REVIEW: Embedded System Hardware

Embedded system hardware is frequently used in a loop ("hardware in a loop"):

- A/D converter
- sample-and-hold
- sensors
- information processing
- environment
- actuators
- display
- D/A converter
**REVIEW: Standard layout of sensor systems**

- **Sensor**: detects/measures entity and converts it to electrical domain
  - May entail ES-controllable actuation: e.g. charge transfer in CCD
- **Amplifier**: adjusts signal to the dynamic range of the A/D conversion
  - Often dynamically adjustable gain: e.g. ISO settings at digital cameras, input gain for microphones (sound or ultrasound), extremely wide dynamic ranges in seismic data logging
- **Sample + hold**: samples signal at discrete time instants
- **A/D conversion**: converts samples to digital domain

---

**Discretization of time**

\[ V_e \text{ is a mapping } \mathbb{R} \to \mathbb{R} \]

\[ V_x \text{ is a sequence of values or a mapping } \mathbb{Z} \to \mathbb{R} \]

Discrete time: sample and hold-devices.
Ideally: width of clock pulse \( \to 0 \)
Sample and Hold

Discretization of values: A/D-converters
1. Flash A/D converter (1)

- Basic element: analog comparator

- Output = `1` if voltage at input + exceeds that at input -.
- Output = `0` if voltage at input - exceeds that at input +.

- Idea:
  - Generate \( n \) different voltages by voltage divider (resistors), e.g. \( V_{\text{ref}}, \frac{3}{4} V_{\text{ref}}, \frac{1}{2} V_{\text{ref}}, \frac{1}{4} V_{\text{ref}} \).
  - Use \( n \) comparators for parallel comparison of input voltage \( V_x \) to these voltages.
  - Encoder to compute digital output.
Discretization of values: A/D-converters

1. Flash A/D converter (2)

- Parallel comparison with reference voltage
- **Applications**: e.g. in video processing

![Flash A/D Converter Diagram]

Discretization of values

2. Successive approximation

**Key idea**: binary search:
- Set MSB='1'
- if too large: reset MSB
- Set MSB-1='1'
- if too large: reset MSB-1
Successive approximation (2)

Digital-to-Analog (D/A) Converters

- Convert digital value to conductivity proportional to the digital value
Operational amplifier

- Use operational amplifier to convert conductivity to voltage: \( V = -\frac{V_{\text{ref}} R_2}{R_1} \)
Design Issues with Sensors

- Calibration
  - Relating measurements to the physical phenomenon
  - Can dramatically increase manufacturing costs
- Nonlinearity
  - Measurements may not be proportional to physical phenomenon
  - Correction may be required
  - Feedback can be used to keep operating point in the linear region
- Sampling
  - Aliasing
  - Missed events
- Noise
  - Analog signal conditioning
  - Digital filtering
  - Introduces latency

Aliasing

\[ e_s(t) = \sin \left( \frac{2 \pi t}{8} \right) + 0.5 \sin \left( \frac{2 \pi t}{4} \right) \]

- Periods of \( p=8,4,1 \)
- Indistinguishable if sampled at integer times, \( p=1 \)
**Aliasing**

**Nyquist criterion** (sampling theory): Aliasing can be avoided if we restrict the frequencies of the incoming signal to less than half of the sampling rate.

\[ p_s < \frac{1}{2} p_N \]

where \( p_N \) is the period of the “fastest” sine wave or

\[ f_s > 2 f_N \]

where \( f_N \) is the frequency of the “fastest” sine wave

\( f_N \) is called the **Nyquist frequency**, \( f_s \) is the **sampling rate**.

See e.g. [Oppenheim/Schafer, 2009]

---

**Graphics**

Images from (Wikimedia Commons)
Anti-aliasing filter

A filter is needed to remove high frequencies

- Ideal filter
- Realizable filter

Possible to reconstruct input signal?

- Assuming Nyquist criterion met
- Let \( \{t_s\}, s = \ldots, -1, 0, 1, 2, \ldots \) be times at which we sample \( g(t) \)
- Assume a constant sampling rate of \( 1/p_s \) (\( \forall s: p_s = t_{s+1} - t_s \)).
- According to sampling theory, we can approximate the input signal using the **Shannon-Whittaker interpolation**:

\[
   z(t) = \sum_{s=-\infty}^{\infty} y(t_s) \frac{p_s(t - t_s)}{\pi p_s(t - t_s)} \sin \left( \frac{\pi}{p_s} (t - t_s) \right)
\]

