Part1. UART

The uart is basically a sub-system used for the transfer of data to and from the microprocessor. We will be looking at the simplified version called the UARTlite It is composed of 4 main pieces or "channels" that can be used for operations which are summarized in the picture below(from datasheet). To think of a conceptual example for the uartlite, let's imagine it is a factory.

Table 4: **XPS UART Lite Registers**

Base Address + Offset (hex)	Register Name	Access Type	Default Value (hex)	Description
C_BASEADDR + 0x0	Rx FIFO	Read	0x0	Receive Data FIFO
C_BASEADDR + 0x4	Tx FIFO	Write	0x0	Transmit Data FIFO
C_BASEADDR + 0x8	STAT_REG	Read	0x4	UART Lite Status Register
C_BASEADDR + 0xC	CTRL_REG	Write	0x0	UART Lite Control Register

The Rxfifo is just the received channel. You can think of this as a data and the data (some ASCII value character) as a box coming along a conveyer belt. Each box is a character and the channel is one character long. Any time a character is typed on a dummy terminal, the character is "placed" in the Rxfifo channel. Reading this character is as easy as loading the value stored at the uartlite base_address.

The Txfifo is the transmission buffer for sending characters out. The maximum character "boxes" that can be held in the channel at any given time is 16. Think of things this way, at one end an operator places boxes of characters (1 per box) in the channel and at the other end, a worker places the boxes safely in storage. Let's pretend that the chars will be lost forever, aka broken like a priceless vase if they fall of the Txfifo conveyer belt. Because of the time it takes for the worker to store the box away, if too many boxes are placed in the channel at any given time, we will lose some of the characters. Placing a character (or some number for example) is as easy as storing a value to a 0x4 offset of the uartlite base_address.

The status reg is the display monitor that shows what is in each channel. You can read the information hear to decide what your program should do accordingly. The bits of the status reg can be seen below and the table summarizes the relevant bits you need to work with.

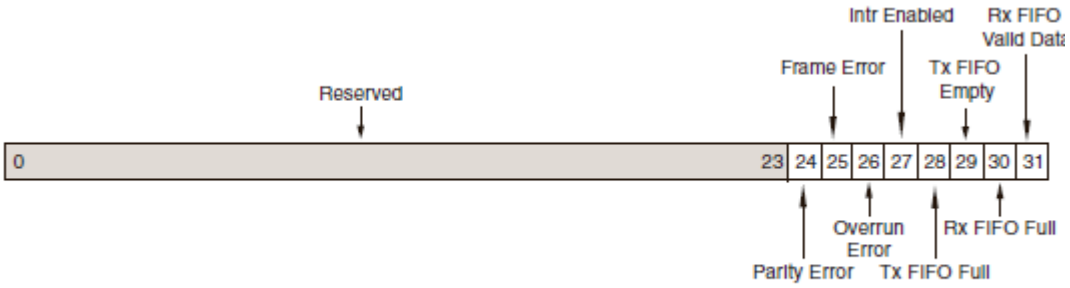


Figure 5: **UART Lite Status Register**

26	Overrun Error	Read	'0'	Indicates that a overrun error has occurred since the last time the status register was read. Overrun is when a new character has been received but the Receive FIFO is full. The received character will be ignored and not written into the Receive FIFO. This bit will be cleared when the status register is read '0' = No interrupt has occurred '1' = Interrupt has occurred
27	Intr Enabled	Read	'0'	Indicates that interrupts is enabled '0' = Interrupt is disabled '1' = Interrupt is enabled
28	Tx FIFO Full	Read	'0'	Indicates if the Transmit FIFO is full '0' = Transmit FIFO is not full '1' = Transmit FIFO is full
29	Tx FIFO Empty	Read	'1'	Indicates if the Transmit FIFO is empty '0' = Transmit FIFO is not empty '1' = Transmit FIFO is empty
30	Rx FIFO Full	Read	'0'	Indicates if the Receive FIFO is full '0' = Receive FIFO is not full '1' = Receive FIFO is full
31	Rx FIFO Valid Data	Read	'0'	Indicates if the receive FIFO has valid data '0' = Receive FIFO is empty '1' = Receive FIFO has valid data

The most important bits will be 28-31. Bit 26 will rarely be seen and you shouldn't see it if you manage the Txfifo decently. Bit 27 while important for checking to make sure you have uart interrupts enabled (if you are using them) it doesn't do anything else. Bit 28 is quite useful. If we want to keep from getting too many characters in the Txfifo, before you place a value in the Txfifo, you can andi. the stat reg value with 8 (1000 binary) to check if it is full. Conversely, repeating this process with a 4 (bit 29) to check if the Txfifo is empty also works well. Bit 30 shouldn't ever really be a concern as you can't humanly type fast enough to fill the Rxfifo (and then my box factory illustration would

fail ☹….). Bit 31 is another key bit as you can do an andi. with 1 to see if there is an ascii character of some time waiting to be read. Reading the stat reg is as easy as loading the value at an offset of 0x8 from the uartlite base_address.

The cntrl reg is the main control hub of the factory. The bits you can control and the summary of them are shown in the picture below.

