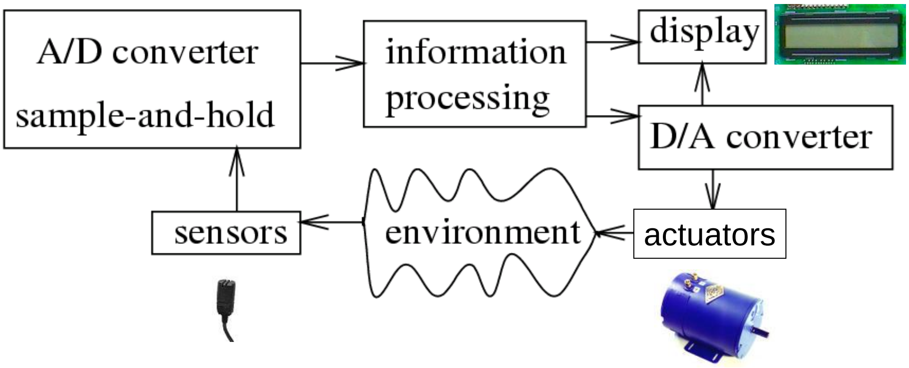
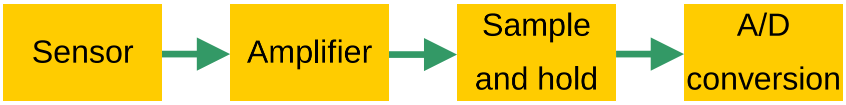
Embedded Systems 12

REVIEW: Embedded System Hardware

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



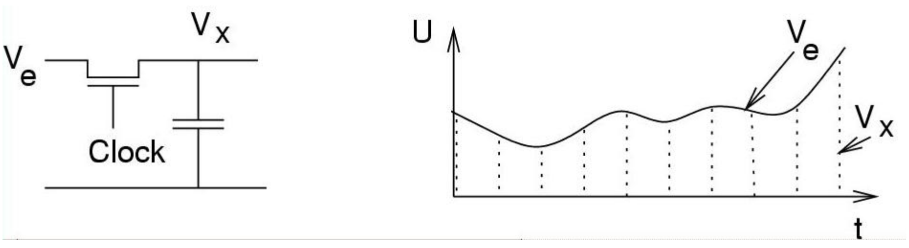
REVIEW: Standard layout of sensor systems


Sensor → Amplifier → Sample and hold → A/D conversion

- 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 is a mapping $\mathbb{R} \to \mathbb{R}$

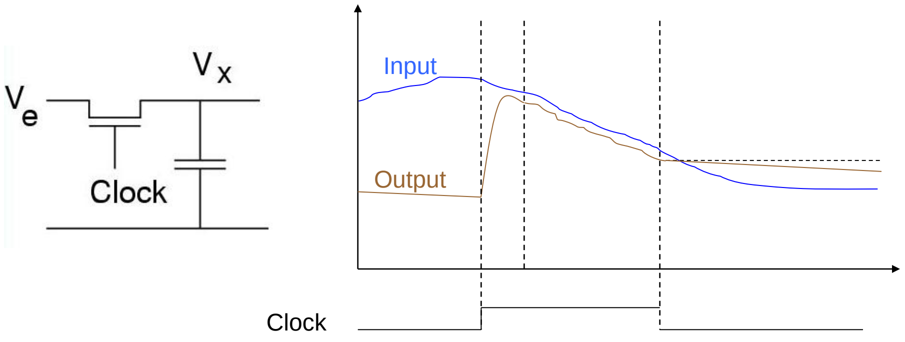


V_x is a **sequence** of values or a mapping $\mathbb{Z} \to \mathbb{R}$

Discrete time: sample and hold-devices.

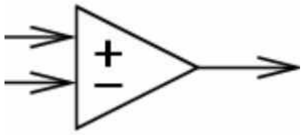
Ideally: width of clock pulse -> 0

Sample and Hold



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

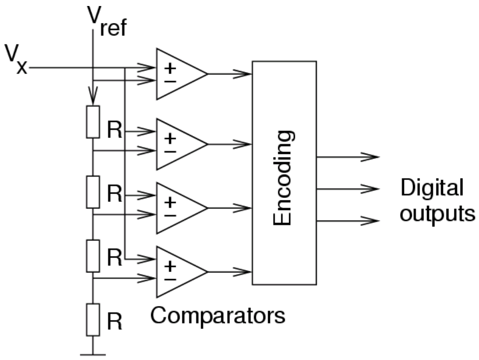
- Basic element: analog comparator



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

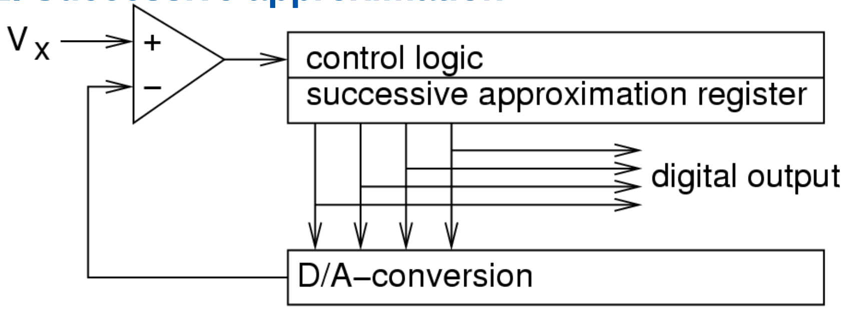
- Idea:
  - Generate *n* different voltages by voltage divider (resistors), e.g. V_{ref}, ¾ V_{ref}, ½ V_{ref}, ¼ V_{ref}.
  - Use *n* comparators for parallel comparison of input voltage V_x to these voltages.
  - Encoder to compute digital output.

