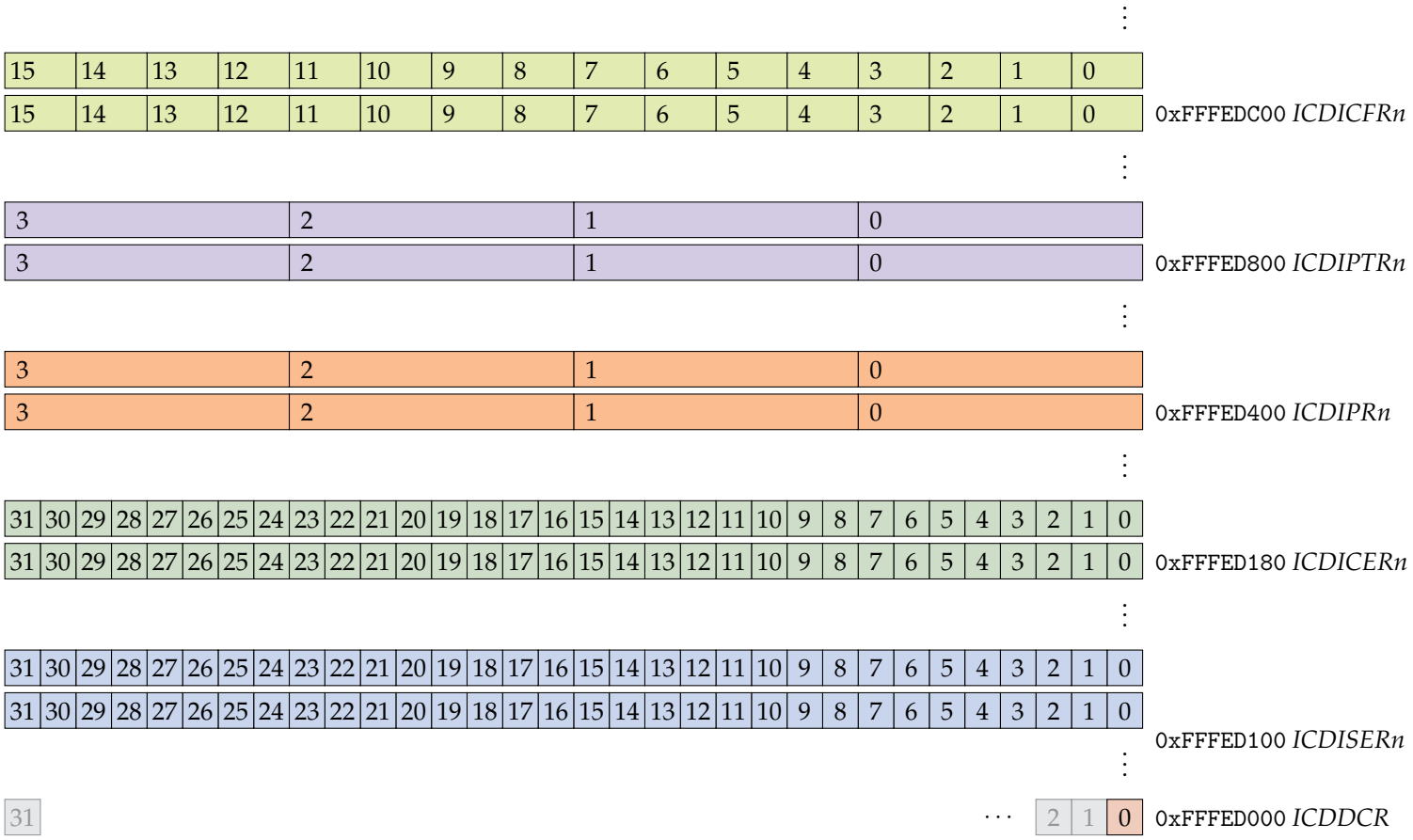

Figure 5: Structure of the GIC distributor registers. Every register with "*n*" at the end of its name is repeated until there is enough space for all IRQ peripherals.