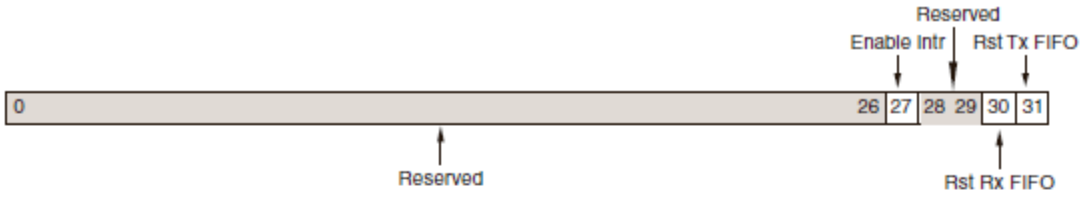


Figure 4: **UART Lite Control Register**

Table 7: **UART Lite Control Register Bit Definitions**

Bit(s)	Name	Core Access	Reset Value	Description
0 - 26	Reserved	N/A	0	Reserved
27	Enable Intr	Write	'0'	Enable Interrupt for the UART Lite '0' = Disable interrupt signal '1' = Enable interrupt signal
28 - 29	Reserved	N/A	0	Reserved
30	Rst Rx FIFO	Write	'0'	Reset/Clear the Receive FIFO Writing a '1' to this bit position clears the Receive FIFO '0' = Do nothing '1' = Clear the Receive FIFO
31	Rst Tx FIFO	Write	'0'	Reset/Clear the Transmit FIFO Writing a '1' to this bit position clears the Transmit FIFO '0' = Do nothing '1' = Clear the Transmit FIFO

Not really used for much other than Masking/unmasking (aka turning the interrupt off during a section of the program), bit 27 can be mainly ignored for most of the programs made in lab. You shouldn't have to worry about this too much, as most of the processes are "automated".

Part2. Interrupts

Interrupts function as an important type of program control. Think of this as an upgraded version of bl (branch and link). If you are not sure about this topic, take a gander at the appendix section of this text. All interrupts require some sort of Boolean condition to be true. Let's summarize this in c/assembly pseudo code. (think of a function call in C)

```
If (condition == true)
   {
        Bl to code_another_place; // go do whatever is at this spot
    Blr; // go back to what I was doing
   }
Else
    Return 0; //do nothing
```

What this "condition" is and when it is true all depends on what interrupt you are used and what settings you have chosen. The main thing to remember is that while these take time, they can make you life easier. With that said, there are two types of interrupts, critical and non-critical. The difference is solely whether an interrupt can be interrupted (critical) or not (non-critical).

In order for the interrupts to know "where to work" in the "factory", you must first tell the ppc where EVPR (Exception-Vector Prefix register) lives. EVPR can live at any multiple value of 64k, ie., 0, lis 0x0001, lis 0x0002 and so on. I recommend using 0 for coding ease. The easiest way to do this is:

```
.set EVPR, 0x3D6
li rA, 0
mtspr EVPR, rA
```

Two key pieces of interrupt enabling are writing a 3 to the MER of the intc and after all initialization is complete, writing a 1 to the EE in the machine status register (MSR) using wrteei 1.

Part3. Intc (interrupt controller)

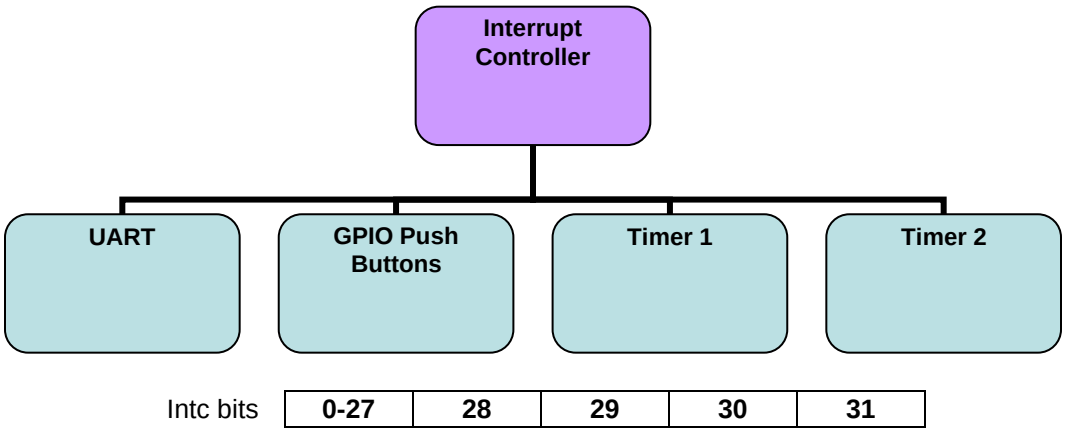
Intc base: lis 0x81800000

The interrupt controller is the main switch board for controlling the interrupts and "works" at an offset of EVPR + 0x500. Composed of 8 different sections (seen below), the intc can seem like a monster but let's see if we can't pull out our shrink ray.