[Oppenheim, Schafer, 2009]
Weighting factor for influence of $y(t_s)$ at time $t$

$$sinc(t - t_s) = \frac{\sin\left(\frac{\pi}{p_s}(t - t_s)\right)}{\frac{\pi}{p_s}(t - t_s)}$$

Contributions from the various sampling instances
(Attempted) reconstruction of input signal

* Assuming 0-order hold

How to compute the $sinc(\ )$ function?

$$z(t) = \sum_{n=-\infty}^{\infty} y(t_n) \frac{\sin \frac{\pi}{f_s} (t-t_n)}{\frac{\pi}{f_s} (t-t_n)}$$

- Filter theory: The required interpolation is performed by an ideal low-pass filter ($sinc$ is the Fourier transform of the low-pass filter transfer function)

Filter removes high frequencies present in $y(t)$
How precisely are we reconstructing the input?

\[ z(t) = \sum_{s=-\infty}^{\infty} \frac{y(t_s)\sin\frac{\pi}{T_s}(t-t_s)}{\frac{\pi}{T_s}(t-t_s)} \]

- **Sampling theory:**
  - Reconstruction using \( \text{sinc}() \) is precise
  - However, it may be impossible to really compute \( z(t) \)

Limitations

- Actual filters do not compute \( \text{sinc}(\cdot) \)
  - In practice, filters are used as an approximation.
  - Computing good filters is an art itself!
- All samples must be known to reconstruct \( e(t) \) or \( g(t) \).
  - Waiting indefinitely before we can generate output!
  - In practice, only a finite set of samples is available.
- Actual signals are never perfectly bandwidth limited.
- Quantization noise cannot be removed.
### Actuators and output

- Huge variety of actuators and outputs
- Two base types:
  - analogue drive (requires D/A conversion, unless on/off sufficient)
    - CRTs, speakers, electrical motors with collector
    - electromagnetic (e.g., coils) or electrostatic drives
    - piezo drives
  - digital drive (requires amplification only)
    - LEDs
    - stepper motors
    - relais, electromagnetic valve (if actuation slope irrelevant)

### Micromotors

- © MCNC
- (TU Berlin)
Interfaces

- Pulse width modulation (PWM)
- General-Purpose Digital I/O (GPIO)

- Parallel
  - Multiple data lines transmitting data
  - Ex: PCI, ATA, CF cards, Bus

- Serial
  - Single data line transmitting data
  - Ex: USB, SATA, SD cards,
Example Using a Serial Interface

In an Atmel AVR 8-bit microcontroller, to send a byte over a serial port, the following C code will do:

```c
while(!(UCSR0A & 0x20));
UDR0 = x;
```

- `x` is a variable of type uint8.
- `UCSR0A` and `UDR0` are variables defined in header.
- They refer to memory-mapped registers.

Send a Sequence of Bytes

```c
for(i = 0; i < 8; i++) {
    while(!(UCSR0A & 0x20));
    UDR0 = x[i];
}
```

How long will this take to execute? Assume:
- 57600 baud serial speed.
- 8/57600 = 139 microseconds.
- Processor operates at 18 MHz.
Each while loop will consume 2500 cycles.
Input Mechanisms in Software

- **Polling**
  - Main loop checks each I/O device periodically.
  - If input is ready, processor initiates communication.

- **Interrupts**
  - External hardware alerts the processor that input is ready.
  - Processor suspends what it is doing, invokes an interrupt service routine (ISR).

Timed Interrupt

- Processor Setup Code
- Processor checks I/O control registers for status of peripheral 1
- Not Ready
- Processor checks I/O control registers for status of peripheral 2
- Not Ready
- Processor checks I/O control registers for status of peripheral 3
- Not Ready
- Processor Setup Code
- Register Interrupt Service Routine
- Initialize Timer
- Execute Task Code
- Update Tick / Sample
- Processor jumps to ISR
- Processor executes task code
- Resume
- Run Interrupt Service Routine
- Timer
- Reset timer
- When timer expires, interrupt processor
- Run Interrupt Service Routine
Example:
Do something for 2 seconds then stop

```c
volatile uint timer_count = 0;
void ISR(void) {
    if(timer_count != 0) {
        timer_count--;
    }
}
int main(void) {
    // initialization code
    SysTickIntRegister(&ISR);
    // other init
    timer_count = 2000;
    while(timer_count != 0) {
        // ... code to run for 2 seconds
    }
}
```

**Example:**
Do something for 2 seconds then stop

- **volatile:** C keyword to tell the compiler that this variable may change at any time, not (entirely) under the control of this program.
- **static variable:** declared outside `main()` puts them in statically allocated memory (not on the stack)
- **Interrupt service routine**
- Registering the ISR to be invoked on every SysTick interrupt
**Embedded System Hardware**

