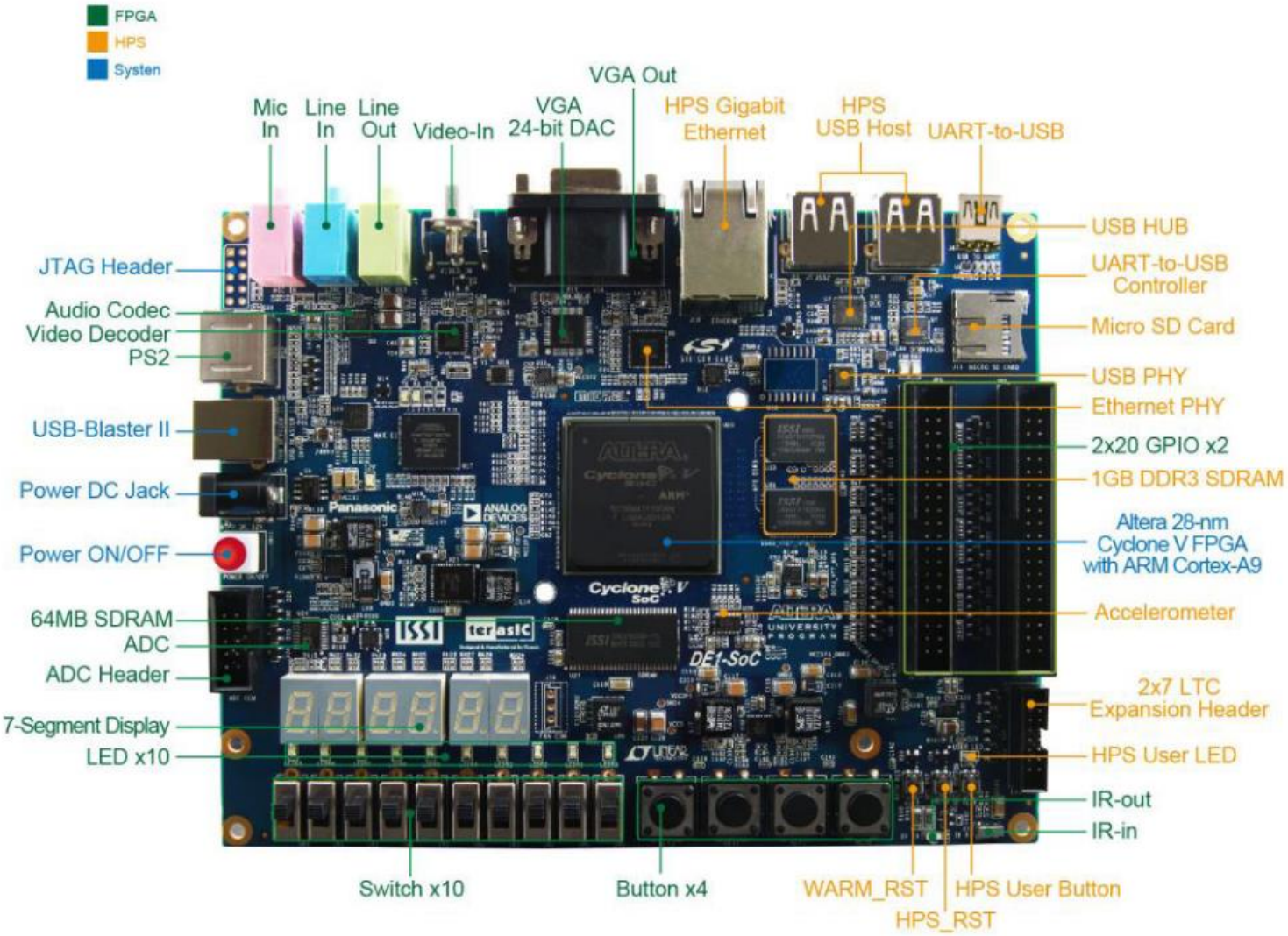
Chapter 4
Input/Output

4.1 Introduction of I/O devices in CPUlator

Objectives

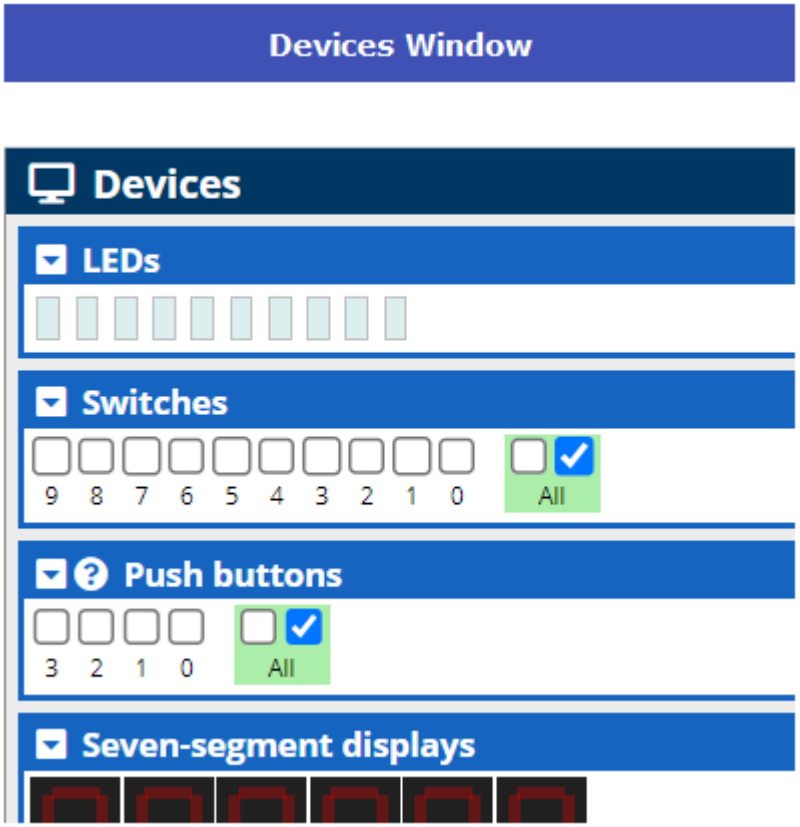
- Learn how to directly control an external device through assembly programs.
- External devices covered include:
  - LED
  - Button input
  - 7-segment LED
  - JTAG UART
  - Timer
- These devices are available in CPUlator as virtual input/output devices in the virtual De1-SoC computer board.

DE1-SoC


FPGA
HPS
Systen
Mic In
Line In
Line Out
Video-In
VGA 24-bit DAC
VGA Out
HPS Gigabit Ethernet
HPS USB Host
UART-to-USB
JTAG Header
Audio Codec
Video Decoder
PS2
USB-Blaster II
Power DC Jack
Power ON/OFF
64MB SDRAM
ADC
ADC Header
7-Segment Display
LED x10
Switch x10
Button x4
WARM_RST
HPS_RST
HPS User Button
IR-out
IR-in
HPS User LED
2x7 LTC Expansion Header
Accelerometer
Altera 28-nm Cyclone V FPGA with ARM Cortex-A9
1GB DDR3 SDRAM
2x20 GPIO x2
Ethernet PHY
USB PHY
Micro SD Card
UART-to-USB Controller
USB HUB
DE1-SoC

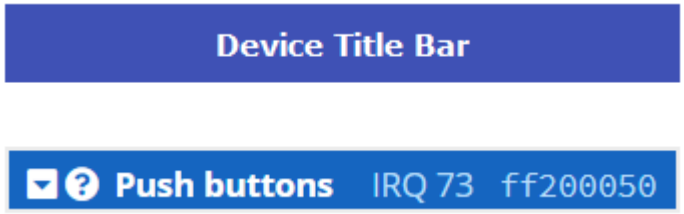
DE1-SoC Devices in CPUlator

- The simulated devices appear in the Devices window



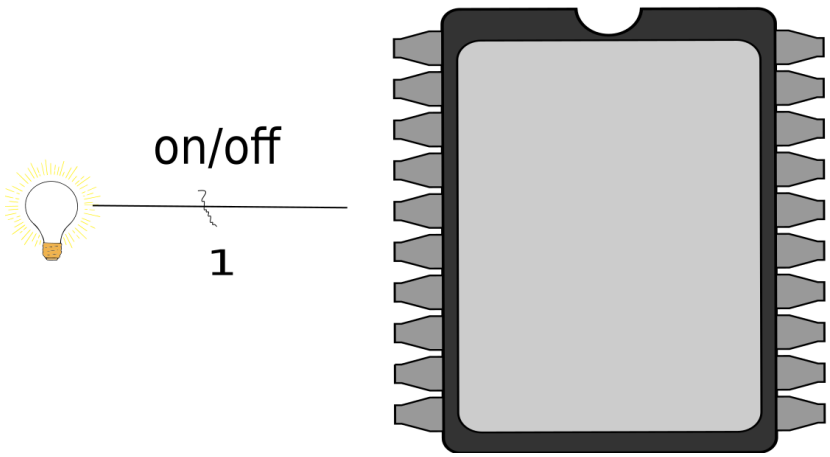
DE1-SoC Devices in CPUlator

- All individual devices panel have a title bar, like:



- All title bars have a device's hexadecimal base address (e.g., FF200050 above) which is used to read from or write to the device memory.
- Some title bars have an IRQ (interrupt request) value (e.g., IRQ 73 above) that is used to interface with that particular device.