The interrupt enable bit (in *ICDISERn*) and target processor byte (in *ICDIPTRn*) **must** be set to enable a given IRQ #: for IRQ # n, the

nth bit above the base address of *ICDISERn* must be set to 1, and the nth byte above the base address of *ICDIPTRn* must be set to 1 or 2 (to identify core 0 or core 1 as the target processor).

The interrupt priority level can be set in the nth byte above the base address of *ICDIPRn* — although the default value of 0 is the highest priority so it is fine to leave it as-is.

If you want the interrupt to be edge-triggered, set the nth pair of bits above the base of *ICDIFRn* to 2. Leaving it as the default value of 0 sets the interrupt to be level-triggered, which is usually fine.

For further reference, the structure of the GIC CPU interface is shown in Figure 6. The two bottom registers, *ICCICR* and *ICCPMR* must be set when initializing interrupts. Remember that when an interrupt is triggered, *ICCIAR* holds the IRQ #. After an interrupt is processed, write that same IRQ # to *ICCEOIR* to clear the interrupt.

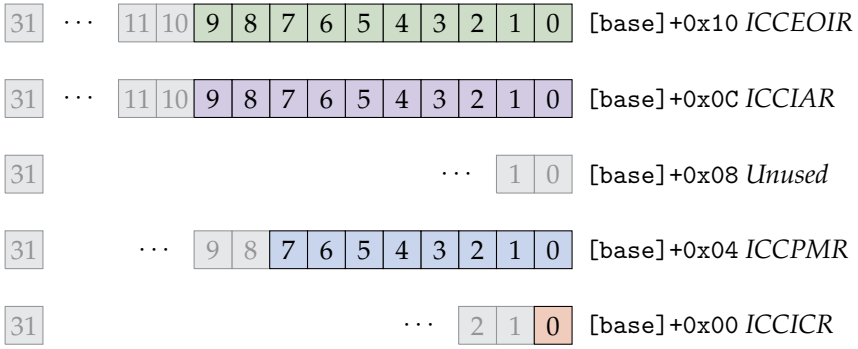

Figure 6: Structure of the GIC CPU interface registers. The base address is `0xFFFEC100`. For some reason there is an unused register at `0xFFFEC108`.

Interrupt Programming on the Nios II

Let's implement the above program using the Nios II microprocessor rather than the ARM Cortex-A9. [2] The simulator also supports the Nios II; in fact the Nios II is actually a microcontroller that can be built on the FGPA in the Cyclone V chipset. We did not study the Nios II in this course, and I **do not** expect you to learn it. The point of this exercise is to demonstrate another way of handling interrupts.

[2] I found the code for this example from an undergraduate lab at UBC from 2011, click here for the link.

- The Nios II allows 32 hardware interrupts, and each of these is assigned a bit in a status register.
- Configuring a Nios II for interrupts is as simple as enabling the appropriate bits in the special-purpose register `ienable`.
- Global interrupts are configured in the `status` register.

The following code is written in Assembly language for the Nios II. It is different from ARMv7 Assembly, but similar enough that you should be able to follow along. There is no need to try and learn every line of code, just understand the basic flow of the program.

The first step is to write a subroutine for handling exceptions. When a peripheral with a given IRQ value n triggers an interrupt, the nth bit in the `ipending` register is set indicating that an interrupt was triggered. In the case of multiple interrupts simultaneously, the Nios II programmer's guide recommends handling each ISR in ascending order; so IRQ 0 always implicitly has the highest priority.

