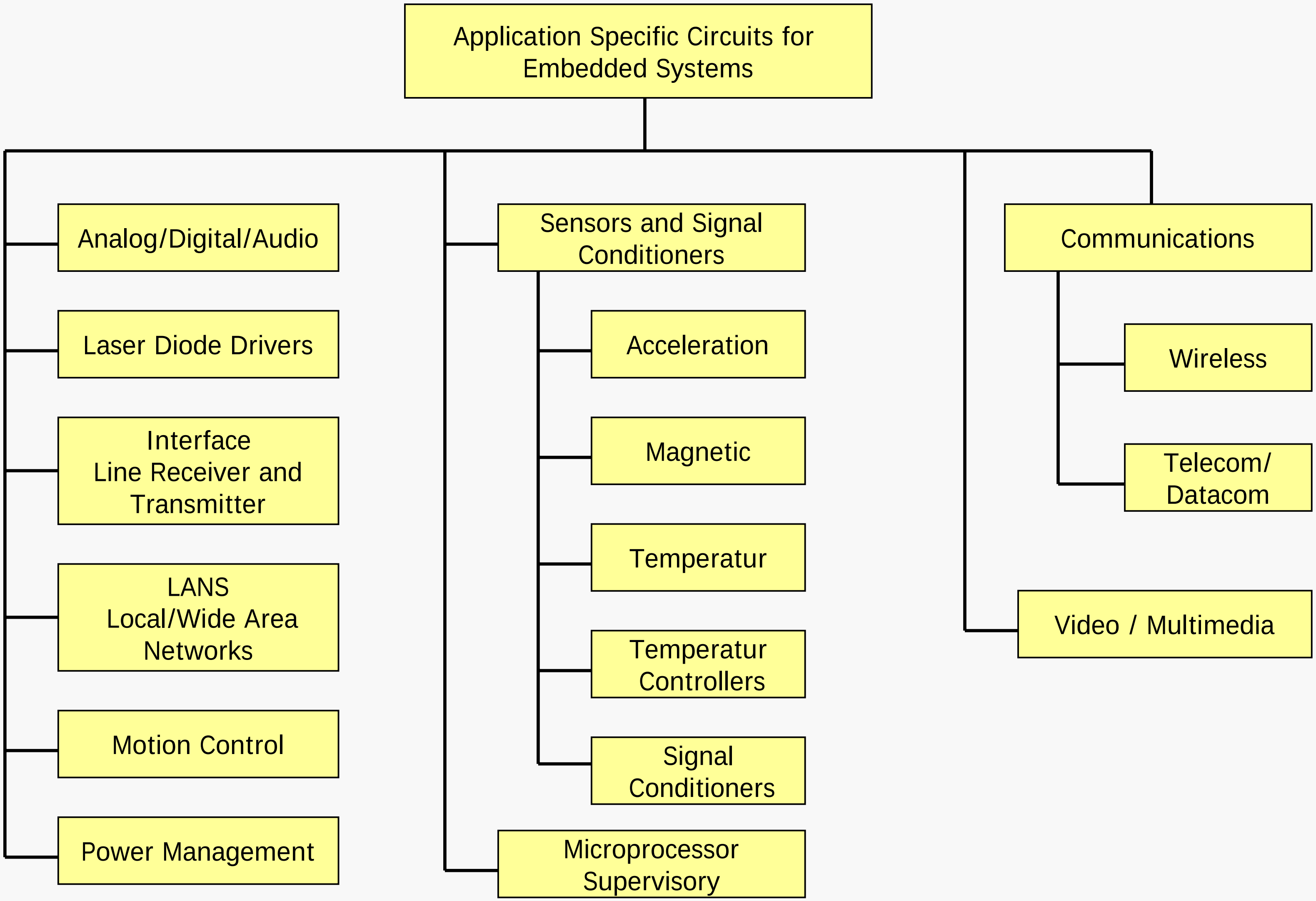

Application Specific Circuits for Embedded Systems
Analog/Digital/Audio
Laser Diode Drivers
Interface Line Receiver and Transmitter
LANS Local/Wide Area Networks
Motion Control
Power Management
Sensors and Signal Conditioners
Acceleration
Magnetic
Temperatur
Temperatur Controllers
Signal Conditioners
Microprocessor Supervisory
Communications
Wireless
Telecom/ Datacom
Video / Multimedia

Sensors	Data Acquisition	Memory	Processor	Output	Actuators
Voltage, Current	A/D	RAM	Micro-processor	D/A	Loud-speaker
Temp-erature	Codec	Dual Port RAM	Micro-controller	Digital	Relay, Led Lamp
Light, Pressure, Noise	Digital (Fifo, Latch Register)	Multi Port RAM	DSP	Display	Switch
Audio, Ultrasound		Shared RAM	Transputer	SSI/SPI	Motor
Position, Velocity Acceleration		Fifo	Gate-Array Core-Processor	CAN, Profibus	....
Human Interfaces		FPGA	DSP Multi-Processor System	Network (Ethernet)	
....				Bus-Interfaces	

A more generall view to different functional blocks used in embedded systems.

Interface techniques for data acquisition and control

for MCU/DSP-based Embedded Sytsems

- MCUs provide high performance in general applications
  - optimised for control flow applications with I/O-signals, complex interrupt interactions and user interfaces

- DSPs provide unbeatable performance in signal-processing applications by high computation power
  - Optimised for high data transfers
  - Single-cycle execution of fundamental processing operations by ALU, MAC and SHIFTER
  - Intensive use of Multi-Instructions (single/multiple execute, fetch and store in one cycle)

  - However lots of secondary processes/tasks have to be executed
    - Data have to be acquired and/or output
    - Signal processing is controlled by additional conditions from the environment and user settings (communication with a master processor, user interfaces, sensors and actuators)

- The benchmark examples of the manufacturers give us the impression, that the optimised code for the used algorithm is the whole work !
  - For example: The code of a FFT algorithm assumes that the data are already available in the internal memory and often all internal busses are occupied by the optimised use of data movements.
  - First the data has to be transfered from a source to the internal/external memory

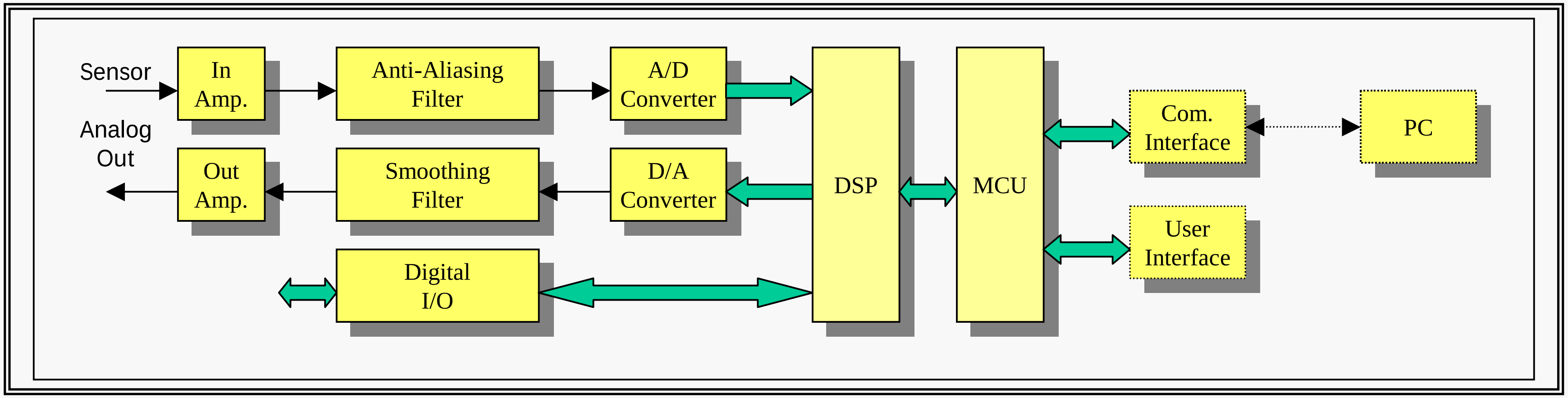
Interface techniques for data acquisition and control