Discretization of values: A/D-converters
1. Flash A/D converter (2)



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

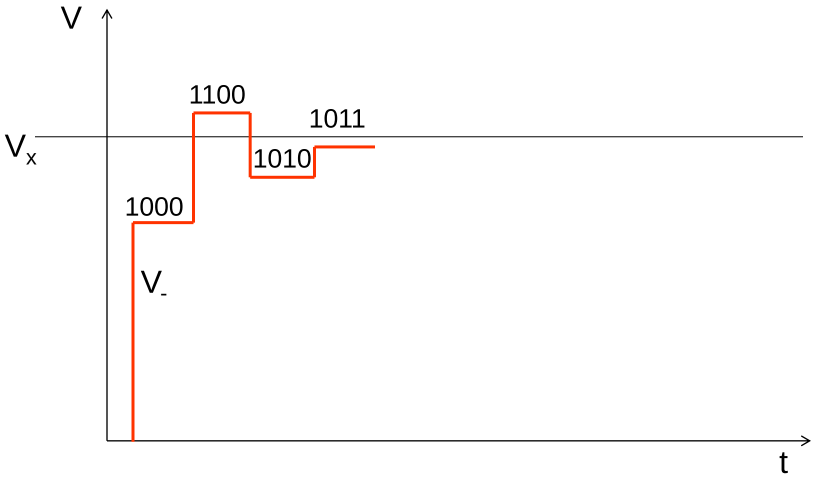
Discretization of values
2. Successive approximation



Key idea: binary search:

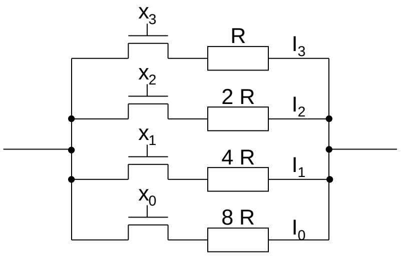
- Set MSB='1'
- if too large: reset MSB
- Set MSB-1='1'
- if too large: reset MSB-1

Successive approximation (2)



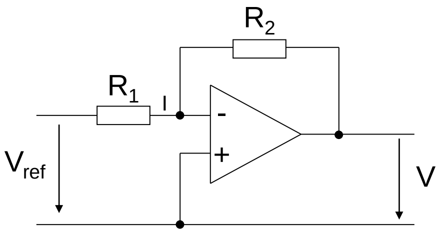
Digital-to-Analog (D/A) Converters

- Convert digital value to conductivity proportional to the digital value

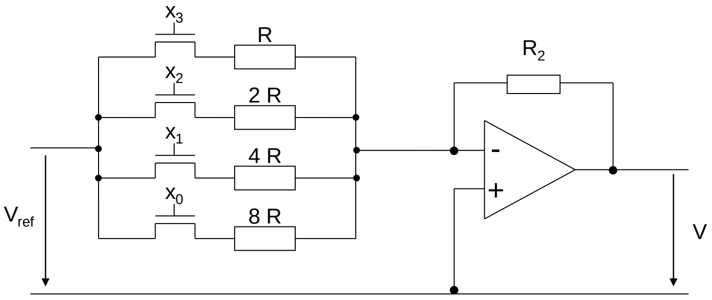


Operational amplifier

- Use operational amplifier to convert conductivity to voltage: $V = - V_{ref} R_2 / R_1$



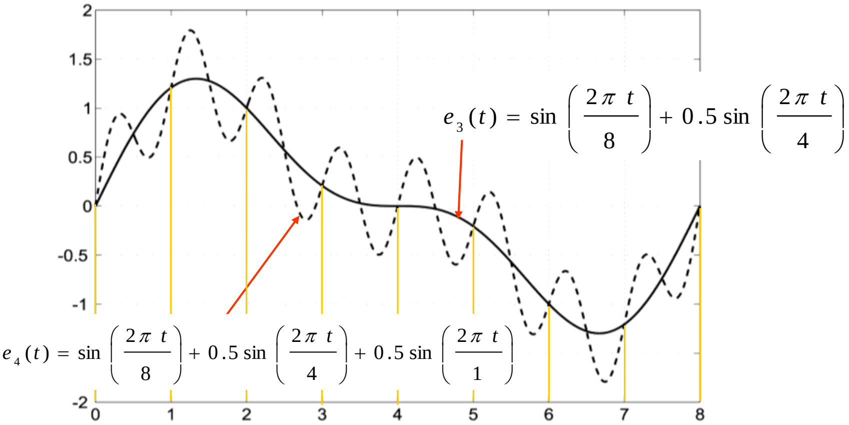
Digital-to-Analog (D/A) Converters (3)



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



$$e_3(t) = \sin\left(\frac{2\pi t}{8}\right) + 0.5\sin\left(\frac{2\pi t}{4}\right)$$

$$e_4(t) = \sin\left(\frac{2\pi t}{8}\right) + 0.5\sin\left(\frac{2\pi t}{4}\right) + 0.5\sin\left(\frac{2\pi t}{1}\right)$$

- Periods of p=8,4,1
- Indistinguishable if sampled at integer times, p_s=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.

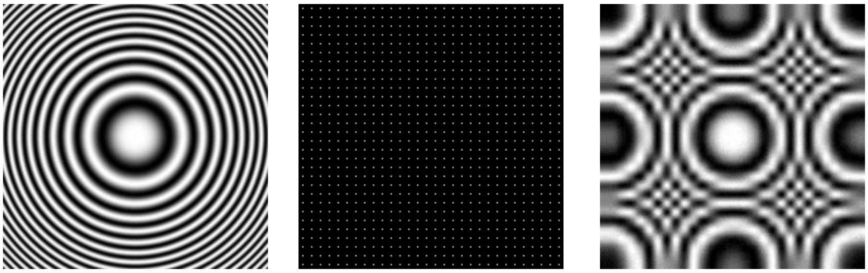
$p_s < \frac{1}{2}\, p_N$ where p_N is the period of the "fastest" sine wave

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]

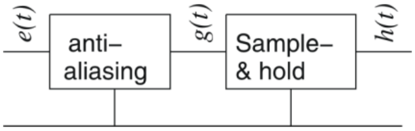
Graphics

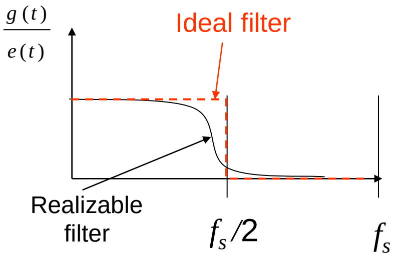


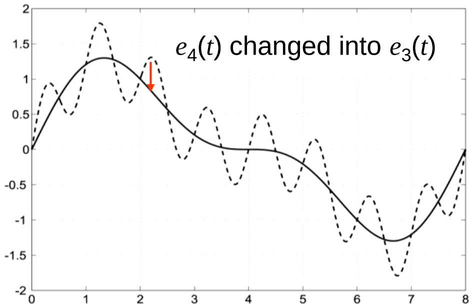
(Wikimedia Commons)

Anti-aliasing filter

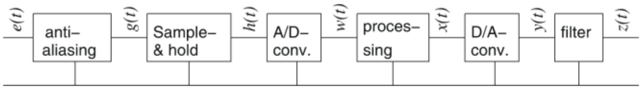
A filter is needed to remove high frequencies

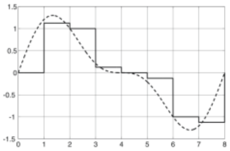






Possible to reconstruct input signal?