Table 4: **XPS INTC Registers and Base Address Offsets**

Register Name	Base Address + Offset (Hex)	Access Type	Abbreviation	Reset Value
Interrupt Status Register	C_BASEADDR + 0x0	Read / Write	ISR	All Zeros
Interrupt Pending Register	C_BASEADDR + 0x4	Read only	IPR	All Zeros
Interrupt Enable Register	C_BASEADDR + 0x8	Read / Write	IER	All Zeros
Interrupt Acknowledge Register	C_BASEADDR + 0xC	Write only	IAR	All Zeros
Set Interrupt Enable Bits	C_BASEADDR + 0x10	Write only	SIE	All Zeros
Clear Interrupt Enable Bits	C_BASEADDR + 0x14	Write only	CIE	All Zeros
Interrupt Vector Register	C_BASEADDR + 0x18	Read only	IVR	All Ones
Master Enable Register	C_BASEADDR + 0x1C	Read / Write	MER	All Zeros

Let's look at a block diagram of what the intc controls.



Bit 28: corresponds to the uart interrupt
Bit 29: corresponds to the push button interrupt
Bit 30: corresponds to the timer 1 interrupt
Bit 31: corresponds to the timer 2 interrupt

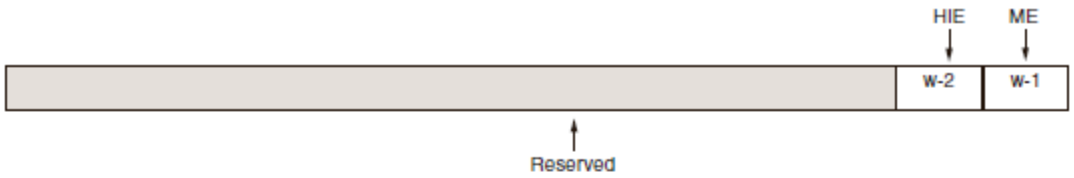
The ISR (interrupt status register) has the duo functionality of being both a read and a write register. Don't worry, as we only will be using the read functionality. When read, the above bits correspond to a pending interrupt. For example, if the value loaded is 0xD, then in binary we have 1101 which means the uart, the push buttons, and timer2 each have an interrupt pending. The ISR is at the base of the intc.

The IER (interrupt enable register) is the section of the intc that makes the magic happen. By writing values to the IER, the corresponding bit lines will be enabled. So if I write 6 to the IER, interrupts from the push buttons and timer 1 will be accepted. The IER lives at an offset of 0x8 of the intc base_address.

The (yes, I'm starting with the again) IAR (interrupt acknowledge register) deals with clearing the flags. What is a flag? It's a 1 bit in the ISR stating the X bit line has an interrupt, hence, if the uart interrupted, with need to write 1000 => 0x8 to the IAR to tell

the system that the interrupt has been taken care of. Note that writing zeros to any bit location here does nothing. The IAR lives at an offset of 0xC from the intc base_address.

The MER (master enable register) is crucial in setting up the interrupts as this is the "master switch" for the intc interrupts. Summarized in the picture are the bits. If this doesn't make sense to you, don't worry, just write a 3 to the MER. The MER lives at an offset of 0x1C from the base of the intc base_address.



Note: w - width of Data Bus

Figure 10: **Master Enable Register (MER)**

Table 12: **Master Enable Register**

Bits	Name	Description	Reset Value
0 : (w(1) – 3)	Unused	**Not used**	All Zeros
(w(1) – 2)	HIE	Hardware Interrupt Enable '0' = Read – SW interrupts enabled Write – No effect '1' = Read – HW interrupts enabled Write – Enable HW interrupts	'0'
(w(1) – 1)	ME	Master IRQ Enable '0' = IRQ disabled – All interrupts disabled '1' = IRQ enabled – All interrupts enabled	'0'

Notes:
1. w - Width of Data Bus

All other sections of the intc should be material covered in higher courses and hence I'm not covering them.

Part4. UART interrupts

Not too much different than the regular uart interacts except some extra set up and flag clearing. The UART interrupt works at evpr + 0x500. Here's some code of the minimum.

```
.org 0x500 # where the uart interupt section lives

# what you want the uart to do
# when an interupt happends
# lives here

li r8, 0x8        # load value for clearing the "flag"
stw r8,IAR(r2) # interupt has been completed
rfi                 # go back to what you were doing


.org 0x2000 # pick 2000 for main or above—where diff interrupts live

#point at UART lite/RxFIFO
lis r22, 0x84000000

#point at base of interupt controller
lis r2,0x81800000

.set IER, 0x8      # distance from base of intc
.set MER,0x1c   # distance from base of intc
.set IAR, 0x0C   # distance from base of intc
.set EVPR,982  # reg value of evpr,  0x3D6 = 982

start: #enable interupts

li r4,3        # remember looks at it in binary
stw r4,MER(r2) # "master interupts"

li r30, 0
mtspr EVPR, r30 # set evpr to 0x00000000
li r30, 0x0010      # 10000 in binary to tell
stw r30,0xc(r22)  # uart it's "working" with interrupts

li r5, 8       # for uart lite interupts
stw r5,IER(r2)

wrteei 1       #enables EE bit to be set
loop: b loop   # sit here and be bored
```

If you notice in the 0x500 section, I have comments saying to tell the uart what to do during an interrupt. This could be a variety of things, but most likely it would be something like loading the value from the Rxfifo and placing it in the Txfifo to echo characters sent from a dummy terminal.

Interrupts for the UART can be triggered in two ways: 1, a character is placed in the Rxfifo, or 2, on the first clock rising edge when the Txfifo is empty.