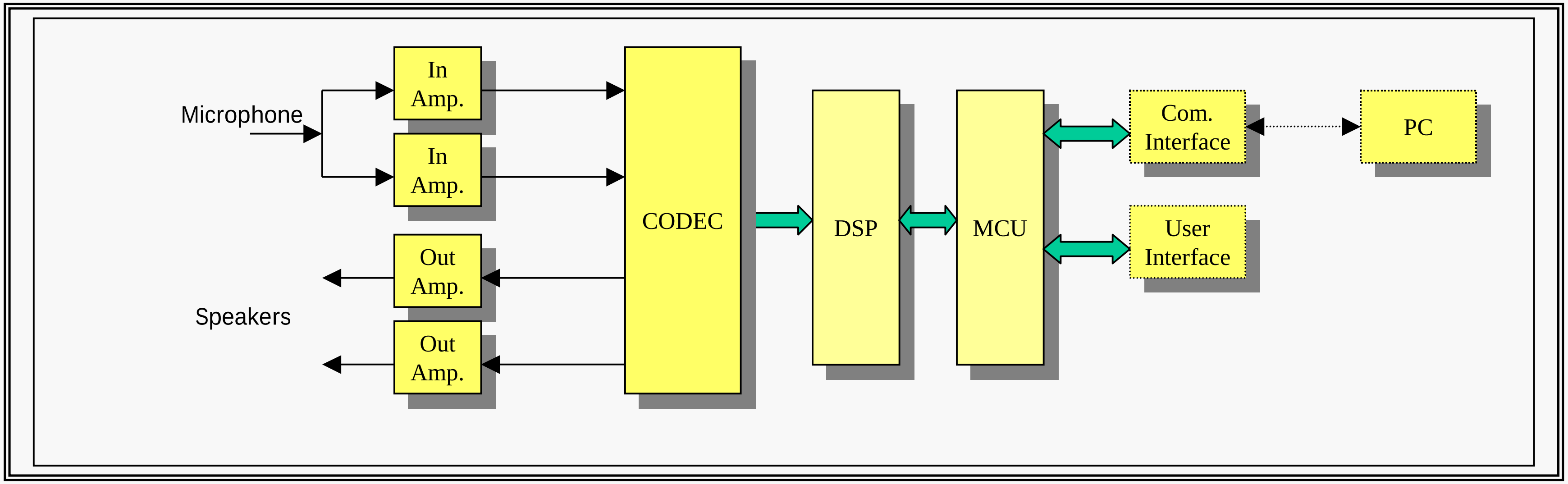
for DSP-based Embedded Sytsems

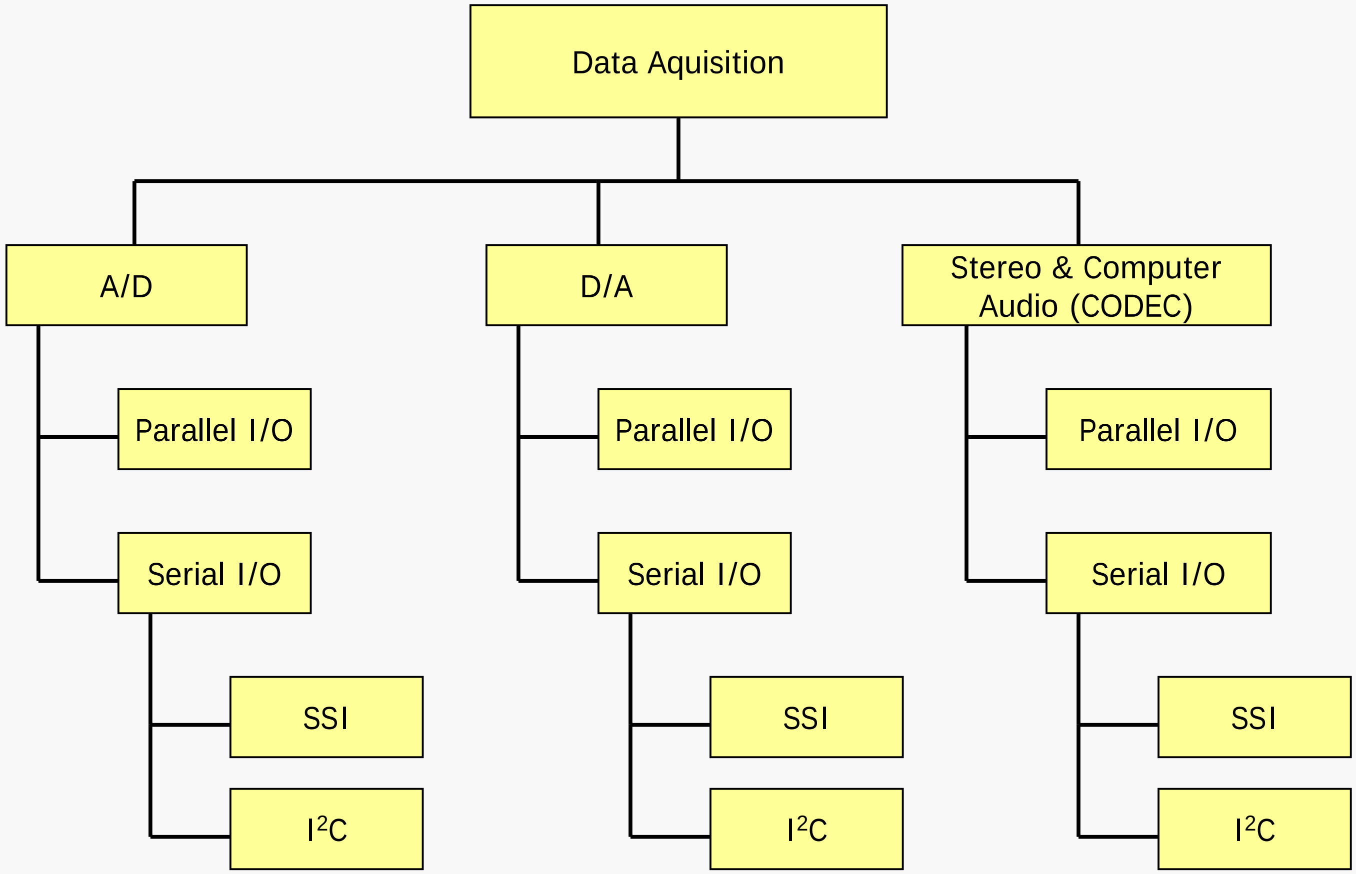
- This is not realistic
  - In parallel with the execution of algorithms, the processor has:
    - To read from A/D-converters
    - To write to D/A-converters
    - store/read data samples in buffers
    - sample I/O-ports
    - communicate (read/write) with host MCUs or communication devices
  - This different tasks often lead to interrupts, bus locks or bus collisions from internal to external memory.

> - Data Aquisition and data output
> - Control Interface
> - DSP and Memory
> - Multiprocessing Techniques
> - Code Development (Compiler/Linker)

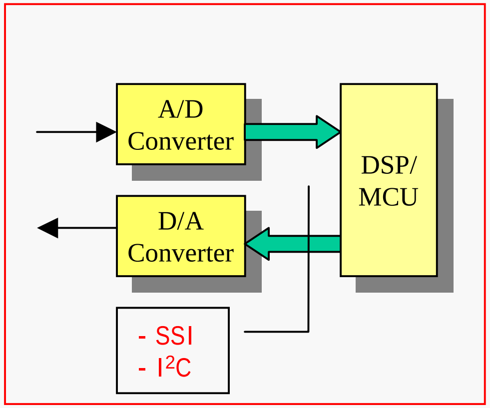
Typical structure of Embedded Systems






Data Aquisition
A/D
D/A
Stereo & Computer Audio (CODEC)
Parallel I/O
Serial I/O
SSI
I2C
Parallel I/O
Serial I/O
SSI
I2C
Parallel I/O
Serial I/O
SSI
I2C

Data Acquisition : Serial Interface



- Most DSP systems processes real-world-data
  - by the aid of A/D and D/A converter
  - digital data transmitted by point-to-point links or networks

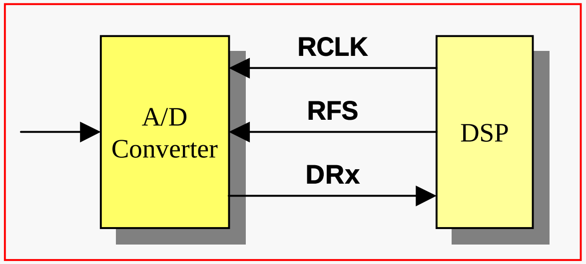
- Aquisition of a data stream by a general synchronous serial interface (SSI or I^{2}C)
  - All general purpose DSP have at least one serial port built in peripheral

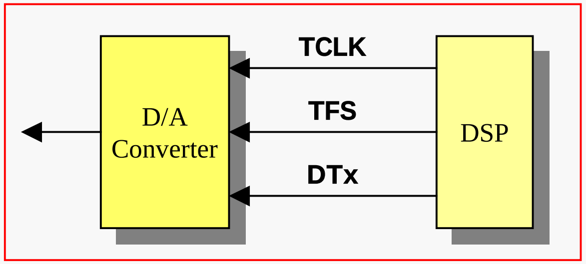
Advantages:

- Industrie standard converters and Codecs can be connected without additional glue logic
- Offer multi-channel-support for time division multiplex interfaces
- Easy to isolate and to decouple in electrically point of view
- Few signals (6) are necessary for bidirectional, independant data transfer
- Independence from external bus operations, SSI is able to work stand-alone
- Different techniques for data aquisition are possible
  - Single write/read transfer
  - DMA weite/read block-buffer transfer
  - Polling mode
  - Interrupt driven transfer

