Announcements

• Homework #2 (due next Tuesday in class!)
  – Make sure to encode your function in smallest possible opcode!
  – Think about it...

• Graduate Students: Feb 21st deadline for platforms!
Minute Quiz
• How do we keep time?
Virtual timers

• You never have enough timers.
  – Never.

• So what are we going to do about it?
  – How about we handle in software?
Virtual Timers

• Simple idea.
  – Maybe we have 10 events we might want to generate.
    • Just make a list of them and set the timer to go off for the *first* one.
      – Do that first task, change the timer to interrupt for the next task.
Problems?

• Only works for “compare” timer uses.
• Will result in slower ISR response time
  – May not care, could just schedule sooner…
Implementation issues

• Shared user-space/ISR data structure.
  – Insertion happens at least some of the time in user code.
  – Deletion happens in ISR.
    • We need critical section (disable interrupt)

• How do we deal with our modulo counter?
  – That is, the timer wraps around.
  – Why is that an issue?

• What functionality would be nice?
  – Generally one-shot vs. repeating events
  – Might be other things desired though

• What if two events are to happen at the same time?
  – Pick an order, do both…
• What data structure?
  – Data needs be sorted
    • Inserting one thing at a time
  – We always pop from one end
  – But we add in sorted order.
• What if we don’t have enough hardware timers?
• Virtual timer library interface

typedef void (*timer_handler_t)(void);

/* initialize the virtual timer */
void initTimer();

/* start a timer that fires at time t */
error_t startTimerOneShot(timer_handler_t handler, uint32_t t);

/* start a timer that fires every dt time interval*/
error_t startTimerContinuous(timer_handler_t handler, uint32_t dt);

/* stop timer with given handler */
error_t stopTimer(timer_handler_t handler);
typedef struct timer
{
    timer_handler_t handler;
    uint32_t time;
    uint8_t mode;
    timer_t* next_timer;
} timer_t;

timer_t* current_timer;

void initTimer() {
    setupHardwareTimer();
    initLinkedList();
    current_timer = NULL;
}

error_t startTimerOneShot(timer_handler_t handler, uint32_t t) {
    // add handler to linked list and sort it by time
    // if this is first element, start hardware timer
}

error_t startTimerContinuous(timer_handler_t handler, uint32_t dt) {
    // add handler to linked list for (now+dt), set mode to continuous
    // if this is first element, start hardware timer
}

error_t stopTimer(timer_handler_t handler) {
    // find element for handler and remove it from list
}
```c
_attribute__((__interrupt__)) void Timer1_IRQHandler() {
    timer_t * timer;
    MSS_TIM1_clear_irq();
    NVIC_ClearPendingIRQ(Timer1_IRQn);
    timer = current_timer;

    if ( current_timer->mode == CONTINUOUS ) {
        // add back into sorted linked list for (now+current_timer->time)
    }

    current_timer = current_timer->next_timer;

    if ( current_timer != NULL ) {
        // set hardware timer to current_timer->time
        MSS_TIM1_enable_irq();
    } else {
        MSS_TIM1_disable_irq();
    }

    (*timer->handler))(); // call the timer handler

    if ( timer->mode != CONTINUOUS ) {
        free(timer); // free the memory as timer is not needed anymore
    }
}
```
Outline

• Minute quiz

• Announcements

• Timers

• Memory Landscape

• Memory Architecture

• Non-volatile Memories

• Volatile Memories
External memory attaches to the processor via the external memory controller and bus

Atmel SAM3U
External memory bus transactions

- Read and write transactions
- Interfacing/handshaking
- Timing constraints
- Access speeds
- Wait states

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read cycle time</td>
<td>$t_{RC}$</td>
</tr>
<tr>
<td>Address access time</td>
<td>$t_{AA}$</td>
</tr>
<tr>
<td>Chip enable access time (CE)</td>
<td>$t_{AC}$</td>
</tr>
<tr>
<td>Output enable to output valid (OE)</td>
<td>$t_{OE}$</td>
</tr>
<tr>
<td>Output hold from address change</td>
<td>$t_{OH}$</td>
</tr>
<tr>
<td>Chip enable to output in low Z (CE)</td>
<td>$t_{LZ}$*</td>
</tr>
<tr>
<td>Output enable to output in low Z (OE)</td>
<td>$t_{OLZ}$*</td>
</tr>
<tr>
<td>Chip disable to output in High Z (CE)</td>
<td>$t_{HZ}$*</td>
</tr>
<tr>
<td>Output disable to output in High Z (OE)</td>
<td>$t_{OHZ}$*</td>
</tr>
</tbody>
</table>
• A: 20-bit address bus
• DQ: 8-bit data bus
• CE#: chip enable
• WE#: write enable
• OE#: output enable