The high-level picture



- From the main chip we want to control an (external) device, here an LED.
- We use one of the LEDs connected to the chip on the CPUlator.
- We want to send **1 bit** to this device to turn it **on/off**.

Low-level picture

Programmers need to:

- Identify the peripheral registers, in charge of LEDs, from the Altera De1-SoC manual:

https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/18.1/Computer_Systems/DE1-SoC/DE1-SoC_Computer_ARM.pdf

- Identify the bits in the registers that control a specific LED
- Then, directly access the device via memory read/writes into these registers
- In the program, turning on an LED is typically just one write operation

Finding the registers in the manual

- The relevant information can be found under Section “2. De1-SoC Computer Contents” from the Altera De1-SoC manual
- Here, we need the information on the LEDs, in Section “Red LED Parallel Port”.
- Note that the base address is also available in the title bar in CPUlator:

LED explained

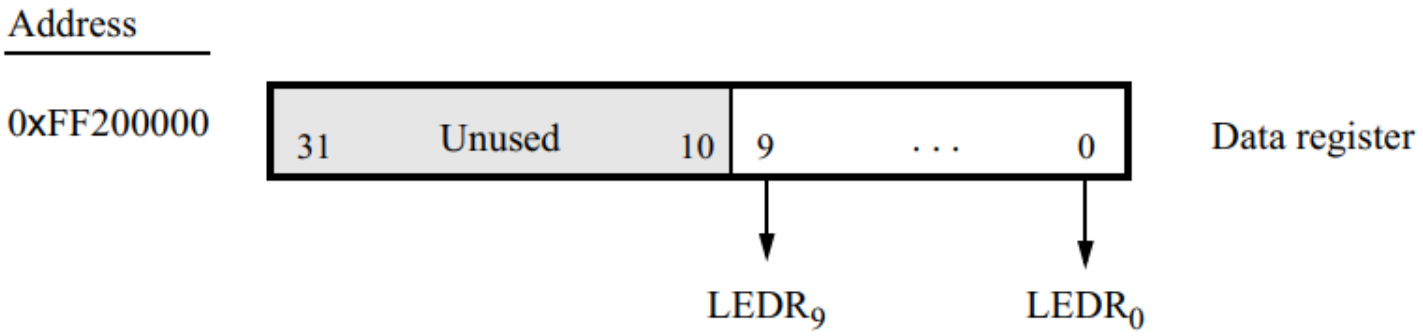


Figure 9. Output parallel port for *LEDR*.

The picture (and discussion in the manual) tells us

- The lowest 10 bits in the register at address 0xFF200020 are in charge of 10 LEDs hard-wired to the chip
- The remaining bits in the register are unused
- Note that the processor has a word-size of 32-bits
- Each of the lower bits controls one LED, and turns it on (bit: 1) or off (bit: 0)

Controlling an LED in ARM Assembler

```
1  // Constants
2  .equ LEDBASE, 0xff200000     // LED base address
3
4  .text  // Code Section
5  .global _start
6  _start:
7
8  ldr r0, =LEDBASE     // physical address of LEDs
9  mov r2, #0b00000001  // bitmask to turn on first LED
10 str r2, [r0]         // turn it on
```

- First, we load the bitmask 0b00000001 (first LED on, all other LEDs off) into register R2.
- Then we write this value into the LED register.
- Result: 

Blinking LED in ARM assembler

```
1  // Constants
2  .equ LEDBASE, 0xff200000     // LED base address
3
4  .text  // Code Section
5  .global _start
6  _start:
7  ldr r2, =LEDBASE @ physical address of the LED
8
9  loop:
10 eor r3, r3, #1 // toggle current value: 1->0, 0->1
11 str R3, [R2] // store value into LED register
12 bl delay // a delay function depending on the blinking frequency
13 b loop // infinite loop
```

Summary

- Controlling a simple external device means logically sending 1 bit of information (on/off)
- Realizing this control means physically writing into special registers which have special meaning
- The information on the special meaning is in the Altera De1-SoC manual
- Once understood, the code for direct device control is fairly short

Programming a Button input device

- The CPUlator simulates 4 push-buttons
- The push-buttons are used as input devices.
- To identify the registers involved, we need to look at “Pushbutton Key Parallel Port” of the De1-SoC Manual.

Push-button details

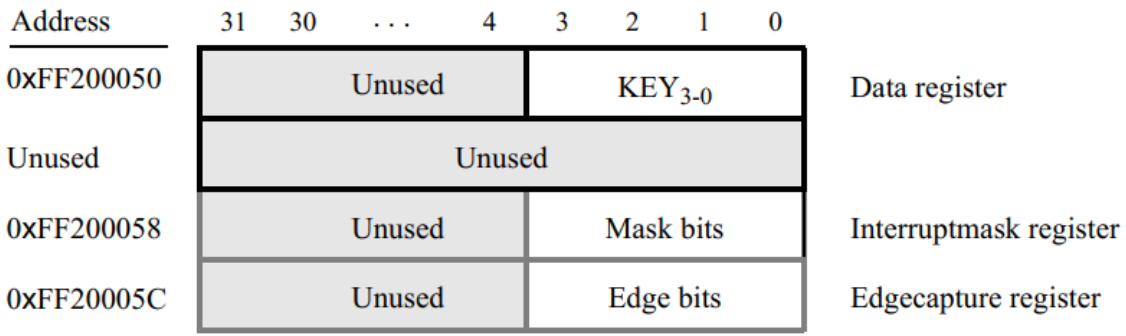


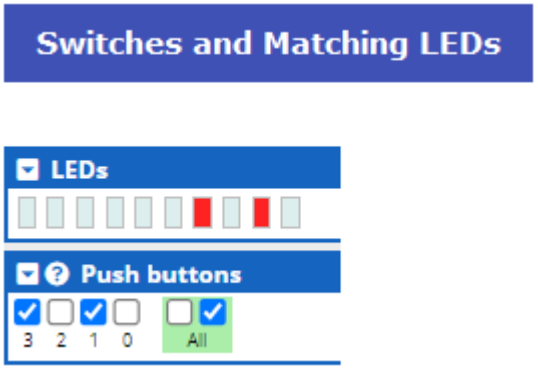
Figure 12. Registers used in the pushbutton parallel port.

- We need to read from address 0xFF200050 to determine the state of the buttons
- There is a one-to-one relationship between the lowest 4 bits in the register and the buttons:
  - if button 0 is pressed, bit 0 will be set to 1
  - if button 1 is pressed, bit 1 will be set to 1
- We will look at the remaining two registers later when we deal with interrupt.

Copying switches to LEDs

```
1  // Constants
2  .equ PUSHBASE,  0xff200050  // Push buttons base address
3  .equ LEDBASE,   0xff200000  // LEDS base address
4
5  .text  // Code Section
6  .global _start
7  _start:
8
9  ldr r0, =PUSHBASE
10 ldr r1, =LEDBASE
11
12 TOP:
13 ldr r2, [r0]    // Read in push buttons - active pressed
14 str r2, [r1]    // Copy to LEDs
15 b TOP
16
17 .end
```

The code above sets LEDs based on the buttons pressed:



7-segment LEDs

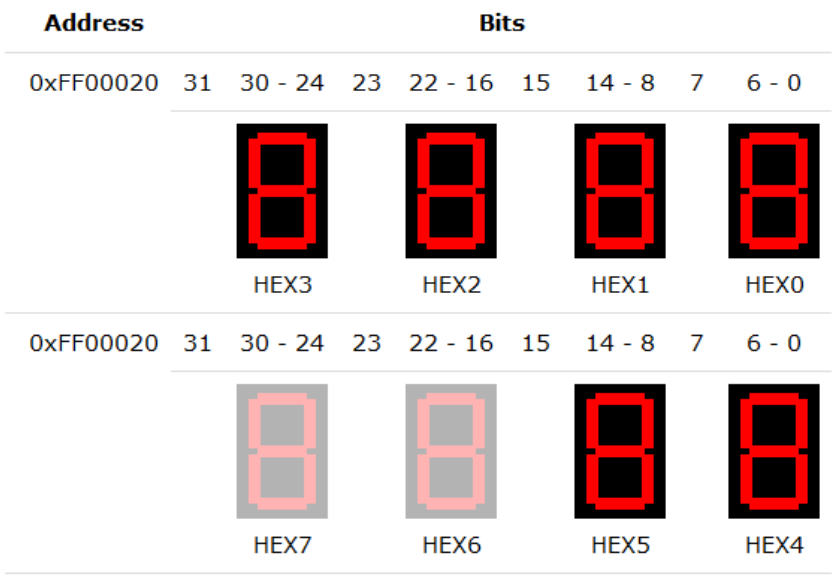
- The CPUlator provides several 7-segment displays.
- The principle to program the display is the same as for the LED
- These displays can be used to display numbers, and some letters
- Therefore, the output is more meaningful than just one LED
- Relevant information is in “7-Segment Displays Parallel Port” of the manual.