Part5. Button interrupts

Button base: lis 0x81420000, Button LED base: lis 0x8141

Push buttons belong to the intc interrupt family, so it also lives at evpr + 0x500 (ill deal with multiple intc interrupts later). These are a bit more tricky, but still manageable. First look at the information in the picture below.

Table 8: **XPS GPIO Interrupt Registers**

Register Name	Description	PLB Address	Access
GIE	Global Interrupt Enable Register	C_BASEADDR + 0x11C	Read/Write
IP IER	IP Interrupt Enable Register	C_BASEADDR + 0x128	Read/Write
IP ISR	IP Interrupt Status Register	C_BASEADDR + 0x120	Read/TOW[1]

Notes:
1. Toggle-On-Write (TOW) access toggles the status of the bit when a value of "1" is written to the corresponding bit

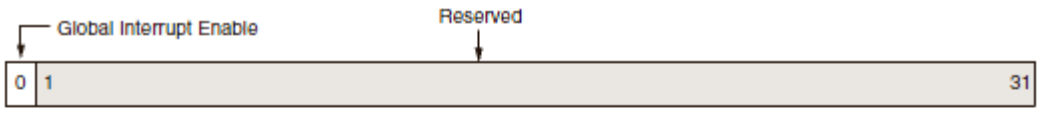


Figure 5: **Global Interrupt Enable Register**

Table 9: **Global Interrupt Enable Register Description**

Bit(s)	Name	Core Access	Reset Value	Description
0	Global Interrupt Enable	Read/Write	'0'	Master enable for the device interrupt output to the system interrupt controller '1' = Enabled '0' = Disabled
1 - 31	Reserved	N/A	'0'	Reserved. Set to zeros on a read

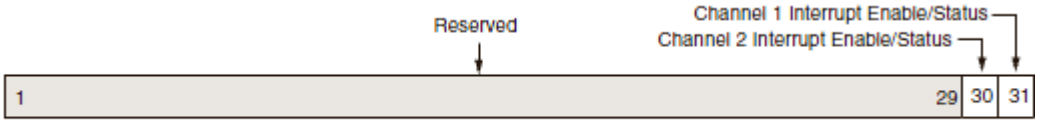


Figure 6: **IP Interrupt Enable and IP Interrupt Status Register**

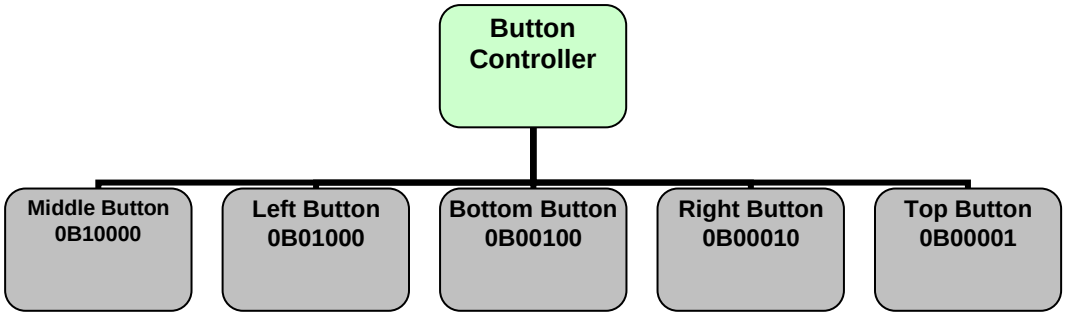
Table 10: **IP Interrupt Enable Register Description**

Bit(s)	Name	Core Access	Reset Value	Description
0 - 29	Reserved	N/A	'0'	Reserved. Set to zeros on a read
30	Channel 2 Interrupt Enable	Read/Write	'0'	Enable Channel 2 Interrupt '1' = Enabled '0' = Disabled (masked)
31	Channel 1 Interrupt Enable	Read/Write	'0'	**Enable Channel 1 Interrupt** '1' = Enabled '0' = Disabled (masked)

Table 11: **IP Interrupt Status Register Description**

Bit(s)	Name	Core Access	Reset Value	Description
0 - 29	Reserved	N/A	'0'	Reserved. Set to zeros on a read
30	Channel 2 Interrupt Status	Read/TOW [1]	'0'	Channel 2 Interrupt Status '1' = Channel 2 input interrupt '0' = No Channel 2 input interrupt
31	Channel 1 Interrupt Status	Read/TOW [1]	'0'	Channel 1 Interrupt Status '1' = Channel 1 input interrupt '0' = No Channel 1 input interrupt

While this all might look scary, it's not. The GIE basically just turns the buttons on as interrupts. Think of things this way, just because the intc power to the button block in the factory is turned on, it doesn't mean that the machines are turned on and running. The IPIER and IPISR work very similar to their uartlite counterparts. In the flow chart I've illustrated which bits correspond to which button/led. Note that since there is 5 buttons—5 bits, the value to turn all button leds on is 0x1F. Also, the TOW for the IPISR means it works also as an IAR when writing, so you can think of it as the "IPIAR" for clearing button flags.