<table>
<thead>
<tr>
<th>CE</th>
<th>OE</th>
<th>WE</th>
<th>DQ₁ to DQ₈</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>X</td>
<td>X</td>
<td>Not Selected</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>H</td>
<td>Read</td>
</tr>
<tr>
<td>L</td>
<td>X</td>
<td>L</td>
<td>Write</td>
</tr>
<tr>
<td>L</td>
<td>H</td>
<td>H</td>
<td>High Z</td>
</tr>
</tbody>
</table>
Basic categories of memory

- **Read-Only Memory (ROM)**
  - Can only be read (accessed)
  - Cannot be written (modified)
  - Contents are often set before ROM is placed into the system

- **Random-Access Memory (RAM)**
  - Can be read/written
  - Term used for historical reasons
  - Technically, ROMs are also random access

- **Volatile memory**
  - Loses contents when power is lost
  - Often stores program state, stack, and heap
  - In desktop/server systems, also stores program executable

- **Non-volatile memory**
  - Retains contents when power is lost
  - Used for boot code in almost every system
## Memory technologies landscape

<table>
<thead>
<tr>
<th></th>
<th>Volatile</th>
<th>Non-Volatile</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RAM</strong></td>
<td>Static RAM (SRAM)</td>
<td>EEPROM</td>
</tr>
<tr>
<td></td>
<td>Dynamic RAM (DRAM)</td>
<td>Flash Memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FRAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MRAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FRAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BBSRAM</td>
</tr>
<tr>
<td><strong>ROM</strong></td>
<td>n/a</td>
<td>Mask ROM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PROM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EPROM</td>
</tr>
</tbody>
</table>
Choosing the right memory requires balancing many tradeoffs

- **Volatility**: need to retain state during power down?
- **Cost**: wide range of absolute $ and $/bit costs
- **Organization**: 64Kbx1 or 8Kbx8?
- **Interface**
  - Serial or parallel?
  - Synchronous or asynchronous?
- **Access times**: critical for high-performance
- **Modify times**: critical for write-intensive workloads
- **Erase process**: at wire-line speed or 5 minutes in UV?
- **Erase granularity**: word, page, sector, chip?
• Minute quiz

• Announcements

• Memory Landscape

• Memory Architecture

• Non-volatile Memories

• Volatile Memories
Internal organization of memory is usually an array

Different memory types (e.g. SRAM vs DRAM) are distinguished by the technology used to implement the memory cell, e.g.:
- SRAM: 6T
- DRAM: 1T/1C

What should be the aspect ratio (# rows vs #cols)?
Memory Cell Sizes

Figure 1.1: Cell size comparison between flash memory and FeRAM. Flash memory has the smallest cell size among all of the nonvolatile memories. The data is from the 2002 International Technology Roadmap for Semiconductors.

Figure from: Ph.D. Dissertation Min She, UC Berkeley, 2003
Physical (on-chip) memory configuration

- Physical configurations are typically square
- Square minimizes length of (word line + bit line)
- Shorter length means
  - Shorter propagation time
  - Faster data access
  - Smaller $t_{rc}$ (read cycle time)

Exercise: Assume $n^2$ memory cells configured as
- $n$-by-$n$ square array. What is the worst case delay?
- $n^2$-by-1 rectangular. What is the worst case delay?