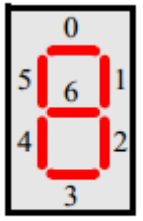
7-segment LEDs

- 6 red 7-segment display

- The individual displays are labelled as HEX0 (rightmost) to HEX5 (leftmost).

7-segment LEDs





Segments

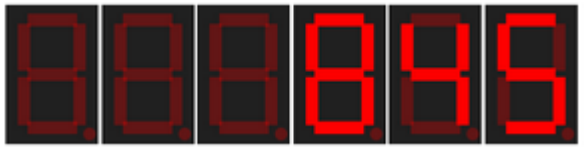
- The 7 segments in the HEX display are each controlled by 1 bit in the HEX data register
- The mapping of the bit positions to the segments is shown in the picture on the right, e.g., bit 0 controls the upper bar in display 1
- We need 7 bits per HEX display in the register (8th bit unused).

Example

Write the digits 845 to the rightmost three 7-segment displays (HEX0 to HEX2):

```
1  // Constants
2  .equ SEVEN_SEGMENT_BASE, 0xff200020 // first four 7-Segment Displays base address
3  // Define digit display bit offsets
4  .equ HEX1,  8
5  .equ HEX2,  16
6  // Define block digit segments
7  .equ FOUR,  0b01100110
8  .equ FIVE,  0b01101101
9  .equ EIGHT, 0b01111111
10
11 .org 0x1000
12 .text
13 .global _start
14 _start:
15
16 ldr r0, =SEVEN_SEGMENT_BASE    // load first seven segment address
17 ldr r1, =FIVE                 // load segments definition of 5
18 ldr r2, =FOUR                 // load segments definition of 4
19 orr r1, r1, r2, lsl #HEX1     // or definitions of 5 and 4 - shift 4 to proper digit
20 ldr r2, =EIGHT                // load segments definition of 8
21 orr r1, r1, r2, lsl #HEX2     // or definitions of 5, 4, and 8 - shift 8 to proper digit
22 // equivalent to:
23 // ldr r1, =#0x007F666D
24 str r1, [r0]                  // write all digits to seven segment display
25
26 _stop:
27 b _stop
28 .end
```

Result:

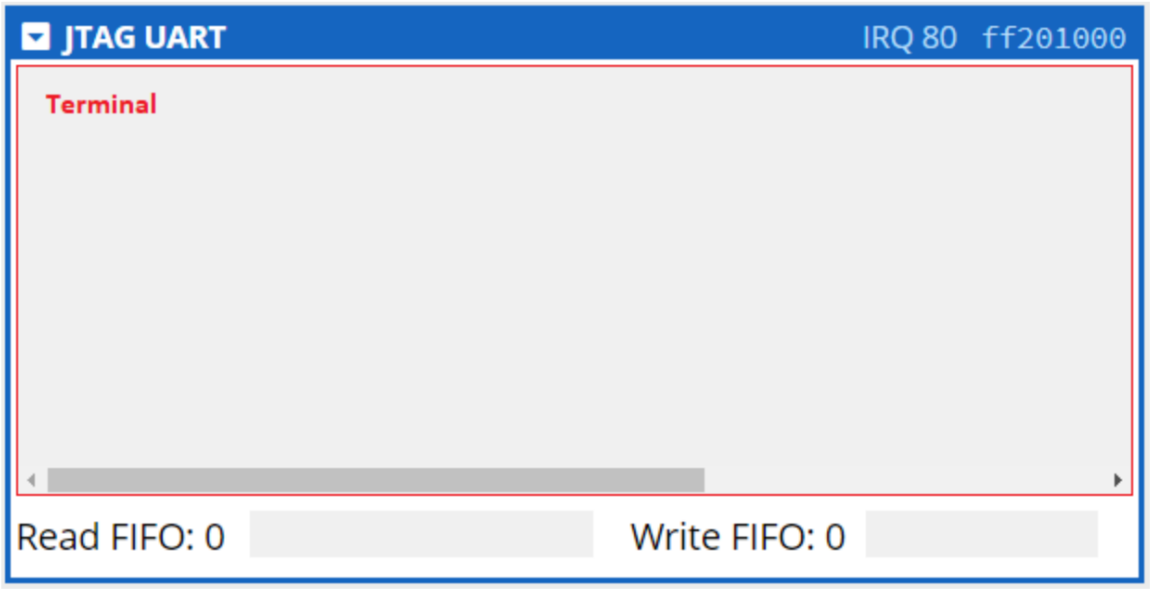


JTAG UART

- A ‘real’ DE-SoC computer can be connected to an external device (e.g., keyboard, monitor) and communicate to it through its JTAG UART.
- CPUlator cannot communicate to an external device, but it can simulate:
  - writing to
  - reading from

  a JTAG UART box.

JTAG UART Console



- Character input is handled by the 64-byte Read FIFO queue
- Character output by the 64-byte Write FIFO queue.

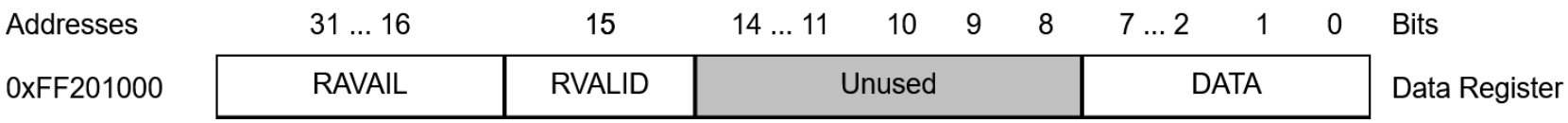
Read FIFO Queue

- Enter characters into the queue by clicking on the terminal area and typing in CPUlator.
- These characters then appear in the Read queue in hexadecimal format.

```
Read FIFO: 10 48 69 20 74 68 65 72 65 2e 0a
```

- Characters entered beyond the 64-byte limit are ignored and no more characters may be entered into the Read FIFO until the queue is *read by the microcontroller*.

UART Data Register



- Data: a character (byte) at the head of the queue.
- RAVAIL: Number of characters in the queue. RAVAIL decrements once after each read.
- RVAILD: 1 if RAVAIL > 0, otherwise 0.

Reading a string (Lab 8)

ReadString subroutine

```
38 ReadString:
39 /*
40 -------------------------------------------------------
41 Reads an ENTER terminated string from the UART.
42 -------------------------------------------------------
43 Parameters:
44   r4 - address of string buffer
45 Uses:
46   r0 - holds character to print
47   r1 - address of UART
48 -------------------------------------------------------
49 */
50
51 stmfd sp!, {r0, r1, r4}  // preserve temporary registers
52 ldr   r1, =UART_BASE     // get address of UART
53
54 rsLOOP:
55 ldr  r0, [r1]       // read the UART data register
56 tst  r0, #VALID     // check if there is new data
57 beq  _ReadString    // if no data, exit subroutine
58 strb r0, [r4]       // store the character in memory
59 add  r4, r4, #1     // move to next byte in storage buffer
60 b    rsLOOP
61 _ReadString:
62 ldmfd sp!, {r0, r1, r4}  // recover temporary registers
63 bx    lr                 // return from subroutine
```

Main Program

```
21 ldr    r4, =First
22 bl    ReadString
```

New String in Memory

Memory (Ctrl-M)

Go to address, label, or register: 1000 Refresh

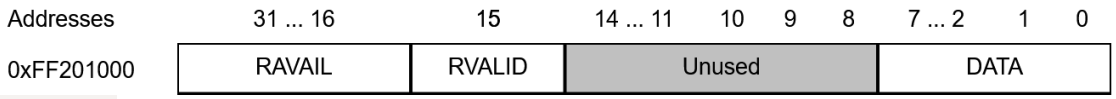
Address	Memory contents and ASCII				
00001000	2c 10 9f e5	2c 40 9f e5	50 50 84 e2	00 00 91 e5	,••• ,@•• PP•• ••••
00001010	02 29 10 e2	05 00 00 0a	ff 00 00 e2	00 00 c4 e5	•)•• •••• •••• ••••
00001020	01 40 84 e2	05 00 54 e1	00 00 00 0a	f6 ff ff ea	•@•• ••T• •••• ••••
00001030	fe ff ff ea	00 10 20 ff	40 10 00 00	00 00 00 00	•••• •• • @••• ••••
00001040	54 68 69 73	20 69 73 20	61 6e 20 69	6e 70 75 74	This is an i nput
00001050	20 73 74 72	69 6e 67 2e	00 00 00 00	00 00 00 00	str ing. •••• ••••
00001060	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	•••• •••• •••• ••••
00001070	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	•••• •••• •••• ••••
00001080	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	•••• •••• •••• ••••
00001090	aa aa aa aa	aa aa aa aa	aa aa aa aa	aa aa aa aa	•••• •••• •••• ••••