I've tried to explain this the easiest way possible by writing the minimal code for the button interrupts (with the corresponding LEDs).

```
.org 0x500 # buttons live here

# place code for what you want to happen
# the the button interupt is triggered

li r16, 0x01
stw r16,IPISR(r10) # clear flags in button ISR
li r16,4
stw r16,IAR(r2) # clear flags in intc

rfi     # go back previous code when interupted


############################
.org 0x3000    # main program lives here, 0x3000 doesn't matter

.set IER, 0x8    # distance from base of intc
```

```
.set MER,0x1C # distance from base of intc
.set IAR, 0x0C  # distance from base of intc
.set IPIER, 0x128 # distance from base of button
.set IPISR ,0x120 # distance from base of button
.set GIE,0x11c    # distance from base of button
.set EVPR,982

#point at UART lite/RxFIFO
lis r23, 0x84000000


#point at base of interupt controller
lis r2,0x81800000


#point at the buttons
lis r10, 0x8142

# point at the coressponding button LEDs
lis r11,0x8141

li r12,0            #  load zero
stw r12,4(r11)  #  set LEDs to be “outputs”

li r13,0x1F     # 00011111 to r13
stw r13,4(r10)  # make button lines input

#enable buttons/leds
lis r22, 0x8000
stw r22, GIE(r10) # set MSB to 1
li r18, 0x01
stw r18,IPIER(r10) # set LSB to 1

#enable interupts
li r20, 0
mtspr EVPR, r20 # set evpr to 0

li r26,3        # remember looks at it in binary
stw r26,MER(r2) # "master interupts"

li r27, 0x4           # for button interupts
stw r27,IER(r2)   # place 100 in intc
wrteei 1              #enables EE bit to be set
# done enabling

loopdloop: b loopdloop  # infinite loop
```

Part 6. Timer interupts

Timer 1 and timer 2 are part of the intc interupts so setting the coresponding bits in the IER. The confusing part comes from whether or not PWM (pulse width modulation is used) and dealing with all the bit values from configuration. Timer 1 (tscr0) and Timer 2 (tcsr1) are identical, so we will look at the datasheet info we need for timer 1.

Table 4: **XPS Timer/Counter Register Address Map**

Register	Address (Hex)	Size	Type	Description
TCSR0	C_BASEADDR + 0x00	Word	Read/Write	Control/Status Register 0
TLR0	C_BASEADDR + 0x04	Word	Read/Write	Load Register 0
TCR0	C_BASEADDR + 0x08	Word	Read	Timer/Counter Register 0
TCSR1	C_BASEADDR + 0x10	Word	Read/Write	Control/Status Register 1
TLR1	C_BASEADDR + 0x14	Word	Read/Write	Load Register 1
TCR1	C_BASEADDR + 0x18	Word	Read	Timer/Counter Register 1

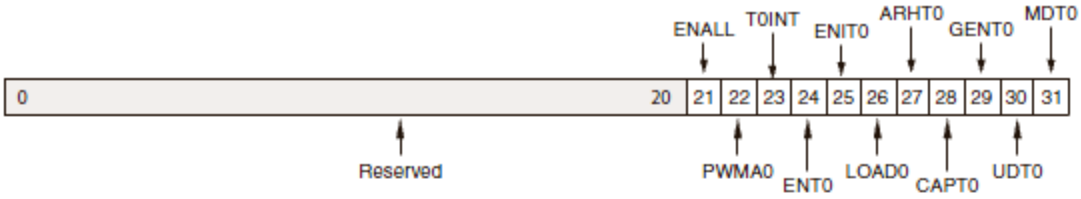


Figure 6: **Timer Control/Status Register 0 (TCSR0)**

Table 7: **Control/Status Register 0 (TCSR0)**

Bits	Name	Description	Reset Value
0 - 20	Reserved	Reserved	-
21	ENALL	**Enable All Timers** 0 = No effect on timers 1 = Enable all timers (counters run) This bit is mirrored in all control/status registers and is used to enable all counters simultaneously. Writing a '1' to this bit sets ENALL, ENT0, and ENT1. Writing a '0' to this register clears ENALL but has no effect on ENT0 and ENT1.	0
22	PWMA0	**Enable Pulse Width Modulation for Timer0** 0 = Disable pulse width modulation 1 = Enable pulse width modulation PWM requires using Timer0 and Timer1 together as a pair. Timer0 sets the period of the PWM output, and Timer1 sets the high time for the PWM output. For PWM Mode, MDT0 and MDT1 must be '0' and C_GEN0_ASSERT and C_GEN1_ASSERT must be '1'.	0
23	T0INT	**Timer0 Interrupt** Indicates that the condition for an interrupt on this timer has occurred. If the timer mode is capture and the timer is enabled, this bit indicates a capture has occurred. If the mode is generate, this bit indicates the counter has rolled over. Must be cleared by writing a '1'. *Read:* 0 = No interrupt has occurred 1 = Interrupt has occurred *Write:* 0 = No change in state of T0INT 1 = Clear T0INT (clear to '0')	0
24	ENT0	**Enable Timer0** 0 = Disable timer (counter halts) 1 = Enable timer (counter runs)	0