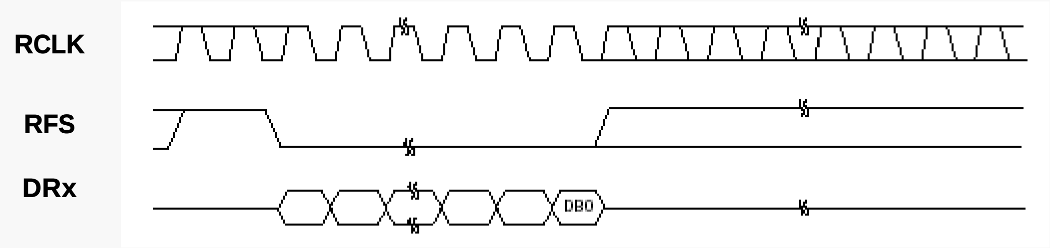
Disadvantage:

- Slow data aquisition performance (typcially up to 200 KHz analog bandwith (max. 1 MHz)





3 Wire Interface

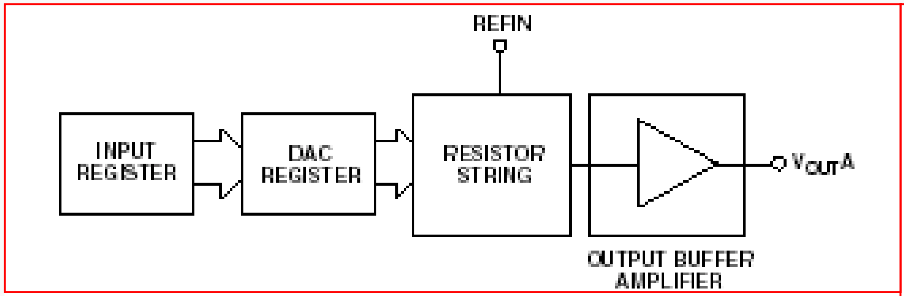


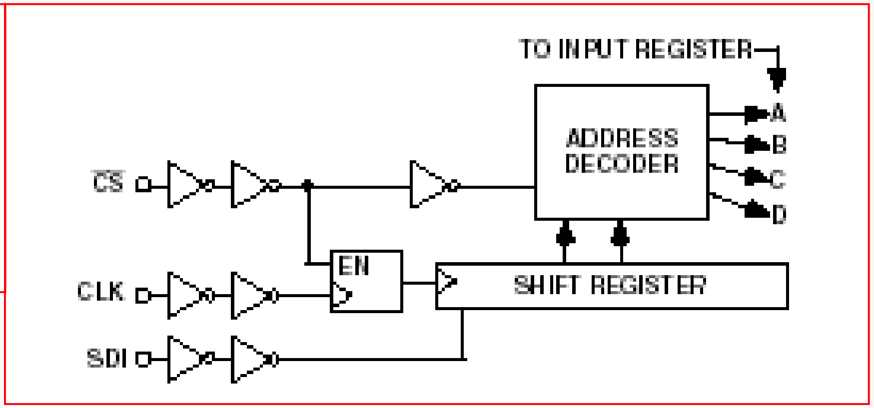
RCLK: Receive Clock
RFS: Receive Frame Signal
DRx: Receive Data

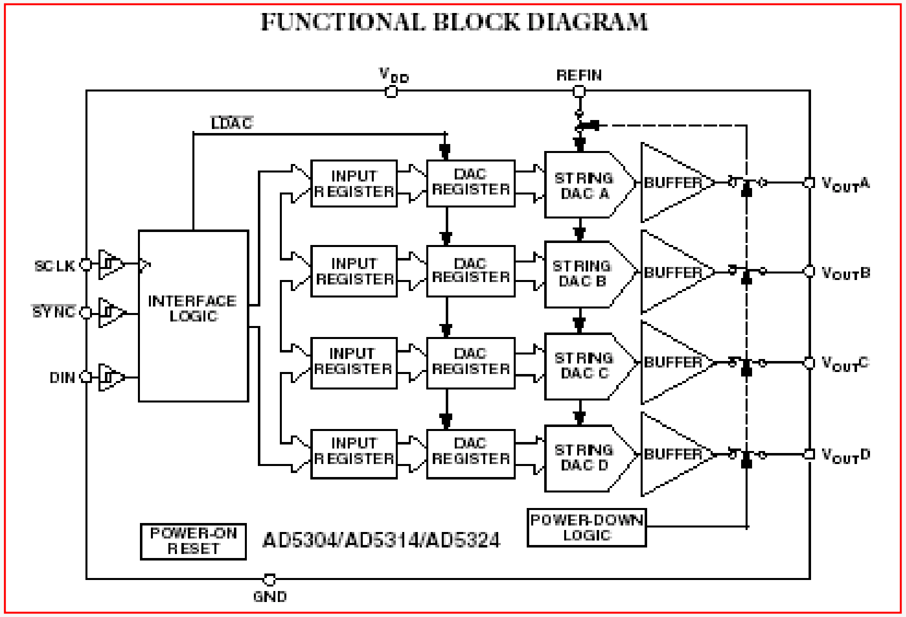
TCLK: Transmit Clock
TFS: TransmitFrame Signal
DTx: Transmit Data

- The mode of the SSI port can be configured by control registers
  - Polarity of Frame signals
  - Frame sync type
  - Clock timing and frame cycle timing (internal or external)
  - Multi-channel support for time division multiplex interface; pre-selected time-slots will be read to reduce transfer time and to releave the Core Processor

- Clocks and frame signals are generated internallly: DSP is master
- Clock and frame signal are generated externally: DSP is slave
- The max. clock frequency is today 30 MHz up to 100 MHz (30 Mbit/sec to 100 Mbit/sec)
- The limitation is not the clock frequency; it is the analog bandwidth and the serializer of the A/D or D/A converter



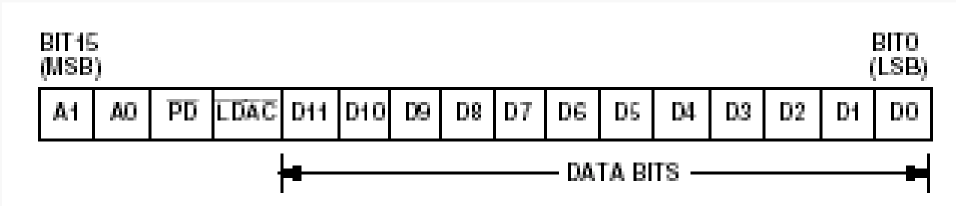




Example: AD5324 / Analog-Devices

- Quad Serial-Input D/A converter
- No additional Glue-Logic
- Four buffered 12-Bit DACs
- DSP-Compatible 3-Wire Serial Interface (SSI)
- Low Power, SPI, QSPI and Microwire Interface
- Additional Update Signal for all DACs

Data Format for a 4 Channel DAC:

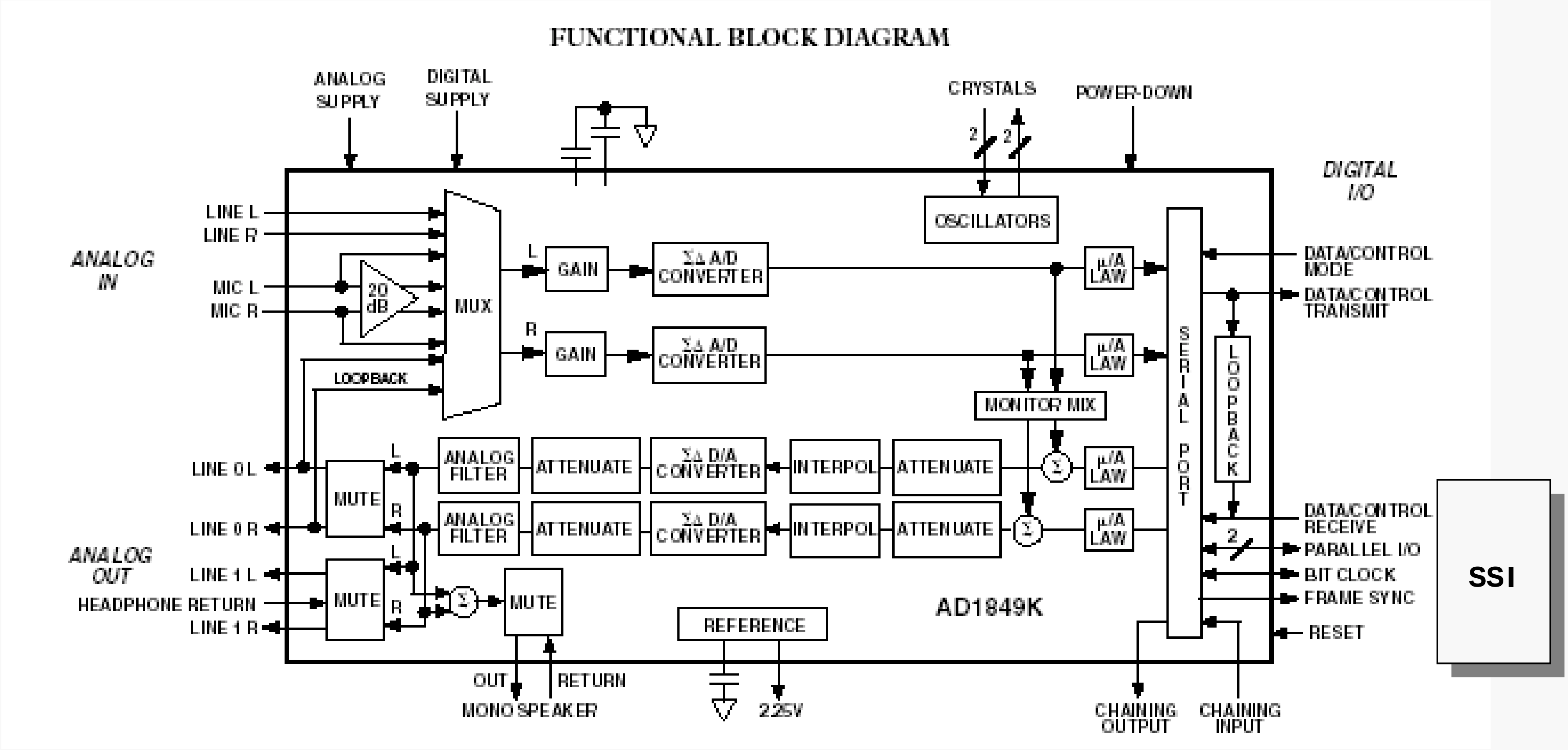


- 12 Bit Data (-2048 to +2047)
- 2 Bit Address (4 DACS addressable)
- 2 Bit Control (PD: Power Down; LDAC: Load Dac Immediate)

- Additional cycles are necessary to encode the data to the DAC
  - Address selection
  - Update Mode
  - Power Down Mode

- Data tansfer from the converter to the memory is possible by
  - Write/read by Core-Processor
    - Polling mode
    - Interrupt mode
    - Not preferable, because the Core-Processor is locked by serial read/write for other tasks
  - DMA mode (Direct Memory Address)
    - Polling mode
    - Interrupt mode
    - Usefull for block buffer transfer
    - Core-Processor is relieved for other tasks

Special serial device for audio applications: Sound-Port-Codec



- Example AD1849 / Analog Devices

General Description

- The Serial-Port 16-Bit Stereo Codec integrates the audio data conversion and control functions in one single integrated device.
- A Codec is a single-chip audio solution for multimedia applications.
- Externel signal components and requirements are limited to three capacitors for line level application.
- So called Anti-Imaging filters are incorporated on-chip.
- The chip includes on-chip monoaural speaker and stereo headphone drive circuits.
- The dynamic range exceeds 80 dB across the audio bandwidth of 20 KHz.
- Sample rates up to 48 KHz are supported from external crystals, an external clock or from the serial clock of the SSI port.

- The Codec works with a pair (Stereo) of Sigma-Delta analog-to-digital converters and a pair of of Sigma-Delta digital-to-analog converters.
- Analog signal are fed to the chip at line levels or microphone levels.
- A programmable gain stage allows the setting of independant gains into the ADC
- The ADCs digital output data can be mixed with the DACs input.
- The channels (with 16-bit data) are available over a single bidirectional serial interface that also manages 16-bit digital input to the DAC and additional control information.
- Additional feature: compress and decompress technique.
- Integration of a digital interpolation filter and a attenuator control circuit for volume adaption.
- Nyquist images and quantization noise are removed from the DACs analog stereo output by an on-chip switched-capacitor and continues-time filters.

Time and Frequency Considerations of SSI based Converters

Serial Interface		
Cycle time [us]	Transfer rate [Mbit/sec]	Transfer time 16 Bit [us]
1	1	16
0.1	10	1,6
0.05	20	0,8
0.03	33	0,48

Processor	
Cycle time [ns]	Count of Instructions
0.025	640
0.025	64
0.025	32
0.025	19

Analog Signal
Frequency
< 32 kHz
< 312 kHz
< 625 kHz
< 1 MHz

Result:

- Use the highest possible data rate on SSI interface
- Use DMA transfer to releave the processor
- Use interrupt technology for event triggering

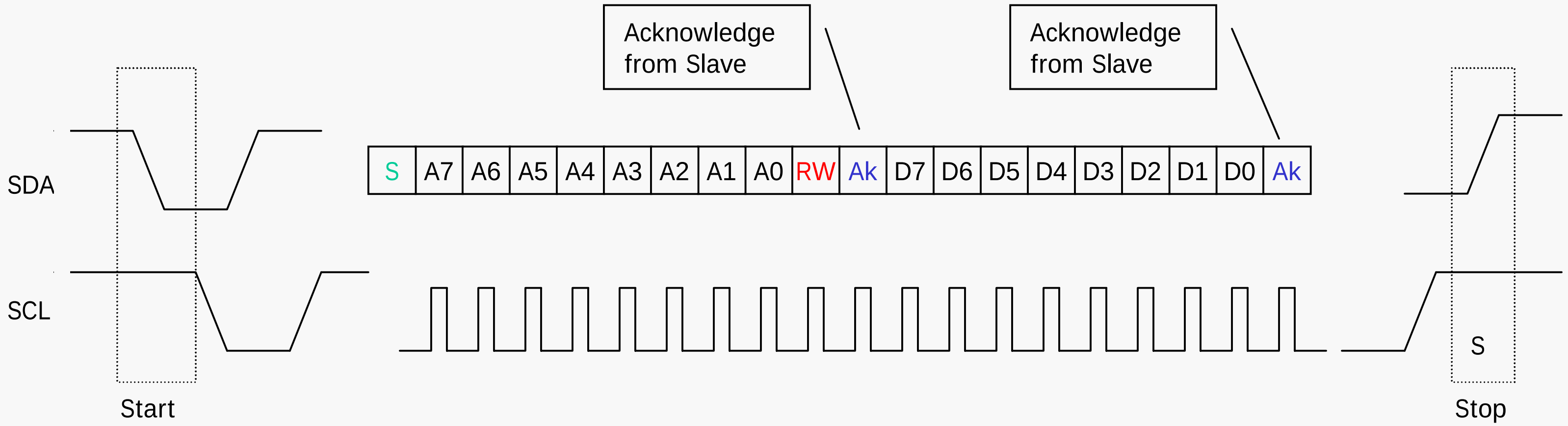
•Typicall situation for audio processing

- 48 kHz sample rate
- 20.8 us / sample
- 20.8 us = interrupt
- 40 MHz DSP <-> at least 832 single instructions

I^2C Interface

- I^2C is a two wire interface with a special protocoll
- Used for slow A/D-, D/A converters and digital input/output devices (100 kHz)
- Uses a serial data I/O line (SDA) and a serial clock (SCL).
- A select line (as Frame) is not required
- Data transfer may be initiated only when the bus is not busy (multiple devices are possible)
  - Each device has a unique address on the bus
- During such transfer, the data line (SDA) must remian stable whenever the clock line (SCL) is high
- Changes in the data line while the clock line is high are interpreted as a START or STOP condition

- A typical transfer format:



I^2C Interface

- A communication is set up by a master sending a start condition, a high-to-low transition on SDA while the serial clock (SCL) input is high
- After the start condition, the device address byte is send; MSB first, including the data direction bit (R/W)
- After receiving the valid address byte, the slave device responds with an acknowledge, a low on SDA during the high of the acknowledge-related clock pulse.
- The data bytes follows the address acknowledge. If the R/W bit is high a read cycle is performed.
- The data byte is followed by an acknowledge sentd from the device.
- Other data bytes are possible always followed by an acknowledge send from the device
- A stop condition, a low-to-high transition on the SDA input/output while the SCL is high, is send by the master after transfer

Data Acquisition : Parallel Interface

- High-speed data acquisition usual requires a parallel interface because of the speed limitation of a serialized process.