String buffer to store values read from UART

```
65 .data
66 .align
67 // The list of strings
68 First:
69 .space  SIZE
```

Reading a string (Lab 8)

```
1  .equ UART_BASE, 0xff201000 // UART base address
2  …
3  .equ VALID, 0x8000  // Valid data in UART mask
4  …
5  ldr  r1, =UART_BASE
6  …
7  ldr  r0, [r1]       // read the UART data register
8  tst  r0, #VALID     // check if there is new data
9  beq  _stop          // if no data, return 0
10 …
```

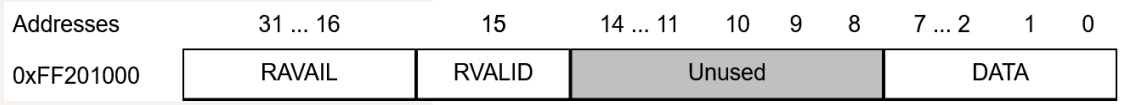


1. Load the contents of the UART data register into r0.
2. Note if there is new data to be read, RVALID (the 15th bit of r0) is set to 1.
   - `tst r0, #VALID` performs the "and" operation between r0 and the mask with only the 15th bit set to 1 (i.e., 0x8000)
   - if RVAILD = 0, the "and" operation result is zero, thereby setting the Z flag to 1. Otherwise Z = 0.
3. Use the `beq` branch instruction to branch to `stop` if RVALID = 0 → no writing to the list is required. Otherwise, the character read will be written to the list.

Reading a string (Lab 8)

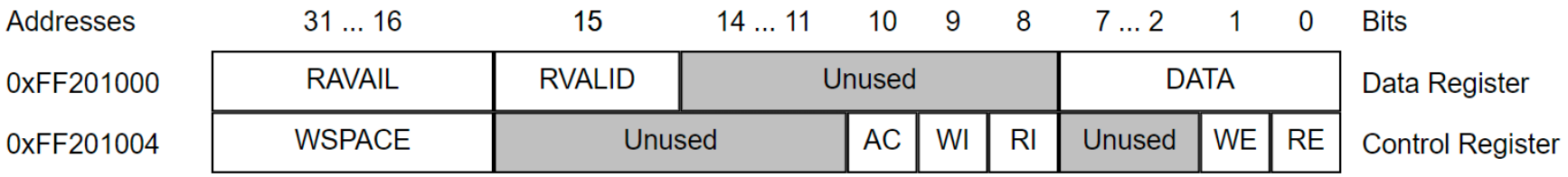
The actual characters in the *read FIFO* are copied into memory by the lines:

```
1  .equ UART_BASE, 0xff201000 // UART base address
2  …
3  ldr  r1, =UART_BASE
4  ldr  r4, =TEXT_strING
5  …
6  strb r0, [r4]      // store the character in memory
7  add  r4, r4, #1    // move to next byte in storage buffer
8  …
```



- r0 contains the whole data register. Only the last byte is data. Thus, `strb` is used to copy the data to the text string in memory.
- move to the next byte location to prepare for writing the next character.

Write FIFO queue (Lab 8)



- To write a character to UART, write to the least significant byte of the Data register.
- The character is then written to the Write FIFO queue.
- Characters in the Write FIFO are automatically removed from the queue and displayed in the Terminal over time.
- The WSPACE (remaining space available in the queue) field in the control register is then decremented.
- Write FIFO queue stores 64 bytes. If the Write FIFO queue is full, the character written to the Data register will be lost.

Writing a string (Lab 8)

WriteString subroutine

```
33 WriteString:
34 /*
35 -----------------------------------------------------
36 Prints a null terminated string to the UART.
37 -----------------------------------------------------
38 Parameters:
39   r4 - address of string to print
40 Uses:
41   r0 - holds character to print
42   r1 - address of UART
43 -----------------------------------------------------
44 */
45 stmfd sp!, {r0, r1, r4}  // preserve temporary registers
46 ldr   r1, =UART_BASE     // get address of UART
47
48 wsLOOP:
49 ldrb  r0, [r4], #1       // load a single byte from the string
50 cmp   r0, #0
51 beq   _WriteString       // stop when the null character is found
52 strb   r0, [r1]           // copy the character to the UART DATA field
53 b     wsLOOP
54 _WriteString:
55 ldmfd sp!, {r0, r1, r4}  // recover temporary registers
56 bx    lr                 // return from subroutine
```

Main program

```
18 ldr r4, =First
19 bl  WriteString
20 ldr r4, =Second
21 bl  WriteString
22 ldr r4, =Third
23 bl  WriteString
24 ldr r4, =Last
25 bl  WriteString
26
27 _stop:
28 b    _stop
```

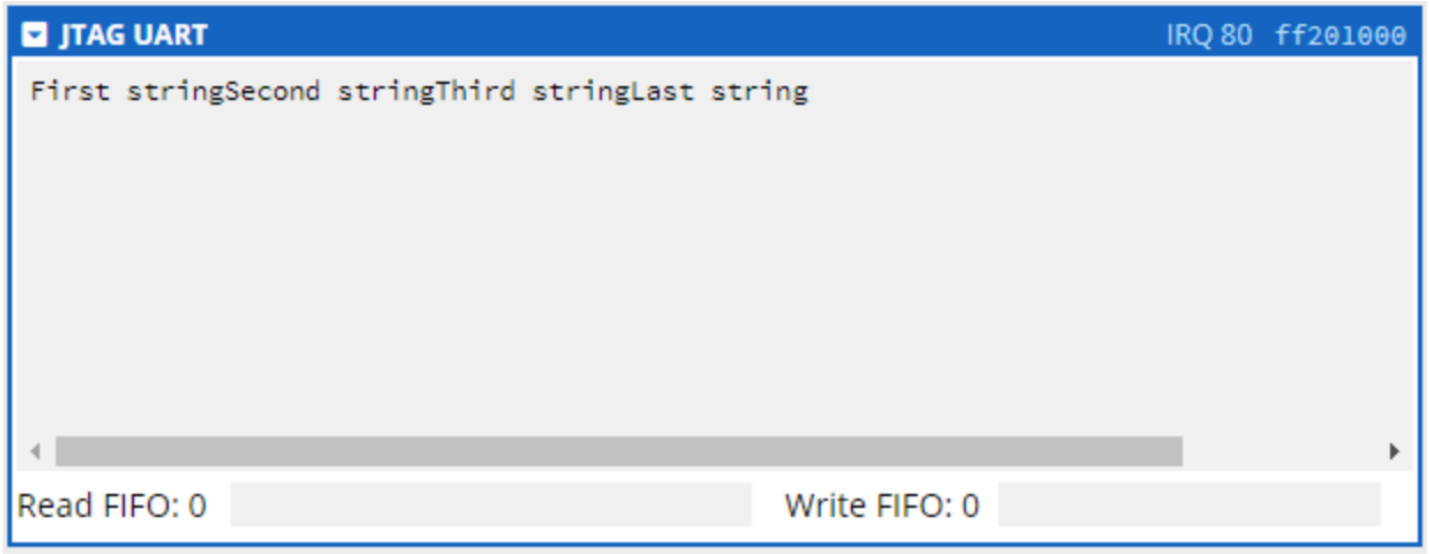
Data strings

```
58 .data
59 .align
60 // The list of strings
61 First:
62 .asciz  "First string"
63 Second:
64 .asciz  "Second string"
65 Third:
66 .asciz  "Third string"
67 Last:
68 .asciz  "Last string"
69 _Last:    // End of list address
```

The `.asciz` directive defines a null-terminated string.

Writing a string (Lab 8)

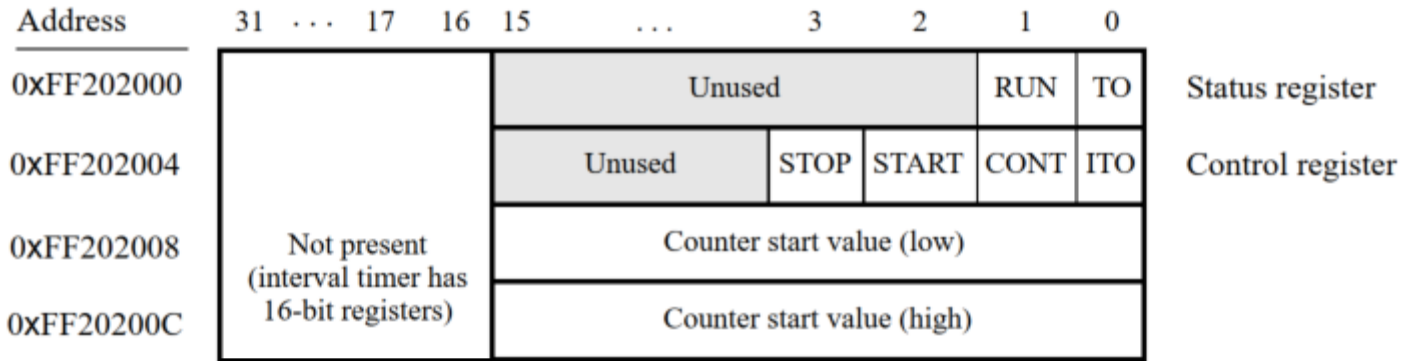
Here is the result of the program:



Timer

- Counter is a register that can be loaded with a binary number (count) which can be decremented per clock cycle.
- We can use this counter to measure time.
- Analogous to setting your iPhone timer to a specific time (e.g., 3 minutes) and an alarm will be sounded after the 3 minutes have elapsed.
- Will introduce the ARM A9 MPCore Private Timer and the Interval Timer here.