Embedded system hardware is frequently used in a loop (“hardware in a loop”):

- A/D converter (sample-and-hold)
- information processing
- display
- D/A converter
- actuators
- sensors
- (physical) environment
- cyber-physical systems

---

**Microcontrollers**

- Integrate several components of a microprocessor system onto one chip: CPU, Memory, Timer, IO
- Low cost, small packaging
- Easy integration with circuits
- Single-purpose
Application Specific Circuits (ASICS) or Full Custom Circuits

- Approach suffers from
  - long design times,
  - lack of flexibility (changing standards) and
  - high costs (e.g. Mill. $ mask costs).

- Custom-designed circuits necessary
  - if ultimate speed or
  - energy efficiency is the goal and
  - large numbers can be sold.

Energy

© Hugo De Man, IMEC, Philips, 2007
Low Power vs. Low Energy Consumption

- Minimizing **power consumption** important for
  - the design of the power supply
  - the design of voltage regulators
  - the dimensioning of interconnect
  - short term cooling
- Minimizing **energy consumption** important due to
  - restricted availability of energy (mobile systems)
    - limited battery capacities (only slowly improving)
    - very high costs of energy (solar panels, in space)
  - cooling
    - high costs
    - limited space
  - dependability
    - long lifetimes, low temperatures

Dynamic power management (DPM)

**Example: STRONGARM SA1100**

- **RUN**: operational
- **IDLE**: a SW routine may stop the CPU when not in use, while monitoring interrupts
- **SLEEP**: Shutdown of on-chip activity

<table>
<thead>
<tr>
<th>State</th>
<th>Power Consumption</th>
<th>Power Fault Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUN</td>
<td>400mW</td>
<td>90µs</td>
</tr>
<tr>
<td>IDLE</td>
<td>50mW</td>
<td>Power fault signal</td>
</tr>
<tr>
<td>SLEEP</td>
<td>160µW</td>
<td></td>
</tr>
</tbody>
</table>

BF - ES
Fundamentals of dynamic voltage scaling (DVS)

Power consumption of CMOS circuits (ignoring leakage):

\[ P = \alpha \cdot C_L \cdot V_d^2 \cdot f \]

- \( P \): power consumption
- \( \alpha \): switching activity
- \( C_L \): load capacitance
- \( V_d \): supply voltage
- \( f \): clock frequency

Delay for CMOS circuits:

\[ \tau = k \cdot C_L \cdot \frac{V_{dd}}{(V_{dd} - V_s)^2} \]

- \( \tau \): delay
- \( k \): constant
- \( C_L \): load capacitance
- \( V_{dd} \): supply voltage
- \( V_s \): threshold voltage

Variable-voltage/frequency example: INTEL Xscale

OS should schedule distribution of the energy budget.
Low voltage, parallel operation more efficient than high voltage, sequential operation

Basic equations
- Power: $P \sim V_{dd}^2$
- Maximum clock frequency: $f \sim V_{dd}$
- Energy to run a program: $E = P \times t$, with: $t$ = runtime
- Time to run a program: $t \sim 1/f$

Changes due to parallel processing, with $\alpha$ operations per clock:
- Clock frequency reduced to: $f' = f / \alpha$
- Voltage can be reduced to: $V_{dd}' = V_{dd} / \alpha$
- Power for parallel processing: $P' = P / \alpha^2$ per operation,
- Power for $\alpha$ operations per clock: $P' = \alpha \times P / \alpha = P / \alpha$
- Time to run a program is still: $t' = t$
- Energy required to run program: $E' = P' \times t = E / \alpha$

Argument in favour of voltage scaling, VLIW processors, and multi-cores

Application: VLIW processing and voltage scaling in the Crusoe processor

- $V_{dd}$: 32 levels (1.1V - 1.6V)
- Clock: 200MHz - 700MHz in increments of 33MHz

Scaling is triggered when CPU load change is detected by software (~1/2 ms).
- More load: Increase of supply voltage (~20 ms/step), followed by scaling clock frequency
- Less load: reduction of clock frequency, followed by reduction of supply voltage

Worst case (1.1V to 1.6V $V_{dd}$, 200MHz to 700MHz) takes 280 ms
Result (as published by transmeta)

Pentium

Crusoe

Running the same multimedia application.