Exercise: Does wire length dominate access time?
- Assume propagation speed on chip is $2/3$ c ($2 \times 10^8$ m/s)
- Assume 1Mbit array is 1 cm x 1 cm
Logical (external) memory configuration

- External configurations are tall and narrow
  - More address lines (12 to 20+, typically)
  - Fewer data lines (8 or 16, typically)

- The narrower the configuration
  - The greater the pin efficiency
  - Adding one address pin cuts data pins in half
  - The easier the data bus routing

- Many external configurations for given capacity
  - 64 Kb = 64K x 1 (16 A + 1 D = 17 pins)
  - 64 Kb = 32K x 2 (15 A + 2 D = 17 pins)
  - 64 Kb = 16K x 4 (14 A + 4 D = 18 pins)
  - 64 Kb = 8K x 8 (13 A + 8 D = 21 pins)
  - 64 Kb = 4K x 16 (12 A + 16 D = 28 pins)
  - 64 Kb = 2K x 32 (11 A + 32 D = 43 pins)
Supporting circuitry is needed to address memory cell and enable reads and writes.

Control signals:
• Select chip
• Select memory cell
• Control read/write
• Map internal array to external configuration (4x4 → 16x1)
Refresher on the memory-bus interface

• Chip Select (CS#)
  - Enables device
  - Ignores all other inputs if CS# is not asserted

• Write Enable (WE#)
  - Enables write tri-state buffer
  - Store D0 at specified address

• Output Enable (OE#)
  - Enable read tri-state buffer
  - Drive D0 with value at specified address
Outline

• Minute quiz
• Announcements
• Memory Landscape
• Memory Architecture
• Non-volatile Memories
• Volatile Memories
Mask ROM

- The “simplest” memory technology
- Presence/absence of diode at each cell denote value
- Pattern of diodes defined by mask used in fab process
- Contents are fixed when chip is made; cannot be changed
- High upfront setup costs (mask costs)
- Small recurring marginal costs
- Good for applications where
  - Cost sensitivity drives design
  - Upgrading contents not an issue
  - e.g. boot ROM, CPU microcode
- Exercise:
  - What “value” does a diode encode?
  - What are the contents:
    - Where $A<2:0> = 101$?
    - Where $A<2:0> = 110$?
EPROM

- **Erasable Programmable Read-Only Memory**
- Constructed from floating gate FETs
  - Charge trapped on the FG erases cell
  - High voltage (13V +) applied to the control gate
    - “Writes” the cell with a 0
    - Allows FG charge to be dissipated
- Erasing means changing form 0 → 1
  - Uses UV light (not electrically!)
  - Electrons are trapped on a floating gate
- Writing means changing from 1 → 0
- Erase unit is the whole device
- Retains data for 10-20 years
- Not used much these days
- Costly because
  - Use of quartz window (UV transparent)
  - Use of ceramic package
- PROM (or OTP) is same, just w/o window
Flash Memory

- Electrically erasable (like EEPROM, unlike EPROM)
- Used in many reprogrammable systems these days
- Erase size is block (not word); can’t do byte modifications
- Erase circuitry moved out of cells to periphery
  - Smaller size
  - Better density
  - Lower cost
- Reads are like standard RAM
- Can “write” bits/words (actually, change from 1 → 0)
  - Write cycle is O(microseconds)
  - Slower than RAM but faster than EEPROM
  - To (re)write from 0 → 1, must explicitly erase entire block
    - Erase is time consuming O(milliseconds to seconds)
- Floating gate technology
  - Erase/write cycles are limited (10K to 100K, typically)
Outline

• Minute quiz

• Announcements

• Memory Landscape

• Memory Architecture

• Non-volatile Memories

• Volatile Memories
Static RAM

- SRAMs are volatile
- Basic cell
  - Bistable core
    - 4T: uses pullup resistors for M2, M4
    - 6T: uses P-FET for M2, M4
  - Access transistors
    - BL, BL# are provided to improve noise margin
- 6T is typically used (but has poor density)
- Fast access times $O(10 \text{ ns})$
- Read/write speeds are symmetric
- Read/write granularity is word
Dynamic RAM

- Requires only 1T and 1C per cell
- Outstanding density and low cost
- Compare to the 6T’s per SRAM cell
- Cost advantage to DRAM technology

- Small charges involved → relatively slow
  - Bit lines must be pre-charged to detect bits
  - Reads are destructive; internal writebacks needed

- Values must be refreshed periodically
  - Prevents charge from leaking away
  - Complicates control circuitry slightly
Questions?

Comments?

Discussion?

Submit Feedback: http://goo.gl/WkJRx