Interval Timer



- Users write an initial count value to the Counter start value register. This controls the time delay generated by the timer.
- **TO**: Set to 1 by the timer when it has reached 0. Reset by writing a 0 into it.
- **START/STOP**: commence/suspend the operation of the timer by writing a 1 to the respective bit.
- **ITO**: Enable interrupt
- **CONT**: If CONT = 1, when the timer reaches 0, it automatically reloads the count value and the timer continues counting down. If CONT = 0, the timer stops after it has reached 0.

Interval Timer

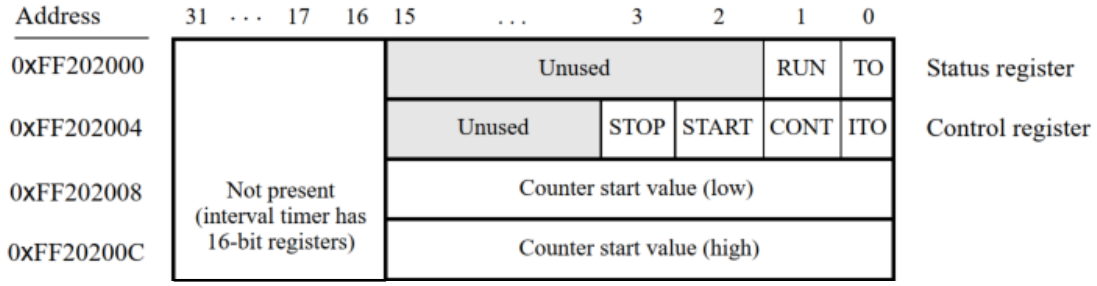
- Clock frequency = 100MHz
- Clock period = 1/100MHz = $\frac{1}{100}\mu s$
- How should we initialize the Count start value register to generate a specific time delay?
- Time delay = Clock period x Count start value
- Count start value = Time delay / clock period
- e.g., Time delay = 1 s

Count start value = $\frac{1}{\frac{1}{100}} \times 10^6 = 100 \times 10^6$

Example

This program toggles the rightmost LED every 1 second:

```
1  .equ TIMER_BASE, 0xff202000  // Interval Timer base addresss
2  .equ LED_BASE, 0xff200000 // LED Base
3
4  .text
5  .global _start
6  _start:
7
8  LDR r0, =TIMER_BASE
9  ldr r3, =LED_BASE
10 mov r4, #0
11 str r4, [r3]
12
13 /* set the interval timer period for scrolling the led displays */
14 ldr r1, =100000000
15 // 1/(100 mhz) x 100x10^6 = 1 sec
16 str r1, [r0, #0x8] // store the low half word of counter
17 // start value
18 lsr r1, r1, #16
19 str r1, [r0, #0xc] // high half word of counter start value
20
21 // start the interval timer, enable its interrupts
22 mov r2, #0 // initialize T0 to be 0
23 str r2, [r0]
24
25 mov r1, #0x7 // START = 1, CONT = 1, ITO = 1
26 str r1, [r0, #0x4]
27
28 loop:
29 wait:
30 ldr r1, [r0] // read T0: T0 would set to 1 if the timer counts to 0
31 tst r1, #0b1
32 beq wait
33 str r2, [r0] // re-intialize T0 to be 0
34 eor r4, r4, #1
35 str r4, [r3]  // toggle LED
36 b loop
37
38 .end
```



ARM A9 Private Timer

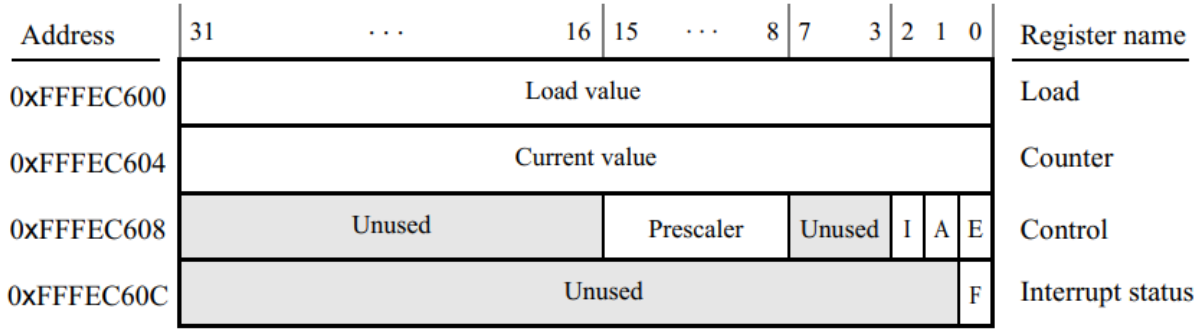


Figure 3. ARM A9 private timer port.

- Users first write an initial count value to the *Load* register. This controls the time delay generated by the timer.
- The timer is started by setting the enable bit E to 1 and it can be stopped by setting E to 0.
- Once enabled, the timer decrements until it reaches 0.
- When it reaches 0:
  - the timer sets F bit in the *Interrupt Status* register, indicating the desired time period has been elapsed. F bit is reset to 0 by writing a 1 to it.
  - the timer will stop if the auto bit A in the *Control* register is set to 0. Otherwise, the timer will automatically reload the value in the Load register and continue decrementing.

ARM A9 Private Timer

- Clock frequency = 200MHz
- Clock period = 1/200MHz = $\frac{1}{200}\mu s$
- How should we initialize the Load register to generate a specific time delay?
- Time delay = Clock period x Load
- Load = Time delay / clock period
- e.g., Time delay = 1 s

$$\text{Load} = \frac{1}{\frac{1}{200}} \times 10^6 = 200 \times 10^6$$

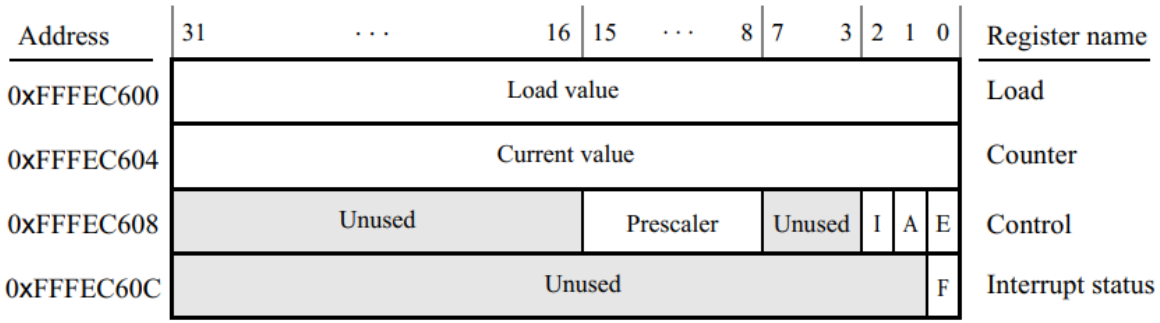
ARM A9 Private Timer

- The *Prescaler* in the *Control* register slows down the counting rate, thereby extending the delay time period.
- The timer decrements every *Prescaler* + 1 clock cycles
  - Prescaler = 0: decrements every clock cycle
  - Prescaler = 1: decrements every two clock cycles.
- Time delay = Clock period x Load x (Prescaler + 1)
- With the same initial Load register value, the time delay is extended by a factor of Prescaler + 1.
- e.g., Load = 200×10^6:
  - Prescaler = 0, time delay = 1s
  - Prescaler = 1, time delay = 2s …..
  - Prescaler = 255, time delay = 256s

Example

This program toggles the rightmost LED every 1 second:

```
1  // Constants
2  .equ PRIVATE_TIMER_BASE, 0xFFFEC600 // Private Timer Base
3  .equ LED_BASE, 0xFF200000 // LED Base
4
5  .org 0x1000
6  .text
7  .global _start
8  _start:
9
10 ldr r0, =LED_BASE
11 mov r2, #0
12 str r2, [r0]
13
14 ldr r1, =PRIVATE_TIMER_BASE // MPCore private timer base address
15 mov r4, #1
16 str r4, [r1, #0xC] //Writing 1 to the interrupt status bit resets it
17
18 ldr r3, =200000000 // timeout = 1/(200 MHz) x 200 x 10^6 = 1 sec
19 str r3, [r1] // write to timer load register
20 mov r3, #0b011 // set bits: mode = 1 (auto), enable = 1
21 str r3, [r1, #0x8] // write to timer control register
22 LOOP:
23 WAIT: ldr r3, [r1, #0xC] // read timer status
24 cmp r3, #0
25 beq WAIT // wait for timer to expire
26 str r4, [r1, #0xC] // reset timer flag bit
27 eor r2, r2, #1 // toggle LEDG value
28 str r2, [r0]
29 b LOOP
30 .end          // write all digits to seven segment display
```



4.2 Interrupt Programming

Why Interrupt?