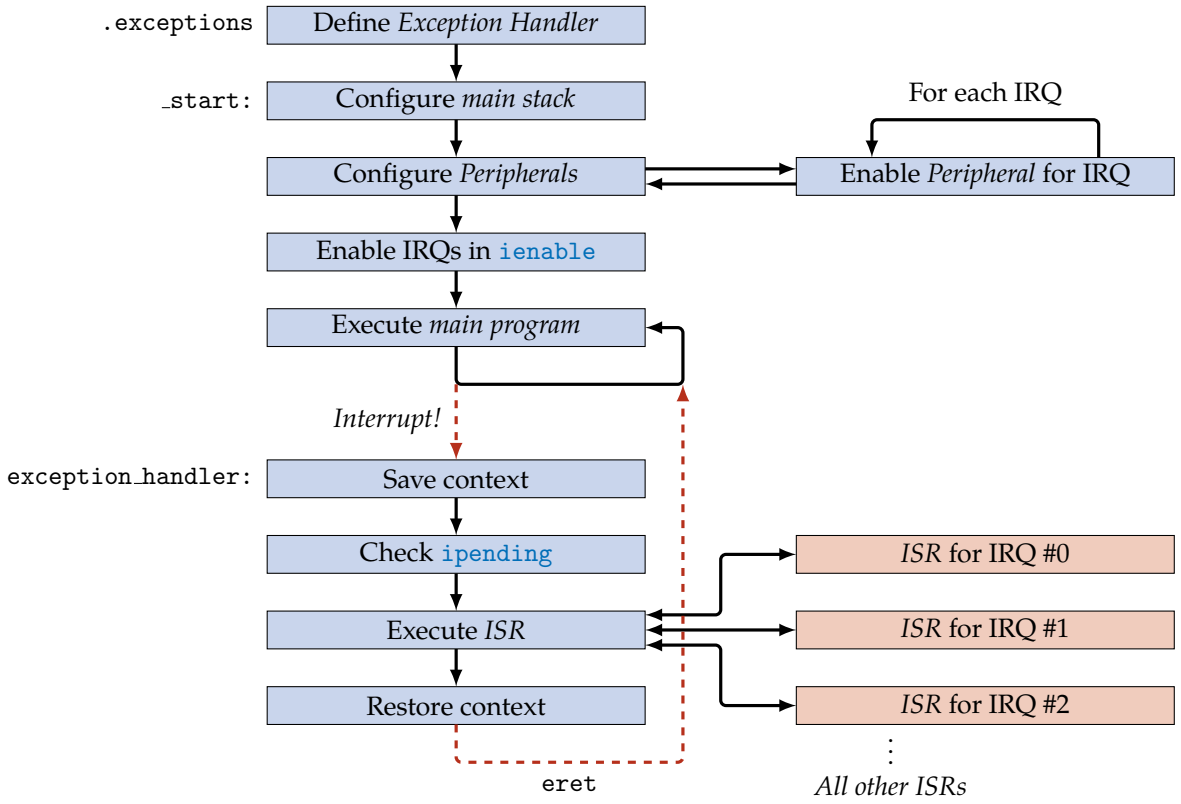

Figure 7: Program flow for enabling interrupts in a Nios II CPU. Everything is done in the same mode. All solid arrows indicate sequential flow or a branch to a normal subroutine. The red dashed arros indicate a hardware-triggered branch from an interrupt.

The code for the exception handler routine is given below. This program only uses interrupts from the timer, which is IRQ 0 in a Nios II. [3] Remember that IRQ 0 corresponds to the 0^{th} bit of a register — unlike an ARMv7 system, the actual IRQ code is never explicitly given.

[3] Even through the Nios II and ARM Cortex-A9 share the same board and peripherals, each has its own way of assigning IRQ numbers to the various devices!

```
exception_handler:
  /* save used regs on stack */
  addi sp, sp, -12
  stw r8, 0(sp)
  stw r9, 4(sp)
  stw ra, 8(sp)

  /* Check if interrupts were enabled by */
  /* examining the EPIE bit. */
  /* EPIE is bit0 of estatus, a copy of */
  /* PIE before the exception */
  rdctl et, estatus
  andi et, et, 1
  beq et, r0, check_software_exceptions
  /* interruptsare enabled */
  /* check if any are pending */
  rdctl et, ipending
  beq et, r0, check_software_exceptions

check_hardware_interrupts:
  /* upon return, execute the interrupted */
  /* instruction */
  subi  ea, ea, 4
  /* should check interrupts one-at-a-time, */
  /* from irq0 to irq31 */
  /* each time the ipending bit is set, we */
  /* should call the proper ISR */
```

```
  /* since we are only expecting irq0, we */
  /* will only check for it */
  andi et, et, 0x1
  beq et, r0, check_next_interrupt
  /* ISR uses r8, r9, and call uses ra */
  call timer_isr

check_next_interrupt:
  /* no more interrupts to check */

check_software_exceptions:
  /* no software exceptions supported */
  /* they should be checked in priority order */
  /* (trap, break, unimplemented) */

done_exceptions:
  /* restore used regs from stack */
  ldw ra, 8(sp)
  ldw r9, 4(sp)
  ldw r8, 0(sp)
  addi sp, sp, 12
  eret
```