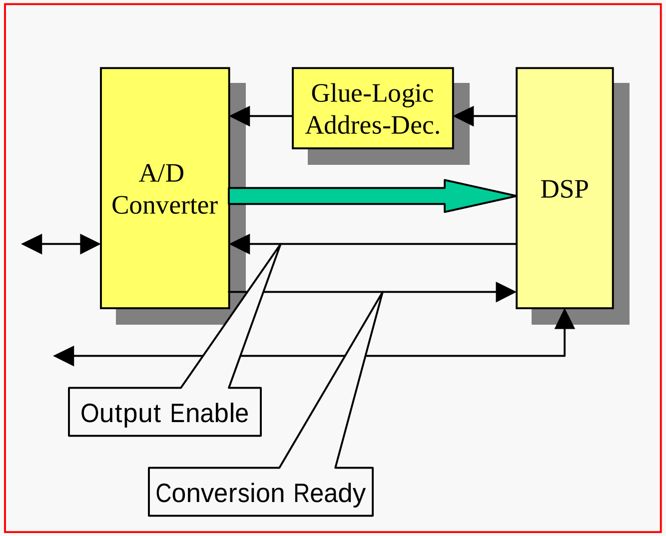
- Aquisition of a data stream by a parallel interface
  - All general purpose DSP have at least one parallel port, the external high-speed memory interface
  - Different techniques for data aquisition are possible
    - Single write/read transfer
    - DMA write/read block-buffer transfer
    - Polling mode
    - Interrupt driven transfer
    - Transfer via additional components (Fifo, Dual-Port-Ram, ...)

Advantages:

- High speed data aquisition

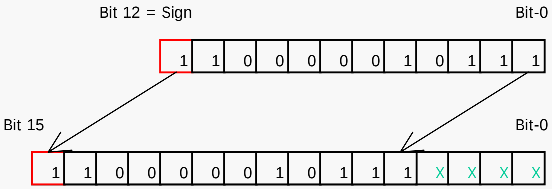
Disadvantages:

- Additional Glue-Logic
- Performance is dependant from external bus operations as memory accesses or othe DMA processes
- Alignment of the data bits of the converter to the bits of the processor format is necessary.



Example: 12-Bit ADC; parallel interface; write twos-complement format to a 16-bit DSP

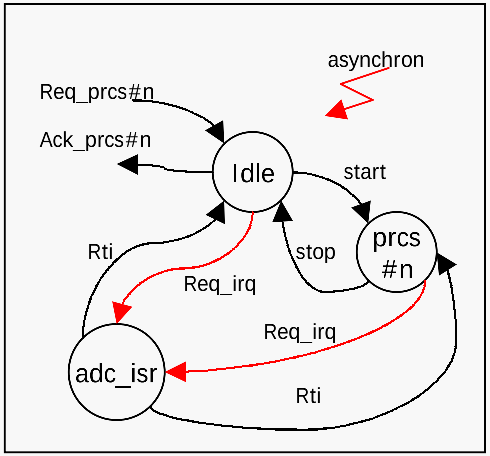
+Vmax: 0100.0000.0000.0000
0 0000.0000.0000.0000
-1 1111.1111.1111.1111
-Vmax: 1000.0000.0000.0000



by additional Hardware or arithmetic shift !

Data rate by the different acquisition modes and transfer techniques

- An ADC (parallel/serial) and an interrupt service routine (ISR) is used for data convert/transfer to an internal buffer; no additional buffering (Fifo,...) for the parallel mode is used



In C:

```
-------------------------------------------
void adc_isr()
{
        buff[index++] = *(int *) adc
}
-------------------------------------------
```

- If interrupt
  - code in the pipeline will be flushed
  - put PC on the stack
  - load PC with the address from IRQ-vector-table
  - save actual context (from prcs#n or idle)
  - do ISR code (load data from ADC to buffer)
  - restore context
  - restore PC address from stack
  - fill execution pipeline
  - continue processing of prcs#n or idle by return from interrupt

Result:

- We need a lot of additional cycles for reading one data from the ADC
- Only continuous acquisition mode with low speed is possible

Problems

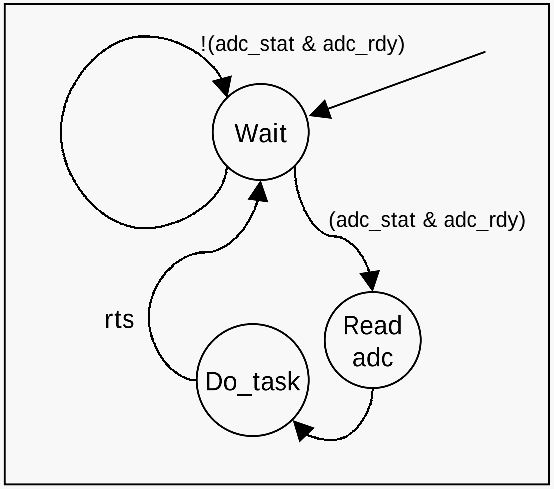
- Cycle time is not the only criterion; length of pipeline is important
- if external memory is accessed the time will increase (additional fetch cycles; in general only one external hardware bus interface is avaliable, speed of external memory !
- C-Compilers will not always track the register usage inside the ISR and therefore will save and/or restore a partial set of registers, which wasts additional cycles)
- The interrupt implementation varies in C; (SHARC: needs 26 cycles (cycle-time: 25 ns) in the fastest mode (= 650 ns !)
  - Solution: fast context switch to the second register/Index set and hand coded ISR entrance !
- It is not preferable to call a function inside the ISR
- If possible, the ISR should only set a flag; a dispatcher should manage the following actions by a normal function call
- Some instructions are not interruptable
- Interrupt nesting has to be used very carefully

Conclusion:

- Interrupt driven data transfer is not preferable above one or two MHz (depends on the processor)
- Use interrupt with the highest priority and enable the nesting mode

Polling Mode

- An ADC (parallel/serial) and polling mode is used for data convert/transfer to an internal buffer; no additional buffering (Fifo,...) for the parallel mode is used
- The ADC ready signal is sampled by the processor-core



Result:
- We need fewer cycles as for the interrupt ISR
- If the DSP has BIT-Test fuctions or is able to test directly an I/O-signal, the process is much faster, but generally it demands hand coded inline assembly statements !

In C:

```
------------------------------------------------------------
void main()
{
        while(1) {
                ....
                while(!(*(int *) adc_stat & adc_rdy));
                        buff[index++] = *(int *) adc;
                        do_task();
                ....
         }
}
------------------------------------------------------------
```

- If interrupt
  - Loop1:
    - load address register with adc_stat to register
    - load constant adc_rdy to register
    Loop2:
      - load adc_stat to register
      - perform ‚&'
      - if result = 0 jump Loop2
    - load address register with adc
    - load address register buffer
    - load register with index
    - load data from adc to register
    - store register to buffer
    - increment index
    - perform the function ‚do_task'
    - jump Loop1

Block-wise Polling Mode

- Polling mode has more efficiency if the dataacquisition is performed block-wise. In this transfer mode, the indices and addresses remain in registers. No re-load is required for each loop.
- In this case the data acquisition is non-continuous; no data are loaded during a signal processing task.

In C:

```
----------------------------------------------------------------------
void main()
{
        while(1) {
              ....
              for(i=0; i<imax; i++) {
                        while(!(*(int *) adc_stat & adc_rdy));
                        buff[index++] = *(int *) adc;
              }
              do_task();
              ....
         }
}
----------------------------------------------------------------------
```

Best result, because we stay inside the loop, no re-load of register and addresses.

Conclusion

- Polling is only useful for block-wise and non-continuous data acquisition at high sample rate.
- For high speed applications interrupts or bus locking processes have to be prevent during polling mode.

DMA Transfer

- Nearly all MCUs and DSPs have efficient DMA controller capabilities without CORE access in order to move:
  - data to and from memory
  - data to and from built in peripheral

- The DMA process is event driven by an external DMA-request signal or by an internal start condition via watching status-register contents (serial transmitter/receiver buffer empty, full, ...).