- A computer is much more than the CPU
  - Keyboard, mouse, screen, disk drives, scanner, printer, sound card, camera, etc.
- These devices occasionally need CPU service
  - But we can’t predict when
- Solution 1: CPU periodically checks each device to see if it needs service – Polling

Disadvantage of Polling

- “Polling is like picking up your phone every few seconds to see if you have a call.”
- You may get a few phone calls a day → 99% of your effort will be wasted checking the phone.
- Takes CPU time even when no request pending

Example of Polling in Timer

This program toggles the rightmost LED every 1 second:

```
1  // Constants
2  .equ PRIVATE_TIMER_BASE, 0xFFFEC600 // Private Timer Base
3  .equ LED_BASE, 0xFF200000 // LED Base
4
5  .org 0x1000
6  .text
7  .global _start
8  _start:
9
10 ldr r0, =LED_BASE
11 mov r2, #0
12 str r2, [r0]
13
14 ldr r1, =PRIVATE_TIMER_BASE // MPCore private timer base address
15 mov r4, #1
16 str r4, [r1, #0xC] //Writing 1 to the interrupt status bit resets it
17
18 ldr r3, =200000000 // timeout = 1/(200 MHz) x 200 x 10^6 = 1 sec
19 str r3, [r1] // write to timer load register
20 mov r3, #0b011 // set bits: mode = 1 (auto), enable = 1
21 str r3, [r1, #0x8] // write to timer control register
22 LOOP:
23 WAIT: ldr r3, [r1, #0xC] // read timer status
24 cmp r3, #0
25 beq WAIT // wait for timer to expire
26 str r4, [r1, #0xC] // reset timer flag bit
27 eor r2, r2, #1 // toggle LEDG value
28 str r2, [r0]
29 b LOOP
30 .end                  // write all digits to seven segment display
```

Address	31 ··· 16	15 ··· 8	7 3	2	1	0	Register name
0xFFFEC600	Load value						Load
0xFFFEC604	Current value						Counter
0xFFFEC608	Unused	Prescaler	Unused	I	A	E	Control
0xFFFEC60C	Unused					F	Interrupt status

Interrupt

- Whenever a device needs the CPU's service, the device notifies it by sending an interrupt signal.
- CPU then stops and serves the device.
- "Polling is like picking up your phone every few seconds to see if you have a call. Interrupts are like waiting for the phone to ring."

ARM Operating Modes

- ARM has seven basic operating modes. The following modes are related to interrupts:
  - Supervisor (SVC): entered when the processor is first powered on
  - IRQ: entered when a normal interrupt is raised
  - FIQ: entered when a high-priority interrupt is raised.
- We just consider IRQ interrupt in this course.

CPSR

- The operating mode is indicated in the CPSR:

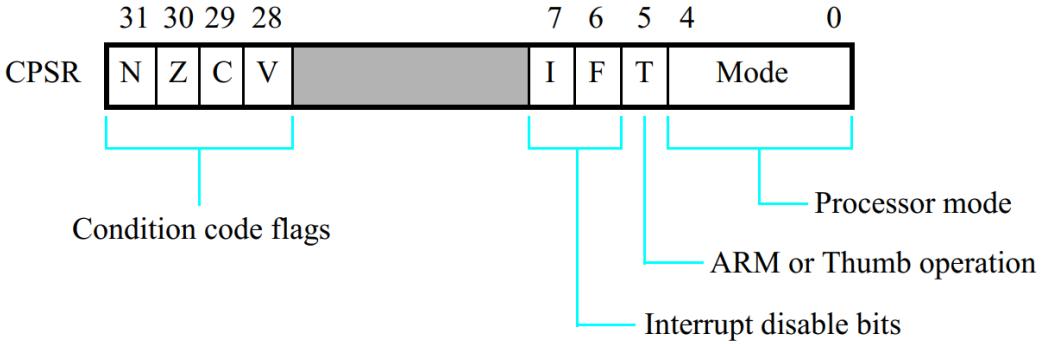
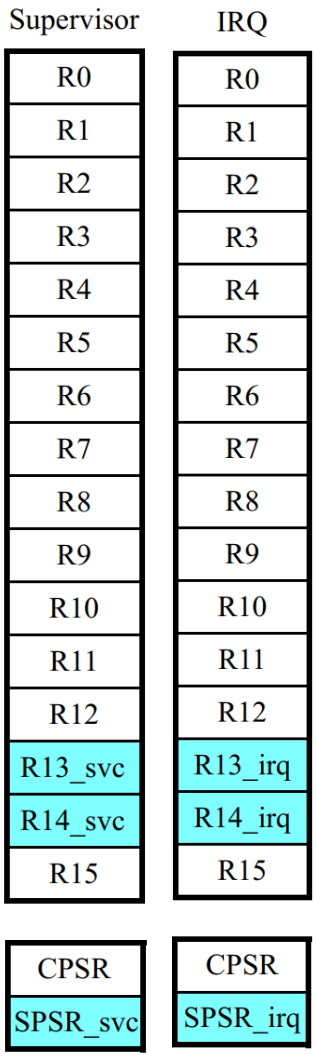


TABLE 1. Mode Bits

$CPSR_{4-0}$	Operating Mode
10000	User
10001	FIQ
10010	IRQ
10011	Supervisor
10111	Abort
11011	Undefined
11111	System

ARM Register Set



- r0-r12, r15 (pc) are common to Supervisor and IRQ modes.
- r13 (sp) and r14 (lr) are unique in each mode.
- CPSR is common to all modes
- SPSR is for preservation and restoration of CPSR, as discussed next.

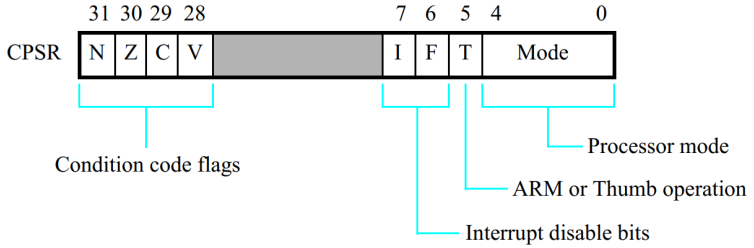
Response to Interrupts

- If the interrupt request is present, the processor:
  - Completes the execution of the current instruction
  - Saves the address of the program counter on the link register so that the processor knows where to return to after handling the interrupt.
  - is redirected to the memory location where the interrupt request can be met. The location is listed in a *vector table*.
- The set of instructions written to meet the request is called an interrupt service routine (ISR). *At the end of the ISR, there must be an indication that the interrupt has been served (so that it would not be served repeatedly).*
- Once the request is accomplished, the processor should find its way back to the next instruction, where it was interrupted.

Entering the interrupt handling workflow

If the interrupt request is present, the processor:

- Completes the execution of the current instruction
- Switches from SVC to IRQ mode.
  - Saves the current pc to the link register of the IRQ mode.
  - Copies CPSR to the SPSR in the IRQ mode.
  - IRQ disable bits set to 1: When handling an interrupt, another interrupt would not be responded to.



- is redirected to the memory location where the interrupt request can be met. The location is listed in a *vector table:*
  - IRQ – 0x00000018

Leaving the interrupt handling workflow

After the interrupt is handled, the processor:

- Copies SPSR back to CPSR, which will:
  - automatically switch back from the IRQ to the SVC mode.
  - Clear the IRQ disable bit, so that the processor can handle other interrupts.
- Move the link register back (minus an offset, will be discussed) to pc to return to where interrupt occurred.

Link register offset

- Recall that due to pipelining, when an instruction executes, the current pc points to 8 bytes in advance of the instruction.
- This is stored in the lr of the IRQ mode
- The current instruction will complete execution before entering the interrupt handling workflow.
- The next instruction that the processor should return to have an address of lr-4.

Link register offset

- Therefore, the ISR is returned to the main function by using the instruction:

```
subs pc, lr, #4
```

- This instruction:
  - Move the pc back to where the interrupt occurred.
  - Copies SPSR back to CPSR, which will:
    - automatically switch back from the IRQ to the SVC mode.
    - Clear the IRQ disable bit, so that the processor can handle other interrupts.