Table 7: **Control/Status Register 0 (TCSR0) *(Contd)***

Bits	Name	Description	Reset Value
25	ENIT0	**Enable Interrupt for Timer0** Enables the assertion of the interrupt signal for this timer. Has no effect on the interrupt flag in TCSR0. 0 = Disable interrupt signal 1 = Enable interrupt signal	0
26	LOAD0	**Load Timer0** 0 = No load 1 = Loads timer with value in TLR0	0
27	ARHT0	**Auto Reload/Hold Timer0** When the timer is in Generate Mode, this bit determines whether the counter reloads the generate value and continues running or holds at the termination value. In Capture Mode, this bit determines whether a new capture trigger overwrites the previous captured value or if the previous value is held. 0 = Hold counter or capture value 1 = Reload generate value or overwrite capture value	0
28	CAPT0	Enable External Capture Trigger Timer0 0 = Disables external capture trigger 1 = Enables external capture trigger	0
29	GENT0	Enable External Generate Signal Timer0 0 = Disables external generate signal 1 = Enables external generate signal	0
30	UDT0	Up/Down Count Timer0 0 = Timer functions as up counter 1 = Timer functions as down counter	0
31	MDT0	Timer0 Mode See the Timer Modes section. 0 = Timer mode is generate 1 = Timer mode is capture	0

Wow, yes that sure is a lot of bits to set. Timer 1 doesn't have the option of PWM, so that makes it things easier. I'm going to show initialization code for an example without PWM and then one with PWM.

```
.set somevalufortimer, 0xValue # the timer relationship for 100MHz 1000=10us
                               # Value= 100E6 *20E-6
#point at timer 2
lis r10,0x83c20000

li r11, somevaluefortimer  # load time value
stw r11, TLR0(r10)         # place in TLR1
li r11, 0xF6               # 11110110  --26th bit set to load value from TLR1
stw r11, TCSR0(r10)
li r11, 0xD6               # 11010110   --26th bit disabled
stw r11,TCSR0(r10)
```

TCR0 and TCR1 are basically where the counting values are stored. TLR0 and TLR1 are where the values for counting are placed.

Example with PWM:

```
.set TCR, 0x3DA
.set TSR,0x3D
.set TLR0, 0x4
.set TLR1,0x14

#point at timer 1
lis r10,0x83c0
#point at base of interupt controller
lis r2,0x8180

li r4,0x36        #write 1 in the 26th, 27th, 29th and 30th
stw r4,0(r10)   #of the TSCR0 register to set the
                #parameters specified below
li r4,0x2
#pulse width modulation
stw r4,IER(r2) # set timer interupts to 'on'
lis r4, 2000@h   # internal base clock, @ 100MHz
ori r4,r4,2000@l # for period 20us, 100E6 *20E-6
stw r4, TLR0(r10) # give timer0 r4's value

#50% duty cycle:
lis r25, 2000@h   # D=Fsys/Finput
ori r25,r25,2000@l #  =>Fin=R25 =100E6/50%
stw r25, TLR1(r10) # give timer1 r25
```

Flags can be cleared at 0x500 the same way as for the uart and button interrupts.

Part8. Pit interrupts

The PIT (programmable interval timer) is much easier to use than the timer 1 or timer 2 and has the added benefit of living at evpr + 0x1000 so if you are using multiple interrupts you don't have to worry about checking what caused the interrupts.

Figure 8-4: **Timer-Control Register (TCR)**

The TCR is a privileged SPR with an address of 986 (0x3DA). It is read and written using the **mfspr** and **mtspr** instructions.

Table 8-3: **Timer-Control Register (TCR) Field Definitions**

Bit	Name	Function	Description
0:1	WP	Watchdog Period 00—2^{17} clocks 01—2^{21} clocks 10—2^{25} clocks 11—2^{29} clocks	Specifies the period for a watchdog-timer event.
2:3	WRC	Watchdog Reset Control 00—No reset 01—Processor reset 10—Chip reset 11—System reset	Specifies the type of reset that occurs as a result of a watchdog-timer event. After a bit is set in the WRC field, it cannot be cleared by software. Only a reset can clear the bit. This prevents errant code from disabling watchdog resets.
4	WIE	Watchdog-Interrupt Enable 0—Disabled 1—Enabled	Enables and disables watchdog interrupts.
5	PIE	PIT-Interrupt Enable 0—Disabled 1—Enabled	Enables and disables programmable-interval timer interrupts.
6:7	FP	FIT Period 00—2^{9} clocks 01—2^{13} clocks 10—2^{17} clocks 11—2^{21} clocks	Specifies the period for a fixed-interval timer event.

Table 8-3: **Timer-Control Register (TCR) Field Definitions *(Continued)***

Bit	Name	Function	Description
8	FIE	FIT-Interrupt Enable 0—Disabled 1—Enabled	Enables and disables fixed-interval timer interrupts.
9	ARE	Auto-Reload Enable 0—Disabled 1—Enabled	Enables and disables the programmable-interval timer auto-reload mode.
10:31		Reserved	

The PIT TCR (timer control register) is used for setting the pit, the watchdog, and the FIT. The watchdog timer isn't used in this lab, so we won't worry about it. In the next section I will deal with setting the PIT and FIT combined, for now, let's look at the PIT. The PIT lives at 987 (0x3DB) and can time values loaded by same method as EVPR. The relationship for the PIT is the same as the timers, 1000 =10us. To give commands to the TCR (0x3DA) and TSR (0x3D8), the .set and mtspr works well. Shown below is the bit information for the TSR. The TSR is used basically to clear flags.