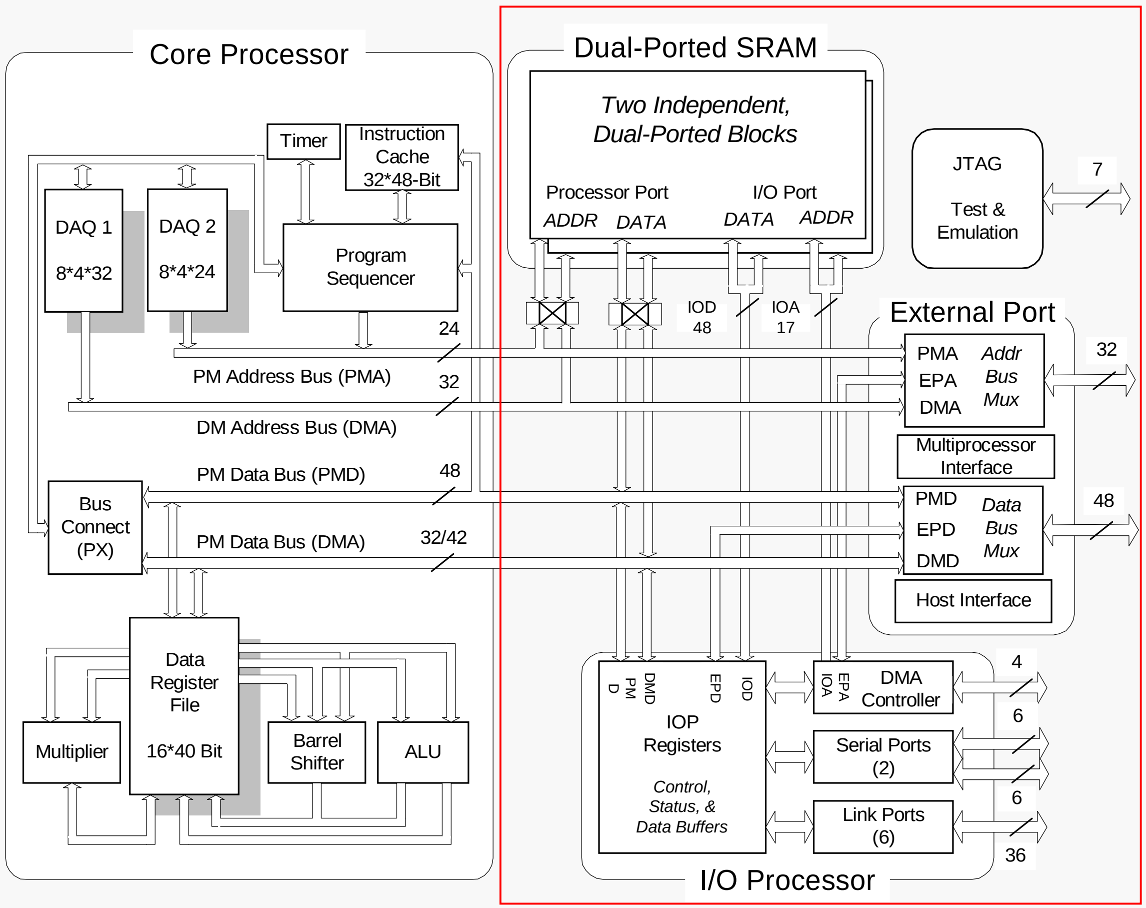
- DMA controller increments automatically addresses or manages circular buffering.
-
- Because the processor-core is not loaded with the action, the acquisition speed depends only from the maximum serial bit rate or the parallel transfer rate via the external bus.

- The process has to be synchronised to the data stream by watching DMA status bits or DMA complete interrupts.

- Best result are given by block-mode transfer (input to buffer, buffer to output, internal buffer transfer)

- DMA data transfer may be continuous too (depends from the application)

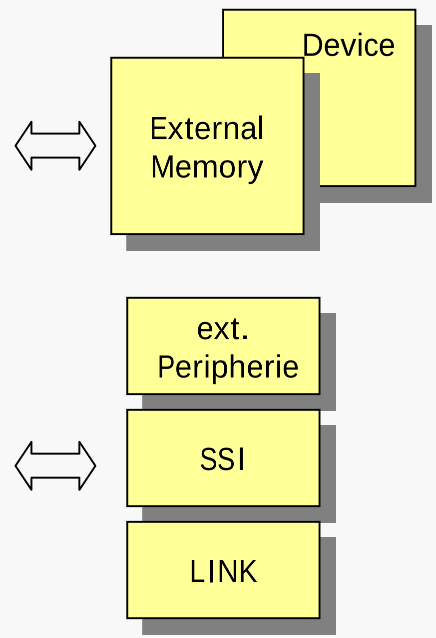
Example SHARC DSP



DMA process:

- internal memory <-> ext. memory and memory mapped devices
- internal memory <-> internal memory of other DSPs
- internal memory <-> host processor
- internal memory <-> serial port I/O
- internal memeory <-> link port I/O
- external memory <-> external peripherie

- DMA runs with cycle steeling from the core



Examples: DMA Transfer

In C:

```
-------------------------------------------------------------------------------

#define    DMA_src          0xe1
#define    DMA_dst          0xe2
#define    DMA_src_inc      0x3e
#define    DMA_dst_inc      0xe4
#define    DMA_count        0xe5
#define    DMA_start        0x3252

volatile int src_buf, dst_buf, src_inc, dst_inc, cnt;

void main()
{
    while(1) {
        ....
        DMA_src = src_buf[0];        /* initialize DMA */
        DMA_dst = dst_buf[0];        /* .. */
        DMA_src_inc = 1;             /* .. */
        DMA_dst_inc = 1;             /* .. */
        DMA_count = 1024;            /* .. */
        DMA_start = START;           /* start DMA */

        while(!dma_ready);           /* polling mode:  wait on DMA
                                        ready */
        do_signal_process();
    }
        IDLE;
}
-------------------------------------------------------------------------------
```

Result:

- The speed depends on the data rate of the used transfer ports (serial, parallel).
- A DSP with 25 ns bus cycle is able to transfer 40 data values per microsecond (40 Mbyte/sec) if internal memory is used and there are no collisions with other DMA processes.
- The process has to be carefully tested for bus idle cycles and possible bus conflicts, in order to determine the achievable maximum speed.

Whereever applicable use a DMA process, because DMA save core-processing time.

Speed-up of data acquisition

- If the acquisition speed is above 20 MHz, the DSP needs additional hardware assistance to perform the required specification.

- Different techniques exists:

If the data-aquisition has a finite count of samples:

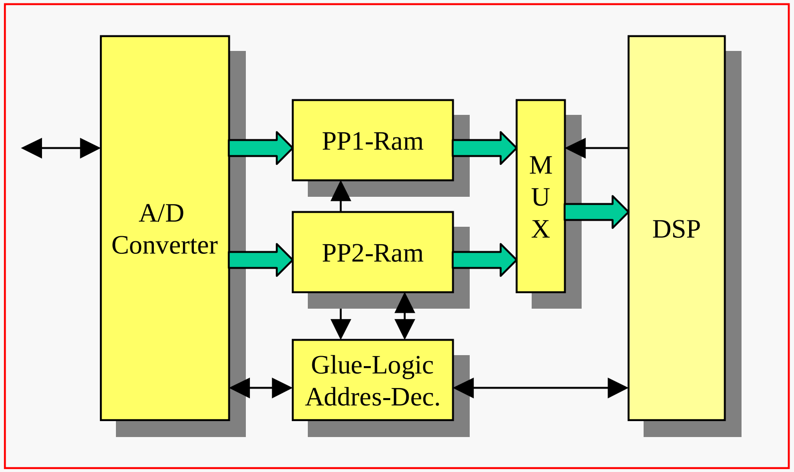
- Integration of a fifo structure for block-read operations.
- Bank switching techniques (Ping-Pong, ...).
- Fit the bit-size of external devices carefully to the bit-size of the processor to save processing time; connect the external bits to the upper data bits. Additional masking and shifter operations depends on the selected processor.

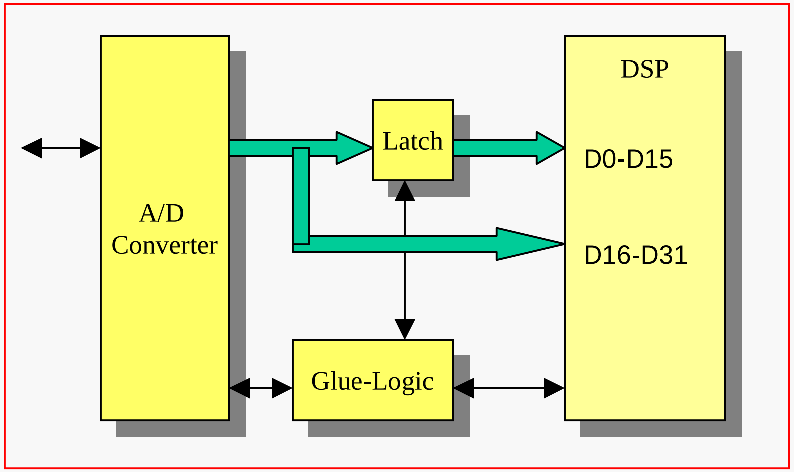
The data-aquisition process is always active and/or faster than the clock cycle:

- Data preprocessing by high speed Gate-Array based devices, e.g. to reduce the sampling frequency (sampling rate) before the data are passed to the processor for further processing steps.
- Use Multiprocessing Systems, More in the Lecture about Signal Processing Techniques