- Assuming Nyquist criterion met
- Let $\{t_s\}$, $s = ...,-1,0,1,2, ...$ 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:**

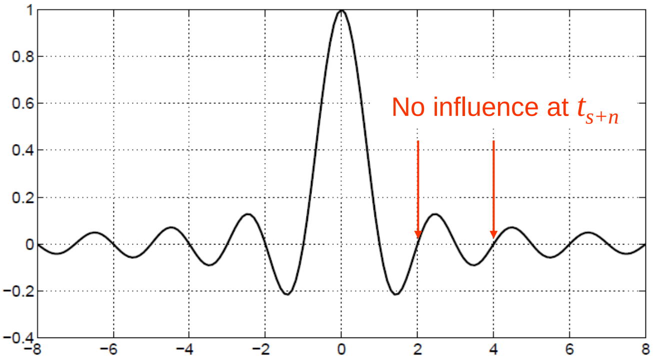
$$z(t) = \sum_{s=-\infty}^{\infty} \frac{y(t_s) sin\frac{\pi}{p_s}(t-t_s)}{\frac{\pi}{p_s}(t-t_s)}$$

Weighting factor for influence of $y(t_s)$ at time t

[Oppenheim, Schafer, 2009]

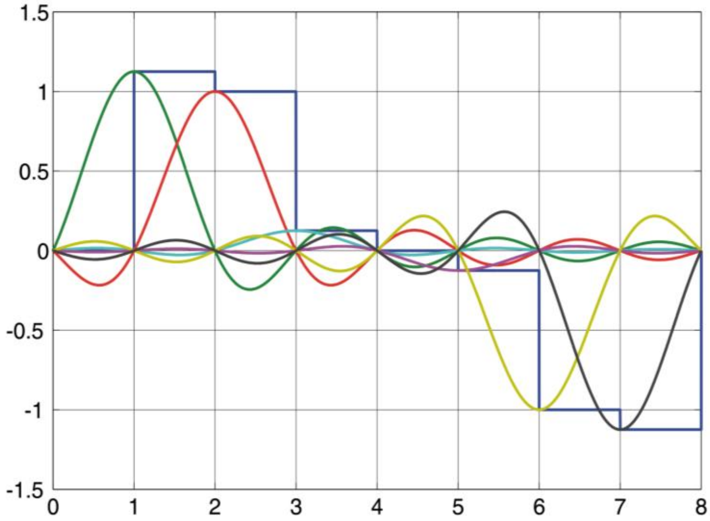
Weighting factor for influence of $y(t_s)$ at time t

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

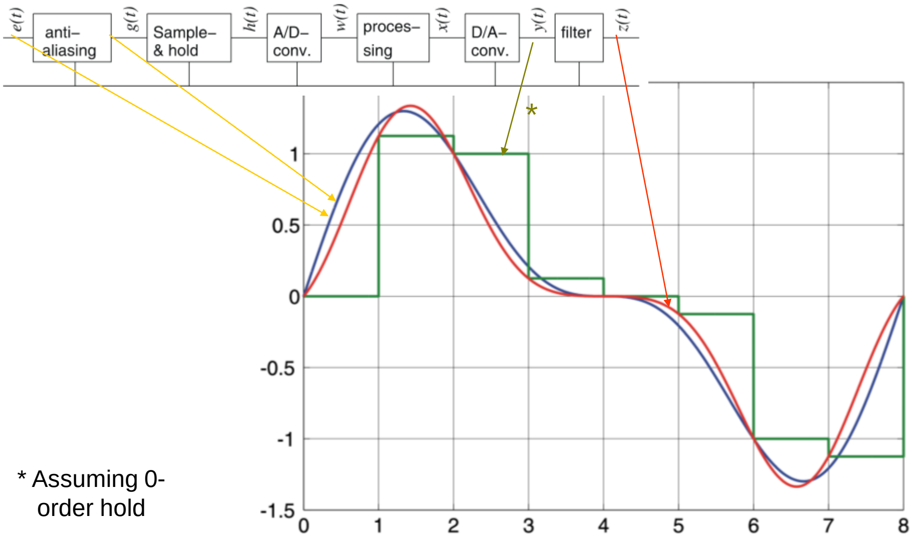


No influence at t_{s+n}

Contributions from the various sampling instances



(Attempted) reconstruction of input signal

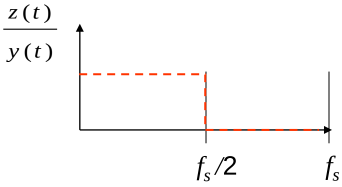


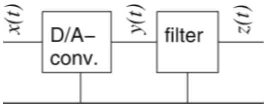
* Assuming 0-order hold

How to compute the *sinc*( ) function?

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

- **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?

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

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

Limitations

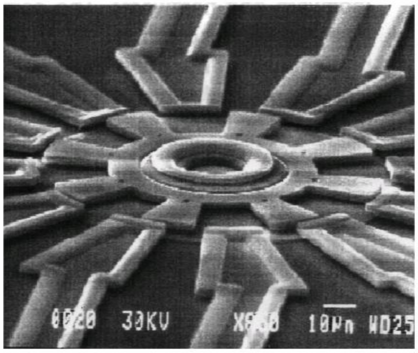
$$z(t) \quad = \quad \sum_{s=-\infty}^{\infty} \frac{y(t_s) sin \frac{\pi}{T_s}(t-t_s)}{\frac{\pi}{T_s}(t-t_s)}$$

- Actual filters do not compute $sinc(\,)$
  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.

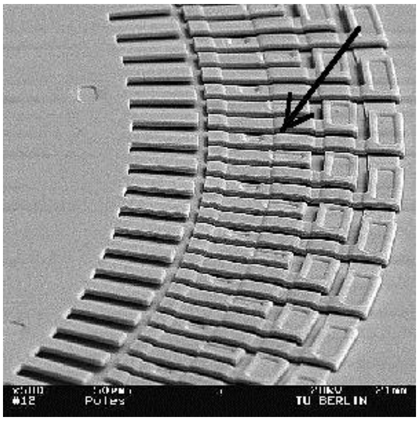
Actuators and output

- 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

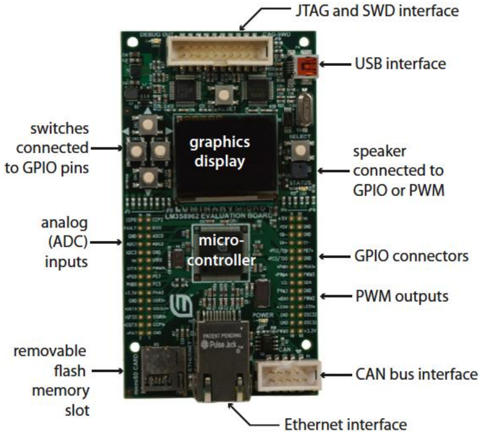


(© MCNC)



