Homework

• Reading
  – Intel 8254 Programmable Interval Timer (PIT) Data Sheet

• Machine Projects
  – Continue on MP3

• Labs
  – Continue in labs with your assigned section
Programmable Interval Timer

• This is just an overview – Read data sheet
• 8254 VLSI chip with three 16 bit counters
• Each counter:
  – Is decremented based on its own input clock
  – Is only decremented while its gate is active
  – Generates its own output clock =
    input clock / count length
  – Generates an interrupt when count value reaches zero
  – Automatically reloads initial value when it reaches zero
PIT Device (Timer 0)

- Simplest device: always is interrupting, every time it down counts to zero
- Can’t disable interrupts in this device!
- Can mask them off in the PIC
- We can control how often it interrupts
- Timer doesn’t keep track of interrupts in progress—just keeps sending them in
- We don’t need to interact with it in the ISR (but we do need to send an EOI to the PIC)
Use of PIT in MP3

• We use PIT counter 0 with 18.2 Hz output to generate an interrupt every 55 millisecs
• ISR increments “tickcount” each time
• We read the 16 bit counter to get value of the counter at 55 millisec / 64K resolution
• This gives us resolution of ____ microseconds (You are to figure that out as part of mp3)
Use of PIT in MP3

- MP3 gives you a start for the required C code in timepack_sapc.c. You add to it.
- You do not need to write the code for the PIT initialization or the interrupt service routine.
- However, you should study the C code to understand how it works:
  - Find where code disables and enables interrupts
  - Find the PIT initialization code
  - Find the PIT ISR code and see what it does
- Figure out how to read the count in progress to get the improved resolution required for mp3
Timer Interrupt Software

• Initialization
  – Disallow interrupts in CPU (cli)
    • Unmask IRQ0 in the PIC by ensuring bit 0 is 0 in the Interrupt Mask Register accessible via port 0x21
    • Set up interrupt gate descriptor in IDT, using irq0inthand
    • Set up timer downcount to determine tick interval
  – Allow interrupts (sti)

• Shutdown
  – Disallow interrupts (cli)
    •Disallow timer interrupts by masking IRQ0 in the PIC by making bit 0 be 1 in the Mask Register (port 0x21)
  – Allow interrupts (sti)
Timer Interrupts:
Interrupt Handler (Two Parts)

• `irq0inthand` – the outer assembly language interrupt handler
  – Save registers
  – Calls C function `irq0inthandc`
  – Restore registers
  – Iret

• `irq0inthandc` - the C interrupt handler
  – Issues EOI
  – Increase the tick count, or whatever is wanted
PIT Characteristics

• PIT chip has four I/O ports assigned to it:

\[ A_1 \quad A_0 \]

\[ \downarrow \quad \downarrow \]

– Timer 0 assigned port 40 = 0100 0000
– Timer 1 assigned port 41 = 0100 0001
– Timer 2 assigned port 42 = 0100 0010
– Control assigned port 43 = 0100 0011
– Chip selected by “chip select” and A₁-A₀
– Other signals include read, write, and data
Control Word Format

• Actually only a byte:

| SC1 | SC0 | RW1 | RW0 | M2 | M1 | M0 | BCD |

• SC1-SC0 select which counter to write/read
• RW1-RW0 to latch value or select which byte of count value
• M2-M0 determines which operating mode
• BCD specifies whether binary or BCD count
• Command formats found in datasheet
Using the PIT in C

• Refer to timer.h
  
  #define TIMER0_COUNT_PORT 0x40
  #define TIMER_CNTRL_PORT 0x43

  /* bits 6-7: */
  #define TIMER0 (0<<6)
  #define TIMER1 (1<<6)

  /* Bits 4-5 */
  #define TIMER_LATCH (0<<4)
  #define TIMER_SET_ALL (3<<4)

  /* Bits 1-3 */
  #define TIMER_MODE_RATEGEN (2<<1)

  /* Bit 0 */
  #define TIMER_BINARY_COUNTER 0
Programming the PIT

• Bits to initialize
  
  \texttt{TIMER0 \mid TIMER\_SET\_ALL \mid TIMER\_RATEGEN}  
  \texttt{\mid TIMER\_BINARY\_COUNTER}

  
• Output to the timer I/O port
  
  \texttt{outpt (TIMER\_CNTRL\_PORT, \ldots);}

  
• Then load the downcount
  
  \texttt{outpt (TIMER\_0\_COUNT\_PORT, count \& 0xFF);}  
  \texttt{\// LSByte}
  
  \texttt{outpt (TIMER\_0\_COUNT\_PORT, count >> 8);}  
  \texttt{\// MSByte}
What Are the PIT Modes?