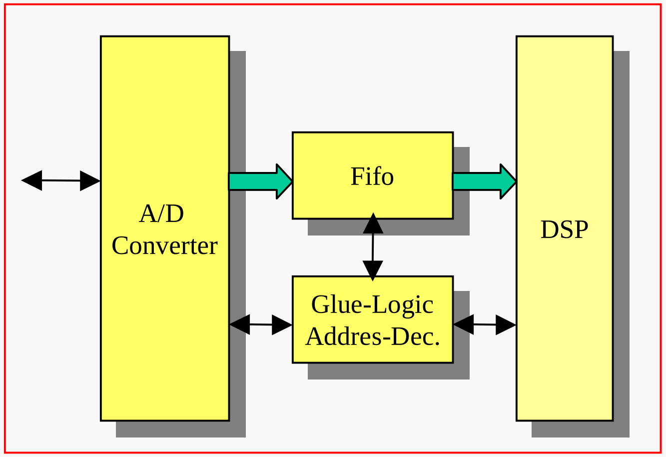
Speed-up of data acquisition by additional buffering

Ping-Pong Memory (PP)





Fifo memory



SHARC-code examples in Assembler for transfer of data via serial interface

- **Assembler for SHARC : Single Word without Interrupt (by Core-Processor)**

```
#include <asm_sprt.h>;
#define N 8
.SEGMENT /pm    seg_pmco;
.global        _SerTransfer;
.extern        _rbuf;
_SerTransfer:
   entry;
            puts=r0;                    /* save data register & address register */
   r0=i0;   puts=r0;
   r0=b0;   puts=r0;
   r0=l0;   puts=r0;

   r0 = 0x00130007;                     /* definitions for serial transfer */
   dm(TDIV0)=r0;                        /* load to ser CLOCK register */
   r0 = 0x000064f1;                     /* definitions for serial transfer division */
   dm(STCTL0)=r0;                       /* load to ser CONTROL register */
   b0=rbuf;                             /* pointer to rbuf[0] */
   l0=@rbuf;                            /* length of rbuf */
   lcntr=N, do tx_loop until ce;        /* if TXO is nit empty, Core-processor halts !! */
         r0 = dm(i0,1);
tx_loop: dm(tx0)=r0;
   r0=gets(1);  l0=r0;                  /* restore data register & address register */
   r0=gets(2);  b0=r0;
   r0=gets(3);  i0=r0;
   r0=gets(4);
   alter(4);
   exit;
.endseg;
```

- **Assembler for SHARC : Single Word with Interrupt**

```
#include <asm_sprt.h>;
#define N 8
.SEGMENT /pm    seg_pmco;
.global        _SerTransfer;
.extern        _rbuf;
_SerTransfer:
   entry;

   /* save the used register on stack !! */

   r0 = 0x00130007;                  /* definitions for serial transfer */
   dm(TDIV0)=r0;                     /* load to ser CLOCK register */
   r0 = 0x000064f1;                  /* definitions for serial transfer division */
   dm(STCTL0)=r0;                    /* load to ser CONTROL register */
   b0=rbuf;                          /* pointer to rbuf[0] */
   l0=@rbuf;                         /* length of rbuf */
   bit set mask imask SPT01;         /* enable sport0 interrupt */
   bit set mode1 IRPTEN;             /* global IRQ enable */
   r0=dm(i0,1);
   dm(TX0)=r0;                       /* kick off sport0 */

  /*  restore the register from stack */
exit;
Serial Interrupt works as a background task-------------------
s0tx: rti(db);
      r0=dm(i0,1);
      dm(TX0)=r0;
.endseg
```

- **Assembler for SHARC : DMA Transfer with Interrupt**

```
#include <asm_sprt.h>;
#define N 64
external .......
_SerDMATransfer:
   entry;
   save the used register !!
   r0=sbuf;          dm(II3)=r0;        /* start pointer of source buffer */
   r0=dbuf;          dm(II1)=r0;        /* start pointer of destination buffer */
   r0=1;             dm(IM3)=r0;        /* DMA increment */
                     dm(IM1)=r0;        /* DMA increment */
   r0=@sbuf;         dm(C3)=r0;         /* lenght of source buffer */
                     dm(C1)=r0;         /* = length of destination buffer */
   r0=0x004421f1;    dm(SRCTL1)=r0;     /* set serial control register 0 */
   r0=0x00130007;    dm(TDIV1)=r0;      /* set clock register */
   r0=0x000465f1);   dm(STCTL1)=r0;     /* set serial control register 1 */

  bit set imask SPRLI;                  /* enable sport1 receive interrrupt */
  bit set mode1 IRPTEN;                 /* global IRQ enable */

  restore the used register !!
exit;
--------------------------------------------
s1rx:   rti;                            /* this interrupt will occur only once */
--------------------------------------------
.endseg;
```

- Assembler for SHARC : DMA

- This example shows the use of on-chip DMA controller to handle serial data. The controller realizes data transfers between internal memory and the SPORTs, providing the most efficient way of block-mode-transfer. When the DMA is set up, the DMA controller operates independently from the SHARC processor-core. The interrupt is only active for one time when an entire data-block has been received (or transmitted). This frees the core to continue other tasks.

- This example sets up a SPORT DMA transfer and receive for serial SPORT1 in the loopback mode, The buffer sbuf is transfered by DMA out of the SPORT. The loopback mode internally attaches the different control signals. The receive DMA places the data in the buffer dbuf.

SHARC-code examples in C and Assembler for transfer of data from a fifo to internal buffer

- AINSI C for SHARC:
- Copy a buffer from Fifo to memory

```
TsCnt = 1024;

void CCpyFifoToMem()
{
   for (i=0; i<TsCnt; i++) rbuf[i]= fifo_adr >> 16;
}
```

11 cycles

```
_L$247:
        r4=dm(_i);
        r4=r8+r4,  r2=dm(i4,m5);
        r2=ashift r2 by -16;
        i0=r4;
        dm(i0,m5)=r2;
        r2=dm(_i);
        r2=r2+1;
        dm(_i)=r2;
        r12=dm(_TsCnt);
        comp(r2,r12);
        if lt jump (pc,_L$247);
!gs     ......
```

(1024 * 11) + 20 = 11284

The 16 bit of the Fio are connected to D31-D16:

- needs a arithmetic shift for 2er complement

- Assembler for SHARC without multfunction
- instructions
- Copy a buffer from Fifo to memory

```
#include <asm_sprt.h>;
.SEGMENT /pm     seg_pmco;
.global         _ASMCpyExtFifoToMem;
.extern         _rbuf;
.extern         _TsCnt;
_ASMCpyExtFifoToMem:
    entry;
            puts=r0;                    /* save data register & address register */
            puts=r2;
    r0=i1;  puts=r0;
    r0=i4;  puts=r0;
    i4=0x440004;                        /* fifo base address */
    i1=_rbuf;                           /* destination buffer */
    r0=dm(_TsCnt);

    lcntr=r0, DO loop1 UNTIL LCE;       /*  (N * 3)*cycle time */
        r2=dm(i4,m5);                   /* load from fifo, m5 = 0 */
        r2=ashift r2 by 0xfffffff0;     /* alignment of 16 bit to 32 bit bus */
loop1:  dm(i1,m6)=r2;                   /* store ti internal buffer */

    r0=gets(1);  i4=r0;              /* restore data register & address register */
    r0=gets(2);  i1=r0;
    r2=gets(3);
    r0=gets(4);
    alter(4);                               (1024 * 3) + 17 = 3089
    exit;
.endseg;
```

- Assembler with multifunction instructions
- Copy a buffer from Fifo to memory

```
#include <asm_sprt.h>;
.SEGMENT /pm     seg_pmco;
.global         _ASMFast;
.extern         _rbuf;
.extern         _TsCnt;
_ASMFast:

    entry;

            puts=r0;
            puts=r2;
            puts=r4;
    r0=i1;  puts=r0;
    r0=i4;  puts=r0;

    i4=0x440004;
    i1=_rbuf;                                   /* pointer to destination buffer[0] */
    r0=dm(_TsCnt);
    r0=ashift r0 by 0xffffffff;                 /* TsCnt / 2 */

    r2=dm(i4,m5);                               /* rbuf[0] */

    lcntr=r0, DO loop3 UNTIL LCE;
        r2=ashift r2 by 0xfffffff0, r4=dm(i4,m5);
        r4=ashift r4 by 0xfffffff0, dm(i1,m6)=r2;
        r2=dm(i4,m5);
loop3:  dm(i1,m6)=r4;


    r0=gets(1);  i4=r0;
    r0=gets(2);  i1=r0;
    r4=gets(3);
    r2=gets(4);
    r0=gets(5);
    alter(5);
    exit;                                       (512 * 4) + 20  = 2068
.endseg;
```

- Assembler for SHARC without multifunction instructions and modulus

```
#include <asm_sprt.h>;
. .SEGMENT /pm    seg_pmco;
.global         _ASMCpyExtAbsFifoToMem;
.extern         _rbuf;
.extern         _TsCnt;
_ASMCpyExtAbsFifoToMem:
    entry;
            puts=r0;
            puts=r2;
    r0=i1;  puts=r0;
    r0=i4;  puts=r0;
    i4=0x440004;
    i1=_rbuf;
    r0=dm(_TsCnt);
    lcntr=r0, DO loop2 UNTIL LCE;
        r2=dm(i4,m5);
        r2=abs r2;                          /* modulus of value */
        r2=ashift r2 by 0xfffffff0;
loop2:  dm(i1,m6)=r2;
r0=gets(1);  i4=r0;
    r0=gets(2);  i1=r0;
    r2=gets(3);
    r0=gets(4);
    alter(4);
    exit;
.endseg;
```