Digital Signal Processing (DSP)

Example: Filtering

\[ x_s = \sum_{k=0}^{n-1} W_{s-k} \cdot a_k \]

Signal at \( t = t_s \) (sampling points)
Filtering in digital signal processing

\[ x_t = \sum_{k=0}^{n-1} w_{s-k} a_k \]

outer loop over sampling times \( t \)

\[
\begin{align*}
&\{ \text{MR:=0; A1:=1; A2:=n-1;} \\
&\quad \text{MX:=x[n-1]; MY:=a[0];} \\
&\quad \text{for (k:=0; k <= (n-1); k++)} \\
&\quad \quad \{ \text{MR:=MR + MX * MY;} \\
&\quad \quad \quad \text{MX:=w[A2]; MY:=a[A1];} \\
&\quad \quad \quad \text{A1++; A2--;} \\
&\quad \} \\
&\quad x[s]:=MR;
\end{align*}
\]

DSP-Processors: multiply/accumulate (MAC) and zero-overhead loop (ZOL) instructions

Multiply/accumulate (MAC) instruction

Zero-overhead loop (ZOL) instruction preceding MAC instruction. Loop testing done in parallel to MAC operations.
Heterogeneous registers

Example (ADSP 210x):

- Address registers A0, A1, A2...
- Address generation unit (AGU)

Different functionality of registers An, AX, AY, AF, MX, MY, MF, MR

Separate address generation units (AGUs)

Example (ADSP 210x):

- Data memory can only be fetched with address contained in A,
- but this can be done in parallel with operation in main data path (takes effectively 0 time).
- A := A ± 1 also takes 0 time,
- same for A := A ± M;
- A := <immediate in instruction> requires extra instruction.
Modulo addressing

Modulo addressing:
\[ Am++ \equiv Am := (Am + 1) \mod n \]
(implements ring or circular buffer in memory)

- x\[t1-1]\]
- x\[t1\]
- x\[t1-n+1]\]
- x\[t1-n+2]\]

n most recent values

Memory, t=t1

sliding window

Memory, t2=t1+1

\[ x[t1-1] \]
\[ x[t1] \]
\[ x[t1-n+1] \]
\[ x[t1-n+2] \]

Saturating arithmetic

- Returns largest/smallest number in case of over/underflows

- Example:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0111</td>
</tr>
<tr>
<td>b</td>
<td>+ 1001</td>
</tr>
<tr>
<td>standard wrap around arithmetic</td>
<td>(1)0000</td>
</tr>
<tr>
<td>saturating arithmetic</td>
<td>1111</td>
</tr>
<tr>
<td>(a+b)/2:</td>
<td>correct 1000</td>
</tr>
<tr>
<td></td>
<td>wrap around arithmetic 0000</td>
</tr>
<tr>
<td></td>
<td>saturating arithmetic + shifted 0111 “almost correct”</td>
</tr>
</tbody>
</table>

- Appropriate for DSP/multimedia applications:
  - No timeliness of results if interrupts are generated for overflows
  - Precise values less important
  - Wrap around arithmetic would be worse.
Multimedia-Instructions/Processors

- Multimedia instructions exploit that many registers, adders etc are quite wide (32/64 bit),
- whereas most multimedia data types are narrow (e.g. 8 bit per color, 16 bit per audio sample per channel)
  - 2-8 values can be stored per register and added. E.g.:

```
+ 4 additions per instruction; carry disabled at word boundaries.
```

Key idea of very long instruction word (VLIW) computers

- Instructions included in long instruction packets. Instruction packets are assumed to be executed in parallel.
- Fixed association of packet bits with functional units.
Very long instruction word (VLIW) architectures

- Very long instruction word ("instruction packet") contains several instructions, all of which are assumed to be executed in parallel.
- Compiler is assumed to generate these "parallel" packets
- Complexity of finding parallelism is moved from the hardware (RISC/CISC processors) to the compiler; ideally, this avoids the overhead (silicon, energy, ..) of identifying parallelism at run-time.

A lot of expectations into VLIW machines

- Explicitly parallel instruction set computers (EPICs) are an extension of VLIW architectures: parallelism detected by compiler, but no need to encode parallelism in 1 word.

Large # of delay slots, a problem of VLIW processors

- Pipeline stages: instruction fetch, instruction decode, instruction execute, register writeback
- Example instructions: add, sub, and, or, sub, mult, xor, div, ld, st, mv, beq
- Large number of delay slots, a problem for VLIW processors
Large # of delay slots, a problem of VLIW processors

The execution of many instructions has been started before it is realized that a branch was required.
Nullifying those instructions would waste compute power
Executing those instructions is declared a feature, not a bug.
How to fill all “delay slots” with useful instructions?
Avoid branches wherever possible.