Bit	Name	Function	Description
0	ENW	Enable Next Watchdog 0—Next watchdog time-out sets TSR[ENW]=1 1—Next watchdog time-out determined by TSR[WIS]	Enables the watchdog-timer event. See **Watchdog-Timer Events**, page 242, for more information.
1	WIS	Watchdog-Interrupt Status 0—No interrupt occurred 1—Interrupt occurred	Specifies whether a watchdog interrupt occurred, or could have occurred had it been enabled.
2:3	WRS	Watchdog Reset Status 00—No reset 01—Processor reset 10—Chip reset 11—System reset	Specifies the type of reset that occurred as a result of a watchdog-timer event, if the event caused a reset.

Bit	Name	Function	Description
4	PIS	PIT-Interrupt Status 0—No interrupt pending 1—Interrupt is pending	If programmable-interval timer interrupts are disabled, this bit specifies whether a PIT interrupt is pending. If they are enabled, the bit specifies whether a PIT interrupt occurred.
5	FIS	FIT-Interrupt Status 0—No interrupt pending 1—Interrupt is pending	If fixed-interval timer interrupts are disabled, this bit specifies whether a FIT interrupt is pending. If they are enabled, the bit specifies whether a FIT interrupt occurred.
6:31		Reserved	

Let's look at some example code for the PIT:

```
.org 0x1000  # section where PIT works

# code for when PIT interrupt happends

lis r4,0x0800  # bit 4 set to PIS clear
mtspr TSR, r4  # write to clear PIS
rfi



################################
.org 0x2000

.set EVPR,982

.set PIT, 987
```

```
.set TCR, 0x3DA

.set TSR,0x3D8

#interrupt initialization
li r4, 0       # for placing evpr at 0x00000000
mtspr EVPR, r4 # place value in evpr
lis r4, 0x01FCA055@h # number to decrement (1/3 of sec)
ori r4,r4,0x01FCA055@l
mtspr PIT,r4   # put in pit
lis r4,0x0440  # pit intrpt enable, reload enable
mtspr TCR, r4  # send to timer control reg
lis r4,0x0800  # bit 4 set to PIS clear
mtspr TSR, r4  # write to clear PIS
wrteei 1

main:

stickygoo : b stickygoo # do nothing forever and ever and ever..
```

Part9. FIT interrupts

Hopefully you aren't having one by trying to use this. The FIT uses the same datasheet info as the PIT, the main difference here is that the FIT lives at evpr +0x1010 so if you are using the pit and the fit, you must branch away as soon as you enter the PIT. Here is some initialization code for using both the PIT, FIT, and PWM

```
.set TCSR0, 0x0
.set TLR0, 0x4
.set TCR0, 0x08
.set TCSR1, 0x10
.set TLR1, 0x14
.set TCR1, 0x18

# timer 1 base address
lis r4,PWMTIMER1
# timer 2 base address
lis r6,TIMER2

#configure EVPR with base address of zero
li r20,0
mtspr 0x3d6, r20

#PWM Setup
li r20,0xe
stw r20,IER(r3)

#initialize pwm timers
#timer0 is period, 4000 is 40 microseconds
li r20,4000
stw r20,TLR0(r4)
timer1 is duty cycle, 600 is 15%
stw r11,TLR1(r4)
```

```
#now configure TCSR0
li r20, 0x2f6
stw r20,TCSR0(r4)
li r20, 0x2d6      #this turns off auto-reload
stw r20,TCSR0(r4)

#now configure TCSR1 of timer1
li r20, 0x2a6
stw r20,TCSR1(r4)
li r20,0x286
stw r20,TCSR1(r4)

#now configure Timer2 for 2 secs on, 1 sec off
#load 2 seconds into TLR0
#17d78402 is 2 seconds
#1 sec off value is BEBC201

lis r20,0xd78402@h
ori r20,r20,0xd78402@l
stw r20,TLR0(r6)

li r20,0xf6
stw r20,TCSR0(r6)
li r20,0xd6
stw r20,TCSR0(r6)

#set up PIT value
li r20,0x3C4F
mtspr PIT, r20

#set up TCR
lis r20,0x6C0     # FIT and PIT
#other options
#lis r20,0x280 # FIT only
#lis r20,0x440  # PIT only, no autoreload
mtspr TCR,r20

#enable external interrupts
wrteei 1
```

Part10. TBR (time based register)

The TBR is the internal clock on which the other timers and clocks are based. It has a 64 bit value that is broken into two sections. The TBU (time base upper) holds bits 0-31 and the TBL (time base lower) holds bits 32-63. Since it takes 0xBEBC200 for the clock to pass one second at 200MHz, only the TBL and lower halfword of TBU will be of concern for most applications since the max time that can be stored is close to 2000 years. Despite the reference guide's long winded, drawn out talk about the TBR, the manipulation of the TBR can be achieved with the following four commands:

```
mttbl rA  # move the value of rA into TBL
mttbu rA  # move the value of rA into TBU
```

mftbl rB # place the current value of TBL into rB
mftbu rB # place the current value of TBU into rB