Generic Interrupt Controller (GIC)

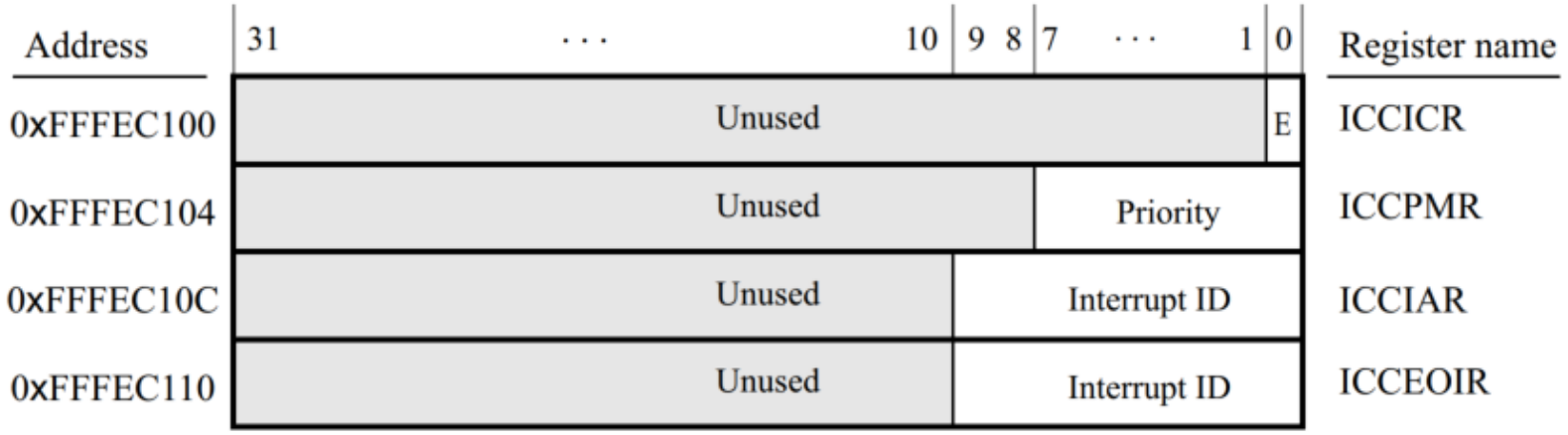
- Interrupts associated with I/O peripherals are handled by GIC.
- The GIC activates IRQ interrupt in the processor.
- Each I/O peripheral is identified in the GIC by an interrupt ID number. The interrupt IDs of some devices in CPUlator are listed below:

I/O Peripheral	Interrupt ID
Cortex-A9 Private timer	29
Cortex-A9 Watchdog timer	30
Interval timer 1	72
Push buttons	73
Interval timer 2	74
JTAG UART	80
HPS watchdog timer 1	203
HPS watchdog timer 2	204

Generic Interrupt Controller (GIC)

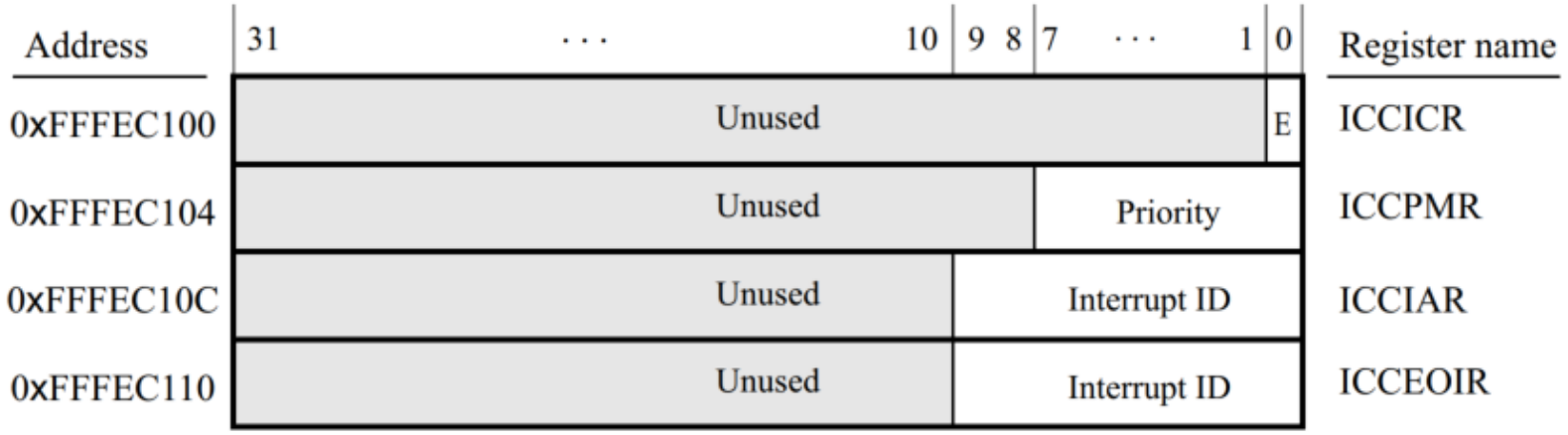
- The GIC consists of the distributor and the CPU Interface
- The distributor receives IRQ interrupt signals from I/O peripherals.
- The CPU interface is responsible for sending IRQ requests to the processor.
- When programming interrupt, we need to ensure both are enabled.
- Key registers needed to be set are listed below.

CPU interface registers



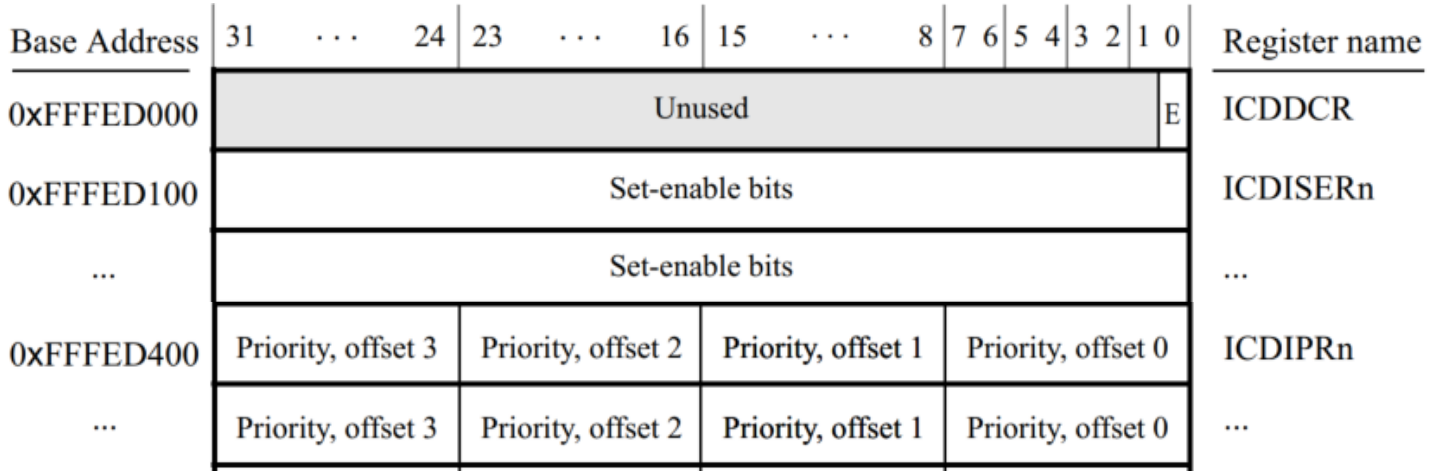
- CPU Interface Control Register (ICCICR): The E bit enables sending of the interrupt
- Interrupt Priority Mask Register (ICCPMR): set a threshold for the priority-level of the interrupts that will be served.
  - 0 is the highest priority interrupt
  - 255 is the lowest priority interrupt
  - ICCPMR = 0: No interrupt will be served
  - ICCPMR = 255: Interrupts with priority higher than 255 will be served

CPU interface registers



- Interrupt Acknowledge Register (ICCIAR)
  - contains the Interrupt ID of the I/O peripheral that has caused an interrupt.
  - ISR must read ICCIAR to determine which I/O peripheral has caused the interrupt.
- End of Interrupt Register (ICCEOIR)
  - Writing the interrupt ID to ICCEOIR indicates that the interrupt has been served.

GIC distributor registers



Base Address	31 ··· 24	23 ··· 16	15 ··· 8	7 6	5 4	3 2	1 0	Register name
0xFFFED000	Unused						E	ICDDCR
0xFFFED100	Set-enable bits							ICDISERn
...	Set-enable bits							...
0xFFFED400	Priority, offset 3	Priority, offset 2	Priority, offset 1		Priority, offset 0			ICDIPRn
...	Priority, offset 3	Priority, offset 2	Priority, offset 1		Priority, offset 0			...

- Distributor control register (ICDDCR)
  - Setting E = 1 enables the distributor
- Interrupt Set Enable Registers (ICDISERn)
  - Enable/disable I/O peripherals with Interrupt IDs
  - e.g., 0xFFFED100 enables/disables whether an interrupt is triggered for each of the I/O peripherals with interrupt IDs from 0 to 31.
- Interrupt Priority Registers (ICDIPRn)
  - Sets the priority level of each interrupt
  - e.g., 0xFFFED400 sets the priority level of each of the I/O peripherals with interrupt IDs from 0 to 3.