(TU Berlin)

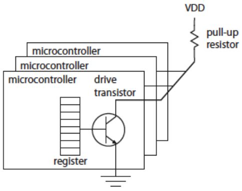
Interfaces

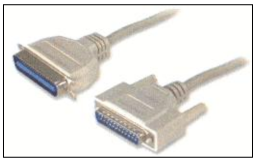


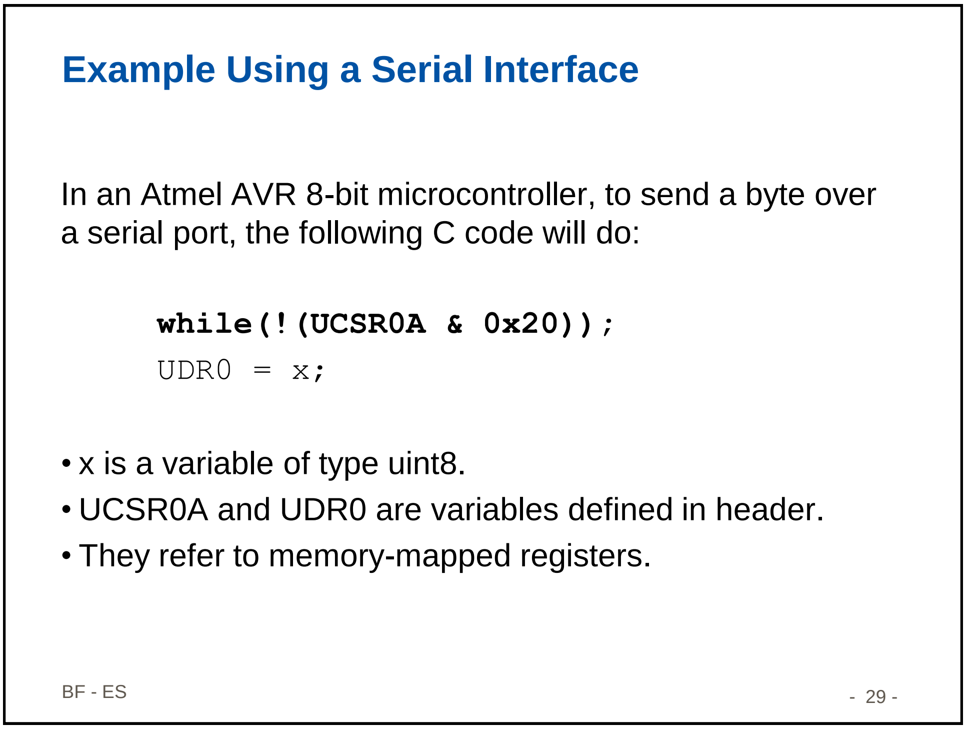
Stellaris®LM3S8962 evaluation board

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,





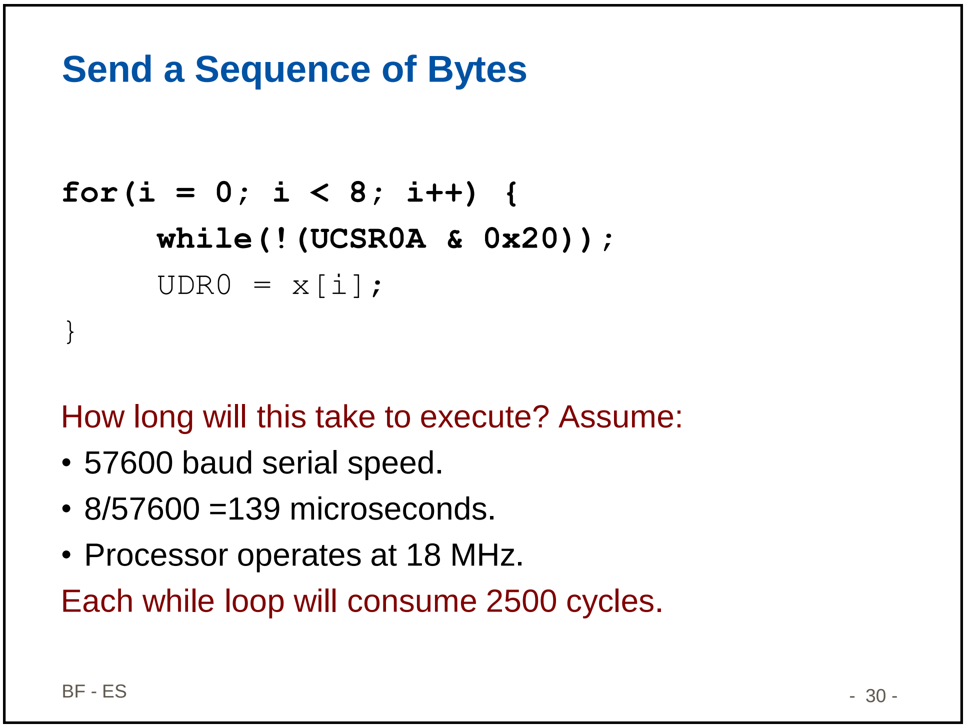


Example Using a Serial Interface

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

```
     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

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

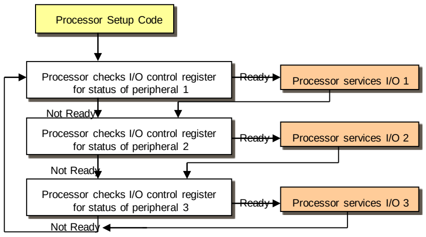
How long will this take to execute? Assume:

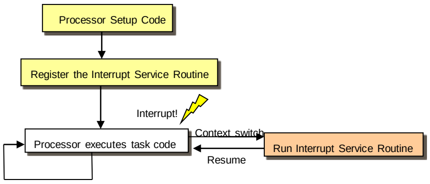
- 57600 baud serial speed.
- 8/57600 =139 microseconds.
- Processor operates at 18 MHz.

Each while loop will consume 2500 cycles.

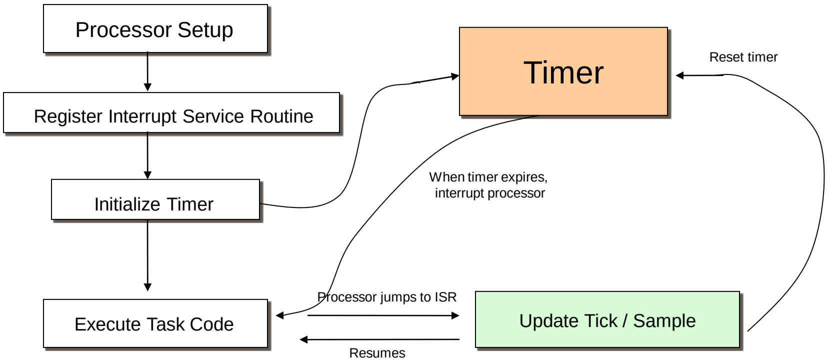
Input Mechanisms in Software

- 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).


Processor Setup Code
Processor checks I/O control register for status of peripheral 1
Ready
Processor services I/O 1
Not Ready
Processor checks I/O control register for status of peripheral 2
Ready
Processor services I/O 2
Not Ready
Processor checks I/O control register for status of peripheral 3
Ready
Processor services I/O 3
Not Ready


Processor Setup Code
Register the Interrupt Service Routine
Interrupt!
Processor executes task code
Context switch
Resume
Run Interrupt Service Routine

Timed Interrupt



Example:
Do something for 2 seconds then stop