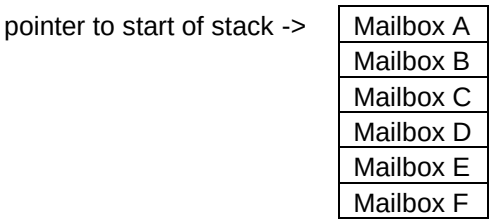
The move to commands can be used to place the value you will for the TBR to start counting from. In other words if I do, li rA, 0x8 and mttbl, then the TBR will start counting up from 0x00000000 00000008. If we wanted to check if 1 second had passed, we could add rA to the time for 1 second (0xBEBC200) and put it into say, rC. From there you just have a loop:

```
Loop: mftbl rB            # get the value of TBL
      cmpw 0,0,rC, rB  # is rC >= rB ?
      bf 0, onesecond    # if rC >= rB branch out of comparison
      b loop  # if not one second, keep checking

onesecond: b onesecond   # TBL is equal or greater than 0xBEBC208
```

Part11. Mailbox system

The mailbox system is a way of polling different interrupts and then setting the flag up on the mailbox if that interrupt needs service. You start by making a stack pointer to some place in memory, and then you decide what order you will make your boxes.



When you hit your ISR (interrupt service routine) at EVPR + 0x500, you could have several interrupts pending. The way to do this would be andi. the intc stat reg value with each bit and if it's 1, set the corresponding mailbox value to one. An example of code for this could be as follows:

```
.org 0x500
#Does UART need servicing?
lwz r23, ISR(r7)
andi. r24, r23, 8
bfl 2, setuartmail  # go set the uart mailbox to the value of 1

#Do push buttons need servicing?
lwz r23, ISR(r7)
andi. r24,r23,4
bfl 2, setgpiomail # go set the button mailbox to 1

#Does timer need servicing?
lwz r23, ISR(r7)
andi. r24,r23,1
bfl 2, settimer1mail # go set the timer 1 mailbox value to 1
```

After clearing the flags and returning from the interrupts, main can be looping with a code that checks each mailbox and services the corresponding "interrupt" code where the mailbox value is reset to 0. A basic set of code is shown for the above code.

```
Mainloop:
lwz r20, UARTMAIL(r1)
cmpwi 4,r20,1
btl 18, serviceuart

lwz r20, PBUTTONMAIL(r1)
cmpwi 3,r20,1
btl 14, servicepbutton

lwz r20,PITMAIL(r1)
cmpwi 6, r20, 1
btl 26, servicetimer1
```

Appendix 1. LEDs

LEDs are a great way to show otherwise hard to comprehend or test operations. For example, if we wanted to see how many times we had a uart interrupt, we could place code that increments the value of the LEDs every time we press a char and enter the intc section at 0x500. The basic commands to get up all LEDs are as follows:

```
# required
li rA, 0
stw rA,4(LED_base)  # turn LEDs "on"
# end of required

# wise practice, always load the initial value of the LEDs, usually this is zero
stw rA,0(LED_base)  # set LED's to initial value in program
```

From there, just write a value to the base of the LEDs. The main 32 led base values are listed here for you convenience:

LED 32 A lis 0x8143
LED 32 B lis 0x8144

Appendix 2. Branch link control

Branching is power, but bl (branch and link) provides a way to go off and do other code and then return with a blr to the same location. If all I want to do is decide whether to go do code set A or code set B, then branching is fine. But if I want to always do code A and depending on the case do set B before set A, then branch and link provides a way to do that. Think of a function call in C++. Let's say we have a function called sort that

sorted boxes by numbering, but only if there was more than 10 boxes. Let's look at pseudo code:

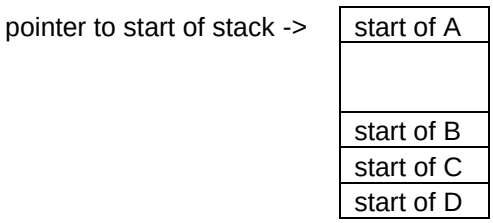
```
If (numboxes >= 10)
    Array.sort(); //go sort
Array.find (x)  // go find the x value in the array
```

The bl would be for array.sort() to sort the boxes.

Appendix 3. Register management

One of the big problems of large code mixed with interrupts and branchlinking is overusing registers. This can be easily solved however by a couple of methods. The operation, mr (move register) has the code of mr rA, rS (where s=source). This can be good for saving a copy below doing some code and then the state of the machine can be restored by doing basically the same thing, mr rS,rA. A big problem with the counter register (ctr) is it can be overused. A prevention to that is to use mfctr rA, do your code, then mtctr rA.

For large amounts of register over usage, stmw (store multiple words) and lmw (load multiple words) can be used along with a stack pointer and storing each "set" of registers into memory. You can even vary how much you store each time because one time you might need to store 15 registers one time and 5 another. Let's call each place in code we store a section starting with A. Each register takes up 0x4 in memory, so if you store stmw r28,0(pointer) for A, then the closest we could place set B would be stmw r24,16(pointer).



So a code example for section for two stores might be:

```
.org 0x500
stmw r24,0(r1)
# can now use registers 25-31
bl code
lmw r24,0(r1)
rfi

code:
# do some stuff
stmw r25,32(r1)
bl morestuff
```

```
lmw r25,32(r1)
blr
```

Appendix 4. Legal