Putting it together …..

- The following provides an example program that displays:
  - 0 in the 7-segment LED when the first button is pushed.
  - 1 when the second button is pushed.
  - 2 when the third button is pushed
  - 3 when the fourth button is pushed
- The program is provided as `PushbuttonInterrupt.s`

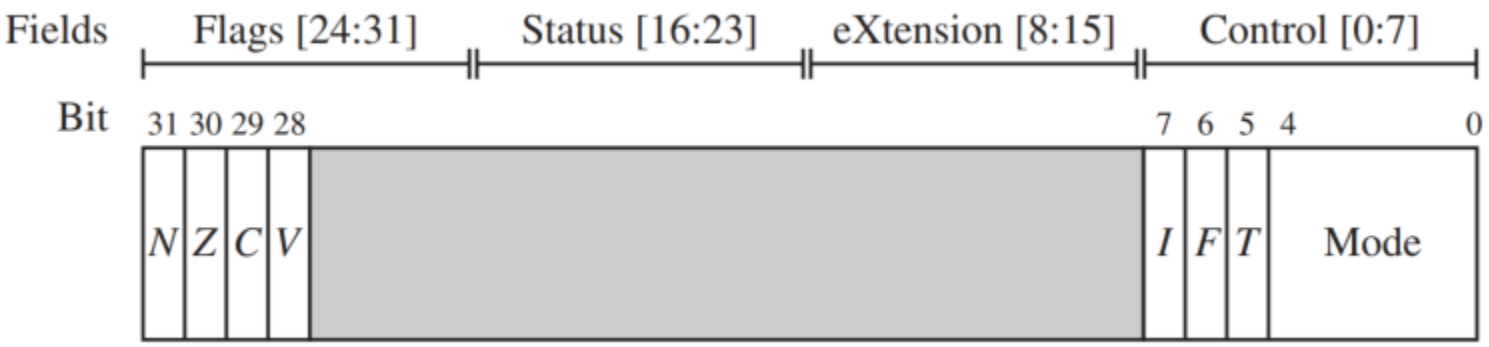
Initialization (1)

```
11 .text
12 .global _start
13 _start:
14     /* Set up stack pointers for IRQ and SVC processor modes */
15     MOV R1, #0b11010010 // interrupts masked, MODE = IRQ
16     MSR CPSR_c, R1 // change to IRQ mode
17     LDR SP, =0xFFFFFFFF - 3
18     /* Change to SVC (supervisor) mode with interrupts disabled */
19     MOV R1, #0b11010011 // interrupts masked, MODE = SVC
20     MSR CPSR_c, R1 // change to supervisor mode
21     LDR SP, =0x3FFFFFFF - 3
```

- The purpose of this code is to set up different stack pointers in the IRQ and SVC modes.
- The stack in IRQ and SVC modes are independent and should not interfere with each other.

Initialization (1)

- The msr instruction transfers the contents of a register into CPSR
- Note that the parameter of the msr instruction is CPSR_c.
- c is the Control field that refers to the least significant byte of CPSR, which controls the interrupt masks and the processor mode.



Initialization (2)

```
23      BL CONFIG_GIC // configure the ARM GIC
```

- The CONFIG_GIC subroutine:
  - enables the distributors and CPU interface
  - Set the priority thresholds (ICCPMR) to be 255 so that all interrupts are enabled.
  - Enable the Push button interrupt.

Initialization (3)

```
25    // write to the pushbutton KEY interrupt mask register
26    LDR R0, =0xFF200050 // pushbutton KEY base address
27    MOV R1, #0xF // set interrupt mask bits
28    STR R1, [R0, #0x8] // interrupt mask register (base + 8)
29
30    LDR R2, =#0xFFFFFFFF
31    STR R2, [R0, #0xC] // interrupt mask register (base + 12).
32    //Reset all the interrupt flag.
```

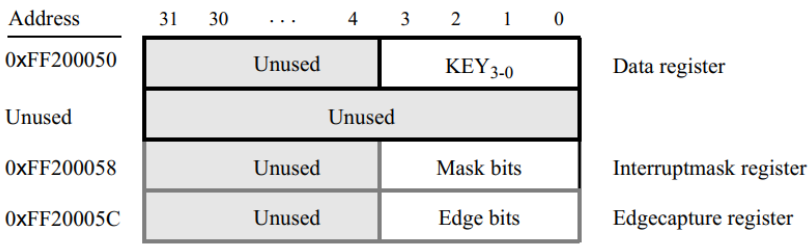


Figure 12. Registers used in the pushbutton parallel port.

- Enable interrupt for all four buttons by enabling all bits in the interrupt mask register.
- Each bit in the edge capture register has a value of 1 if the corresponding bit location has changed from 1 to 0.
- Writing to this register clears all values in this register.

Initialization (4)

```
34      // enable IRQ interrupts in the processor
35      MOV R0, #0b01010011 // IRQ unmasked, MODE = SVC
36      MSR CPSR_c, R0
37  IDLE:
38      B IDLE
```

- Clear the IRQ disable bit in CPSR (i.e., enable IRQ)
- Enter an infinite loop and wait for the push button interrupt to be triggered.

Exception vector table

```
1 .section .vectors, "ax"
2 B _start // reset vector
3 B SERVICE_UND // undefined instruction vector
4 B SERVICE_SVC // software interrrupt vector
5 B SERVICE_ABT_INST // aborted prefetch vector
6 B SERVICE_ABT_DATA // aborted data vector
7 .word 0 // unused vector
8 B SERVICE_IRQ // IRQ interrupt vector
9 B SERVICE_FIQ // FIQ interrupt vector
```

- Different exceptions are directed to different addresses. The entries in the vector table above provide branches to various exception service routines.
  - e.g., when an IRQ is triggered, the pc will be redirected to 0x00000018 at which the instruction `b service_irq` will branch to the ISR handling the IRQ interrupt.

ISR

```
53 SERVICE_IRQ:
54     PUSH {R0-R7, LR}
55 /* Read the ICCIAR from the CPU Interface */
56     LDR R4, =0xFFFEC100
57     LDR R5, [R4, #0x0C] // read from ICCIAR
58 FPGA_IRQ1_HANDLER:
59     CMP R5, #73
60 UNEXPECTED:
61     BNE EXIT2 // if not recognized, stop here
62     BL KEY_ISR
63 EXIT_IRQ:
64 /* Write to the End of Interrupt Register (ICCEOIR) */
65     STR R5, [R4, #0x10] // write to ICCEOIR
66 EXIT2:
67     POP {R0-R7, LR}
68     SUBS PC, LR, #4 // S flag set, and the PC as destination register:
69     //Updates the PC and copies the SPSR into the CPSR
```

This comparison checked whether it is the push buttons triggering an interrupt.

This ensures that GIC knows that the interrupt has been served already, so that it will not be repeatedly served.

Restore CPSR:

- automatically switch back from the IRQ to the SVC mode.
- Clear the IRQ disable bit, so that the processor can handle other interrupts.

ISR

```
KEY_ISR:
    stmfd sp!,{lr}
    LDR R0, =0xFF200050 // base address of pushbutton KEY port
    LDR R1, [R0, #0xC] // read edge capture register
    MOV R2, #0xF
    STR R2, [R0, #0xC] // clear the interrupt
    LDR R0, =0xFF200020 // based address of HEX display
CHECK_KEY0:
    MOV R3, #0x1
    ANDS R3, R3, R1 // check for KEY0
    BEQ CHECK_KEY1
    MOV R2, #0b00111111
    STR R2, [R0] // display "0"
    B END_KEY_ISR
CHECK_KEY1:
    MOV R3, #0x2
    ANDS R3, R3, R1 // check for KEY1
    BEQ CHECK_KEY2
    MOV R2, #0b00000110
    STR R2, [R0] // display "1"
    B END_KEY_ISR
CHECK_KEY2:
    MOV R3, #0x4
    ANDS R3, R3, R1 // check for KEY2
    BEQ IS_KEY3
    MOV R2, #0b01011011
    STR R2, [R0] // display "2"
    B END_KEY_ISR
IS_KEY3:
    MOV R2, #0b01001111
    STR R2, [R0] // display "3"
END_KEY_ISR:
    ldmfd sp!,{pc}
    //BX LR
.end
```

If zero, then button 0 is not pressed.

If zero, then button 1 is not pressed.

If zero, then button 2 is not pressed.

If not button 0, 1 or 2, then it must be button 3 that is pressed.

- Identify which button has been pressed by checking the edge capture register
- After reading, clear the register, to allow the process to start afresh for the next interrupt.

Address	31 30 · · · 4	3 2 1 0	
0xFF200050	Unused	KEY_{3-0}	Data register
Unused	Unused		
0xFF200058	Unused	Mask bits	Interruptmask register
0xFF20005C	Unused	Edge bits	Edgecapture register

Figure 12. Registers used in the pushbutton parallel port.