```
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
  }
}
```

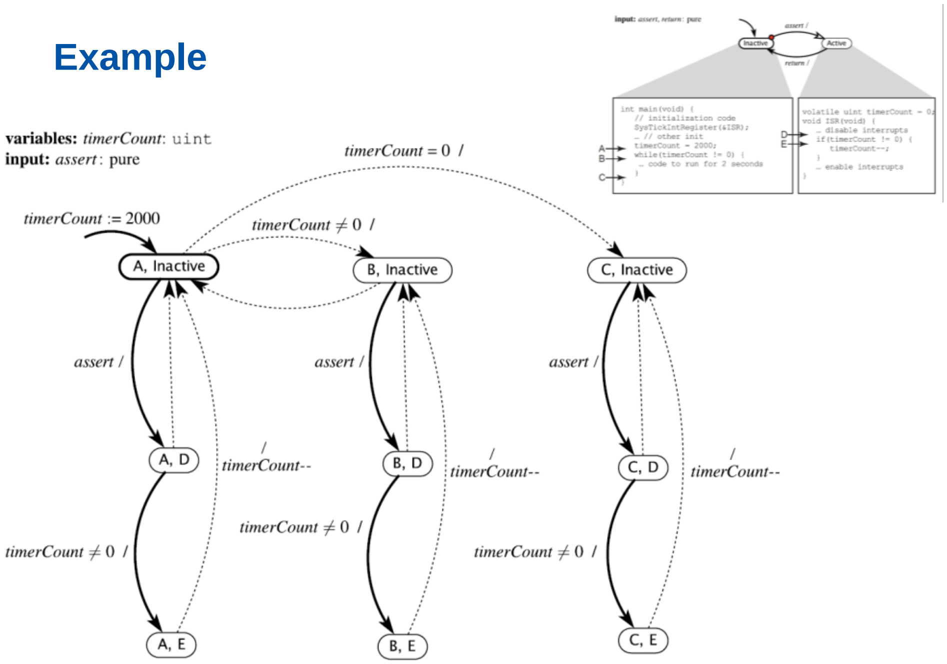
static variable: declared outside main() puts them in statically allocated memory (not on the stack)

volatile: C keyword to tell the compiler that this variable may change at any time, not (entirely) under the control of this program.

Interrupt service routine

Registering the ISR to be invoked on every SysTick interrupt

Example

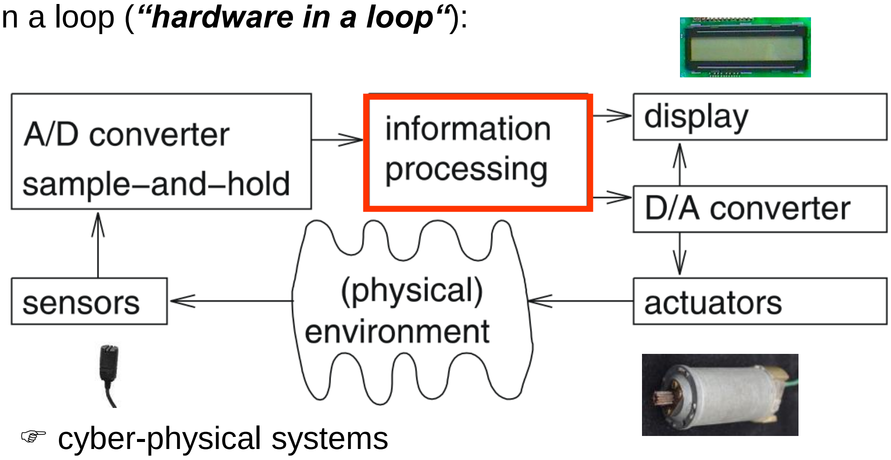


```
int main(void) {
   // initialization code
   SysTickIntRegister(&ISR);
   … // other init
   timerCount = 2000;
   while(timerCount != 0) {
    … code to run for 2 seconds
   }
}
```