This code should be reasonably easy to read. As mentioned above, a Nios II interrupt for IRQ value n is determined by checking the nth bit in `ipending`, in contrast to an ARM system which instead should have the IRQ value n stored in a the memory-mapped *ICCIAR* register.

The timer hardware is configured the same way as on an ARM system — the actual timer is the same hardware, with the same memory-mapped registers. Here the timer can interval can be set and the timer can be configured for countdown-and-repeat with interrupts all at once.

```
/* set up timer to send interrupts */
/* parameter r4 holds the # cycles for */
/* the timer interval */
setup_timer_interrupts:
  /* set the timer period */
  /* extract low halfword */
  andi r2, r4, 0xFFFF
  stwio r2, TIMER_START_LOW(r23)
  /* extract high halfword */
  srli r2, r4, 16
  stwio r2, TIMER_START_HIGH(r23)

  /* start timer (bit2), count continuously */
  /* (bit1), enable irq (bit0) */
  movi r2, 0b0111
  stwio r2, TIMER_CONTROL(r23)

  ret
```

In this code, the memeory-mapped addresses for the timer registers were already defined by the precompiler (like `TIMER_CONTROL`). Furthermore, as noted in the code, this subroutine expects the timer interval to be stored in register `r4`.

Setting up the CPU to receive interrupts is much simpler on a Nios II than on an ARM. All we need to do to enable interrupts from IRQ value n is to set the nth bit in the *interrupt enable register* `ienable` and then enable global interrupts in the `status` register.

```
/* set up CPU to receive interrupts from timer */
setup_cpu_interrupts:
  /* bit0 = irq0 = countdown timer device */
  movi r2, 0x01
  wrctl ienable, r2
  /* bit0 = PIE */
  movi r2, 1
  wrctl status, r2

  ret
```

Finally, the ISR for the timer is written as a normal subroutine.

```
/* every interval, increment interrupt_counts */
/* and display on LED */
timer_isr:
  /* clear source of interrupt */
  stwio r0, TIMER_STATUS(r23)

  /* process the interrupt */
  movia r9, interrupt_counts
  ldw r8, 0(r9)
  addi r8, r8, 1
  stw r8, 0(r9)
  /* show count on LED */
  stwio r8, LED(r23)

  /* return from ISR */
  ret
```

Again, the memory-mapped addresses for the timer and for the LEDs were already defined by the precompiler. The flag `interrupt_counts` is a global variable holding the number of $0.5\,\mathrm{s}$ intervals that have elapsed — remember, an ISR cannot accept any parameters as it can be called at any time without warning. Given the simplicity of this program, and the fact that a Nios II does not have banked registers, it seems possible (to me, anyway, but I'm no expert in the Nios II) that this count could also be stored in a general-purpose register that is only used for that purpose.

The main program is then as follows. Note that this also contains the code to configure the stack.

```
_start:
  movia r23, IOBASE
  /* make sure stack is initialized */
  movia sp, STACK_END
  movia r4, interrupt_counts
  stw r0, 0(r4)

  /* # of timer cycles in 0.5 s */
  movia r4, 50000000
  call setup_timer_interrupts
  call setup_cpu_interrupts

loop:
  br loop
```

Here register `r23` is used to hold the base address of all peripherals. This base address is then offset by the appropriate amount to reach the necessary peripheral — `TIMER_STATUS(r23)` really refers to address `0x2000 + 0xFF200000`, as the base address for peripherals is `0xFF200000`. Note that `r23` is be accessed from the main code and from ISRs without any issue, so quite possibly another register could be used the same way to hold the current number of $0.5\,\mathrm{s}$ intervals elapsed.

I am too lazy to try and learn Nios II Assembly to modify the above code to include the push buttons to clear the count, but you are welcome to try.