• Mode 0: Count value loaded and countdown occurs on every clock signal; Out from counter remains low until count reaches 0 when it goes high

• Mode 2: Counts down from loaded value; when count has decremented to 1, OUT goes low for one clock pulse and then goes high again; count is reloaded and process repeats
What Are the PIT Modes? (Cont’d)

• Mode 3: Functions as a divide by n square wave generator, where n is the count value; OUT starts high and alternates between low and high.

    ▼ ▼ ▼ ▼ ▼ ▼ ▼ ▼
PIT Interface to Processor

- Internal counters are 16 bits (2 bytes)
- Only 8 pins connected to the data bus
- Must use a sequence of two one-byte “in” or “out” instructions to transfer full 16 bits
- Must “latch” count in progress to read a valid 16 bit value only 8 bits at a time
Reading the Count Values

• Want to read the count value without disturbing the count in progress

• Have to consider that the counter is changing while we are attempting to read it

• Best way to read the count is to use the counter latch command to temporarily latch the count

  outpt(TIMER_CNTRL_PORT, TIMER0 | TIMER_LATCH);
  count = inpt(TIMER0_COUNT_PORT);
  count |= (inpt(TIMER0_COUNT_PORT) << 8);

• Reading the count lets the latch again follow the count
Converting Counts to Real Time

• Can count from 1 to 64K, that is from approximately one microsecond to 55 milliseconds

• What if we want something longer than that?
  – Have to perform repeated counts
    • Example: 200 milliseconds = 3 * 55 + 35
    • Would need three full counts (called ticks) plus a partial count (called downcounts)
  
  – But how do you know when a tick has occurred?
    • Could poll the device
    • Better to use an interrupt

  – If interrupt occurs on every tick, which is counted, then the elapsed time in microseconds is approximately:
    • \([#\text{ticks} * 65536 + (\text{startcount} - \text{stopcount})]/1.193\)
Measuring the Time Interval

• Given
  – Event A happens 35K downcounts before tick 201
  – Event B happens 43K downcounts before tick 203

How much time has elapsed?

• Graphically:

Elapsed time =

202 ticks + (65K – 43K) downcounts
- (200 ticks + (65K – 35K) downcounts)

= 2 ticks – 8K down counts (where 1 tick = 64 * 1K down counts)
Main Memory Access

- One bus cycle per memory read or write
- Calculating bus cycles:
  - Instructions fetched 8 bytes at a time
    - Average: one eighth of a bus cycle per byte of an instruction
- Data reads/writes – byte, word, or long word at a time

- `movl %eax, %edx` reg/reg instruction (2 bytes)
  - .25R
- `movl %eax, total` reg/mem instruction* (5 bytes)
  - 0.625R+1W
- `movl %edx, total` reg/mem instruction (6 bytes)
  - 0.75R+1W

*eax as accumulator is special case
Cache Memory

- Located between CPU and main memory
  - Holds a copy of most recently accessed data
- L1 cache implemented with “hidden” registers
  - Not visible to assembly language programmer
- L2 cache implemented with SRAM memory
  - Faster access than DRAM main memory
  - Typically larger than L1 cache
- Either cache speeds up access to most recently and frequently accessed memory locations
- Avoids a bus cycle to main memory when data is found in cache (“cache hit”)
Cache Memory

Control Bus (M/IO#, W/R# and D/C# Signals)

Address Bus

L2 Cache (SRAM)

Main Memory (DRAM)

I/O Devices

Data Bus

Processor

Fetch and Execute Control

“Regular” Registers

L1 Cache (‘'Hidden’’ Registers)