```
volatile uint timerCount = 0;
void ISR(void) {
   … disable interrupts
   if(timerCount != 0) {
      timerCount--;
   }
   … enable interrupts
}
```

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

sensors

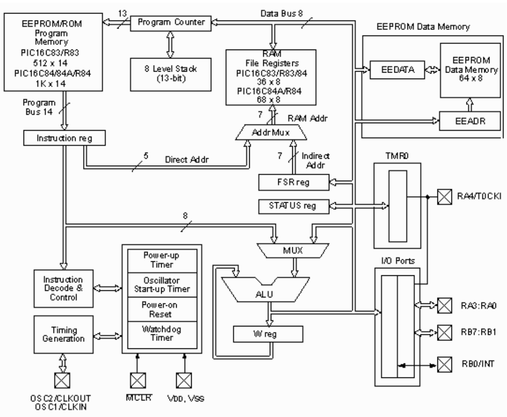
(physical) environment

actuators

☞ cyber-physical systems

Microcontrollers

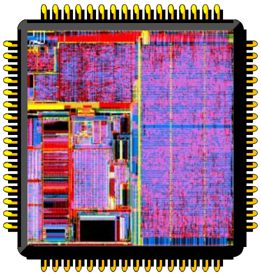
- Integrate several components of a microprocessor system onto one chip
  CPU, Memory, Timer, IO
- Low cost, small packaging
- Easy integration with circuits
- Single-purpose



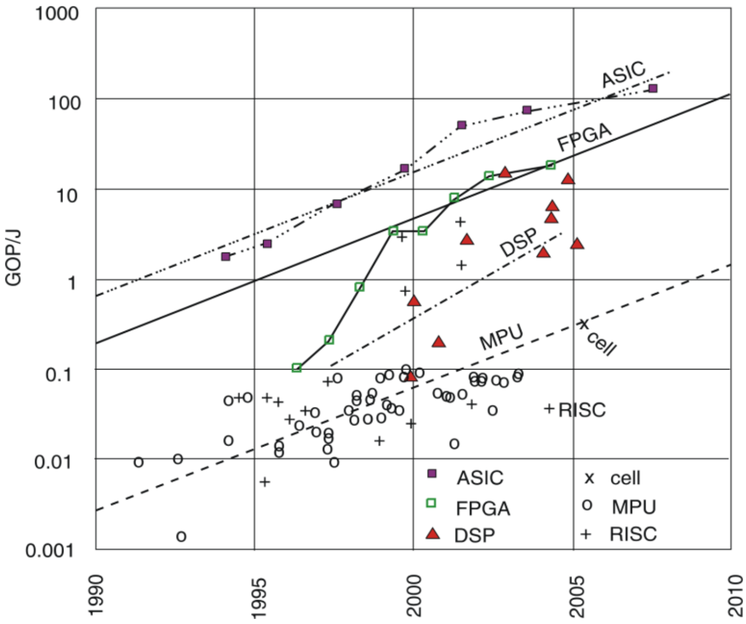
PIC16C8X

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

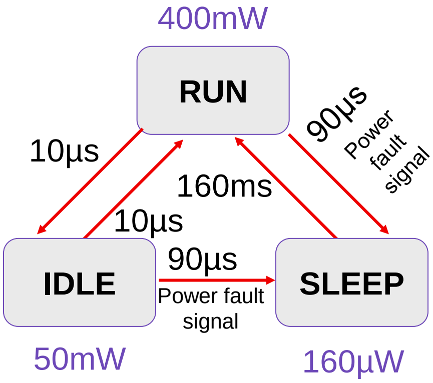
Low Power vs. Low Energy Consumption

- 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

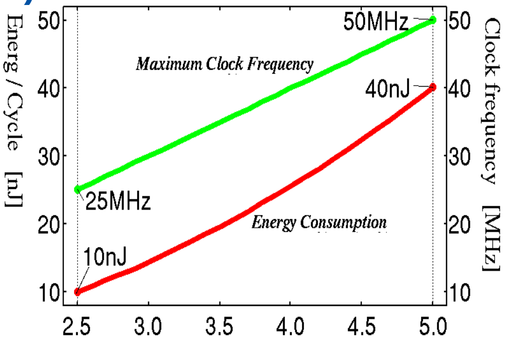
Dynamic power management (DPM)

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



Fundamentals of dynamic voltage scaling (DVS)



[Courtesy, Yasuura, 2000]

Power consumption of CMOS circuits (ignoring leakage):

$$P = \alpha\, C_L V_{dd}^2\, f \quad \text{with}$$

α : switching activity

C_L : load capacitance

V_{dd} : supply voltage

f : clock frequency

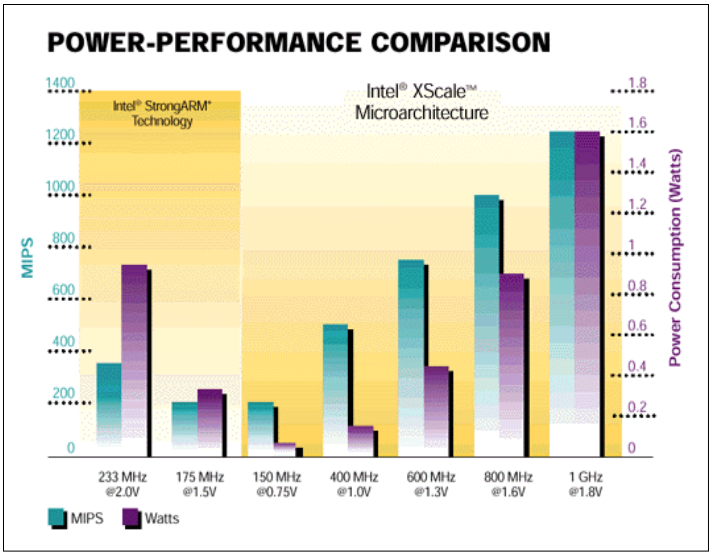
Delay for CMOS circuits:

$$\tau = k\, C_L \frac{V_{dd}}{(V_{dd} - V_t)^2} \quad \text{with}$$

V_t : threshhold voltage

$(V_t < \text{than } V_{dd})$

Variable-voltage/frequency example: INTEL Xscale



From Intel's Web Site

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 α operations per clock:

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

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

Rough approxi-mations!

Application: VLIW processing and vol-tage scaling in the Crusoe processor

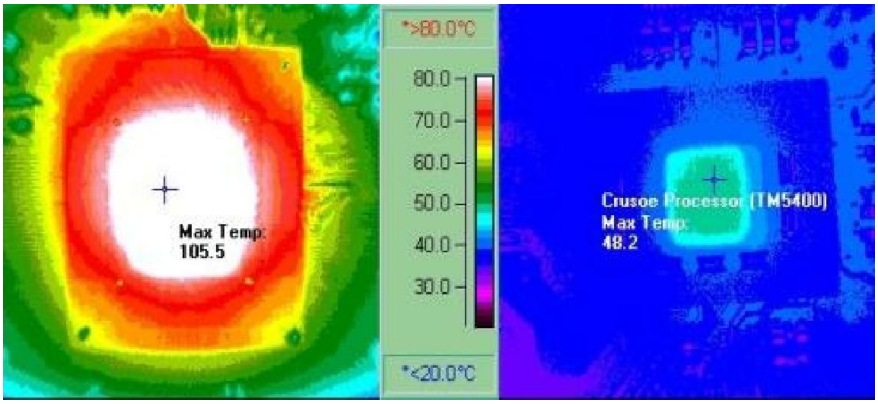
- V_{DD}: 32 levels (1.1V - 1.6V)
- Clock: 200MHz - 700MHz in increments of 33MHz

  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

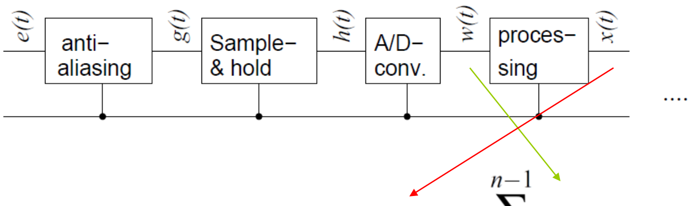


Running the same multimedia application.

[www.transmeta.com]

Digital Signal Processing (DSP)

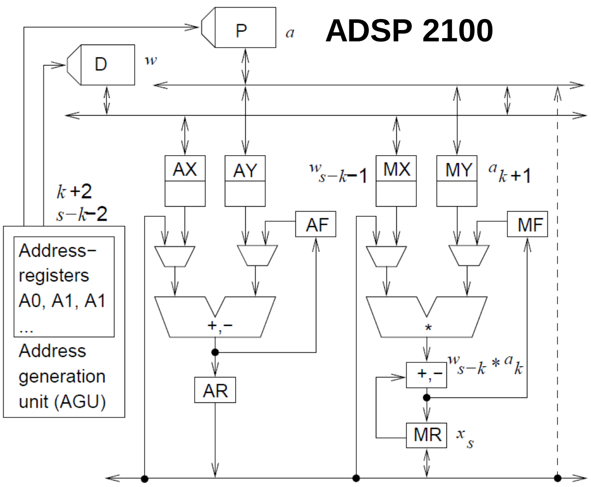
Example: Filtering



$$x_s = \sum_{k=0}^{n-1} w_{s-k} * a_k$$

Signal at $t=t_s$ (sampling points)

Filtering in digital signal processing



$$x_s = \sum_{k=0}^{n-1} w_{s-k} * a_k$$

outer loop over sampling times t_s

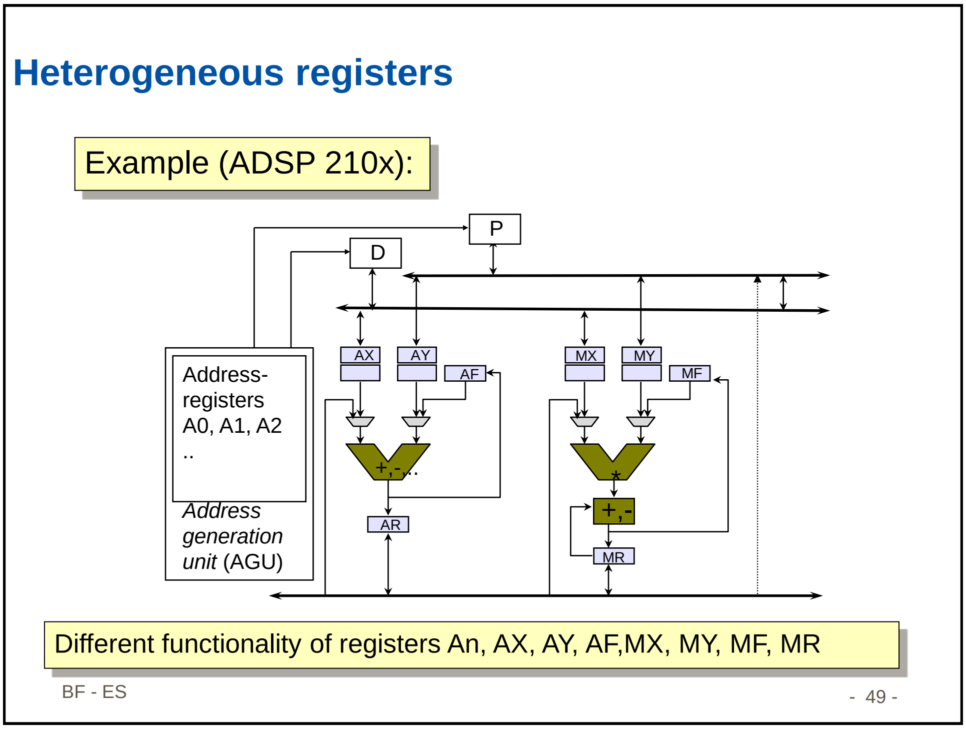
```
{ MR:=0; A1:=1; A2:=s-1;
  MX:=w[s]; MY:=a[0];
  for (k=0; k <= (n−1); k++)
   { MR:=MR + MX * MY;
     MX:=w[A2]; MY:=a[A1];
     A1++; A2−−;
   }
  x[s]:=MR;
}
```

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

```
MR:=0; A1:=1; A2:=n-2; MX:=x[n-1]; MY:=a[0];
for ( j:=1 to n)
  {MR:=MR+MX*MY; MY:=a[A1]; MX:=x[A2]; A1++; A2--}
```

Multiply/accumulate (MAC) instruction

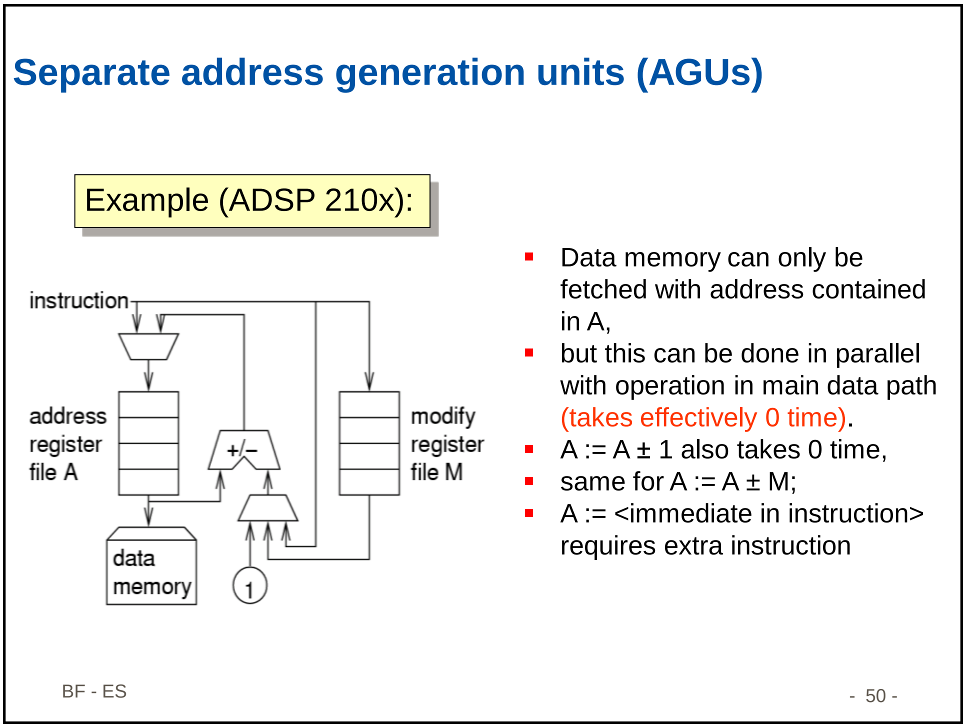
Zero-overhead loop (ZOL) instruction preceding MAC instruction.
Loop testing done in parallel to MAC operations.



Heterogeneous registers

Example (ADSP 210x):

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



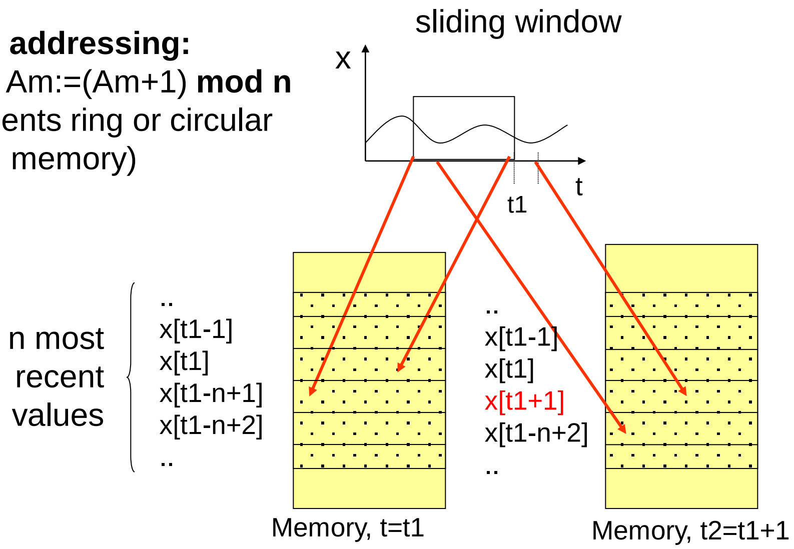
Separate address generation units (AGUs)

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++ ≡ Am:=(Am+1) **mod n**
(implements ring or circular buffer in memory)



sliding window

n most recent values

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

Memory, t=t1

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

Memory, t2=t1+1

Saturating arithmetic

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

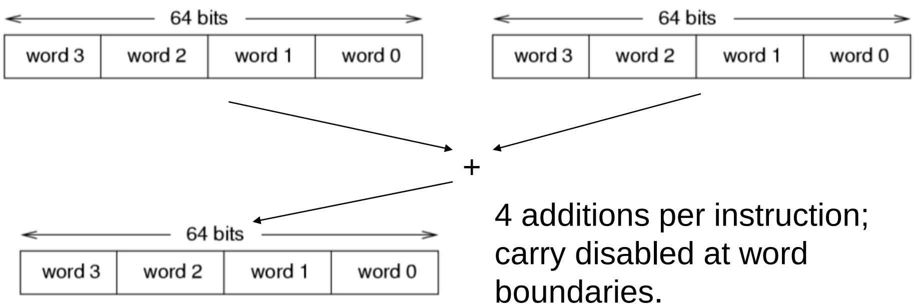
a			0111
b		+	1001
standard wrap around arithmetic			(1)0000
saturating arithmetic			1111
(a+b)/2:	correct		1000
	wrap around arithmetic		0000
	saturating arithmetic + shifted		0111 „almost correct“

- 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)

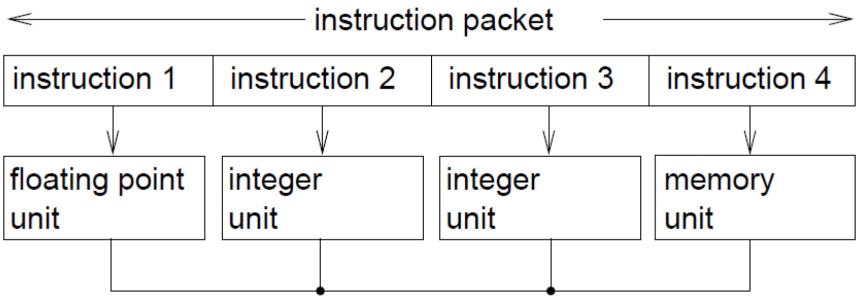
☞ 2-8 values can be stored per register and added. E.g.:



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.



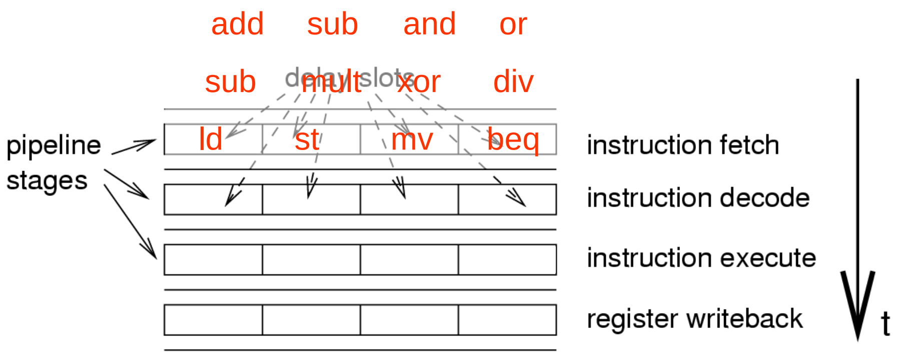
Very long instruction word (VLIW) architectures

- 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



add sub and or

sub mult xor div

delay slots

ld st mv beq

pipeline stages

instruction fetch

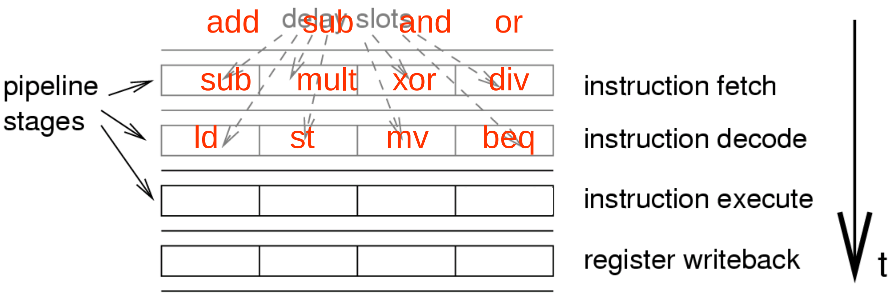
instruction decode

instruction execute

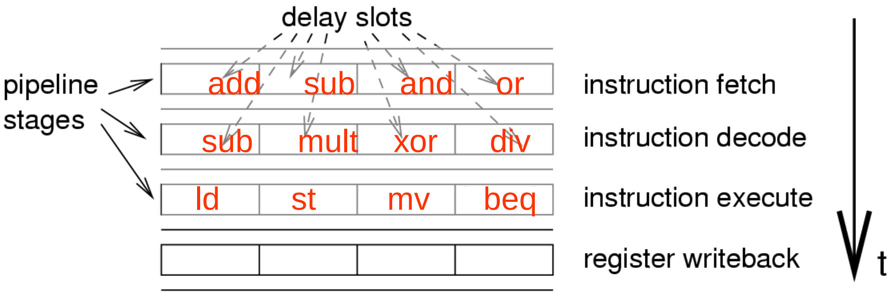
register writeback

t

Large # of delay slots, a problem of 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.