(1024 * 4) + 17 = 4117

- Assembler for SHARC with multifunction instructions and modulus function

```
.SEGMENT /pm    seg_pmco;
.global         _ASMFastAbs;
.extern         _rbuf;
.extern         _TsCnt;
_ASMFastAbs:
    entry;
            puts=r0;
            puts=r2;
            puts=r4;
    r0=i1;  puts=r0;
    r0=i4;  puts=r0;
    i4=0x440004;
    i1=_rbuf;
    r0=dm(_TsCnt);
    r0=ashift r0 by 0xffffffff;
    r2=dm(i4,m5);                   /* rbuf[0] */
    r2=ashift r2 by 0xfffffff0;     /* 16 Bit ->> */
    lcntr=r0, DO loop4 UNTIL LCE;
        r2=abs r2, r4=dm(i4,m5);            /* modulus of value */
        r4=ashift r4 by 0xfffffff0, dm(i1,m6)=r2;
        r4=abs r4, r2=dm(i4,m5);            /* modulus of value */
loop4:  r2=ashift r2 by 0xfffffff0, dm(i1,m6)=r4;
    r0=gets(1);  i4=r0;
    r0=gets(2);  i1=r0;
    r4=gets(3);
    r2=gets(4);
    r0=gets(5);
    alter(5);
    exit;
.endseg;
```

(512 * 4) + 20 = 2068

Data Output

The results of the tasks may be output via D/A converters, transfered to other systems for further processing or visualization, controls actuators via I/O-ports, field buses, networking components etc... For D/A-converters, we use the same techniques and have the same restrictions.

Control Interface Technology

- Not all DSPs/MCUs can operate without external control devices.
- Most systems require additional devices (MCUs) for result-data transfer, visualisation and user interface capabilities.

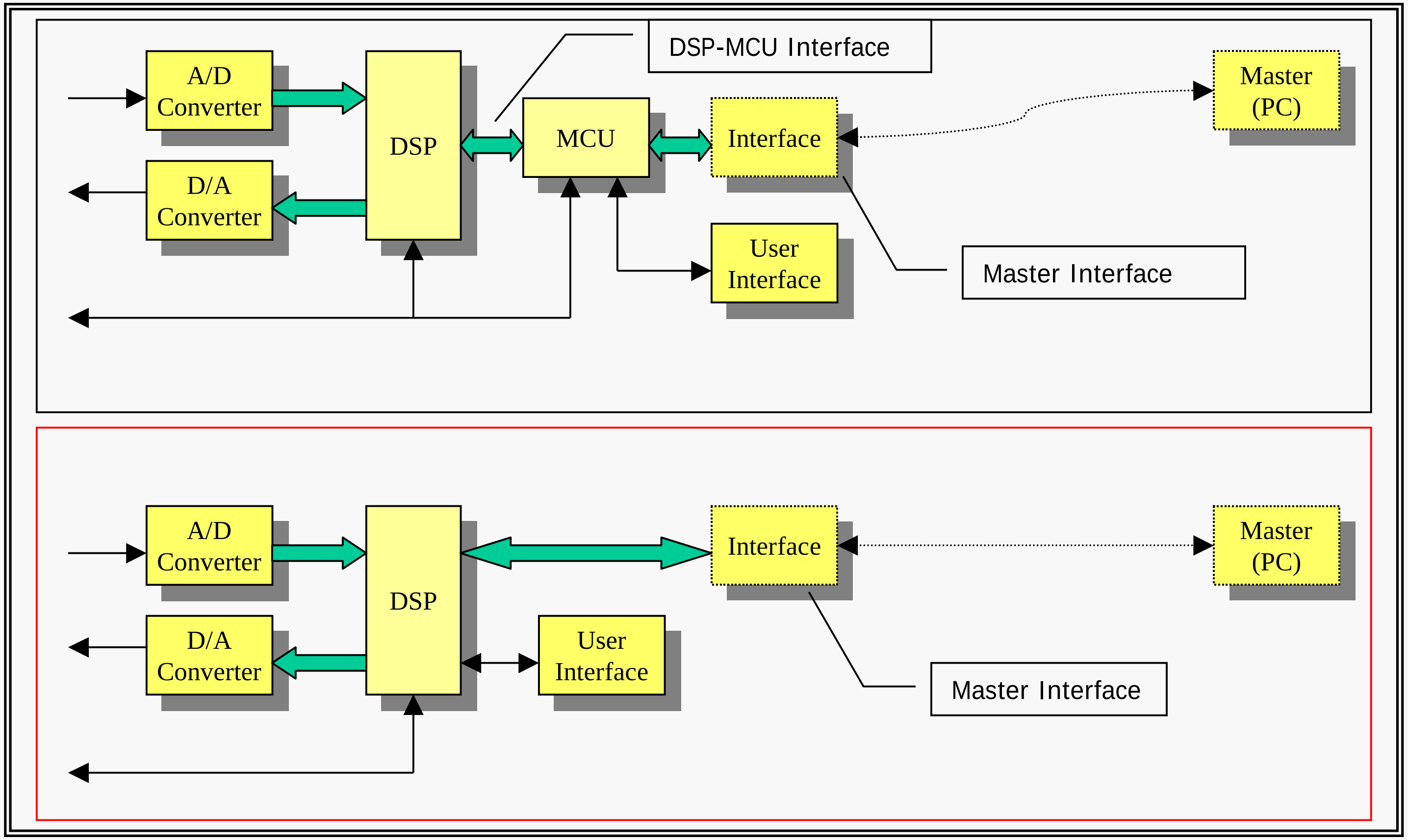
Solutions:

- Direct connection of keypads and displays to the DSP
- Direct bus interface by a micro-controller or a Master-system (Isa-Interface, PCI-Interface)
- Usage of bidirectional serial ports to conect to a HOST
- Integration of serial and parallel interfaces as RS232, RS422, RS485
- Integration of networks (Ethernet, Can, USB, Profibus)

- The choice depends on the overall application and the given requirements

Control Interface Technology

Typical structure of Embedded Systems



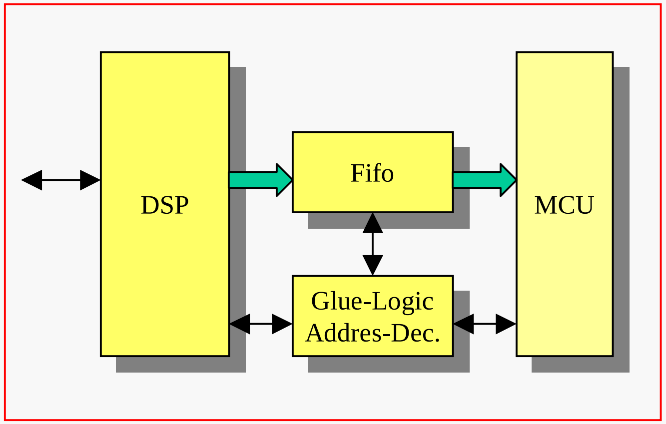
Control Interface Technology

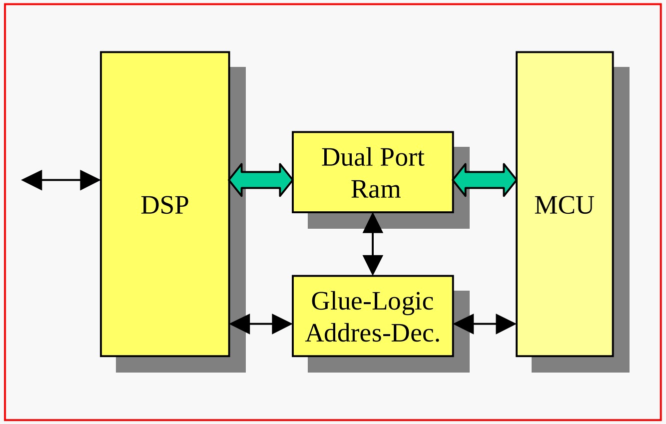
- DSP to MCU (Master) by the built in parallel Host interface
  - Host interface uses its own address, data and control siognals (independant)
  - The host will take the control over the DSP via bus arbitration logic
    - problems if the host is slow, the host locks the performance of the DSP
  - Bidirectional register or latches with interrupt signals
  - Bidirectional fifo structure with additional GLUE logic (in a Gate Array !)
  - Dual-Port-Memory with interrupt signals
  - Serial interfaces as SSI, SPI (fits in a CPLD or a small Gate Array)

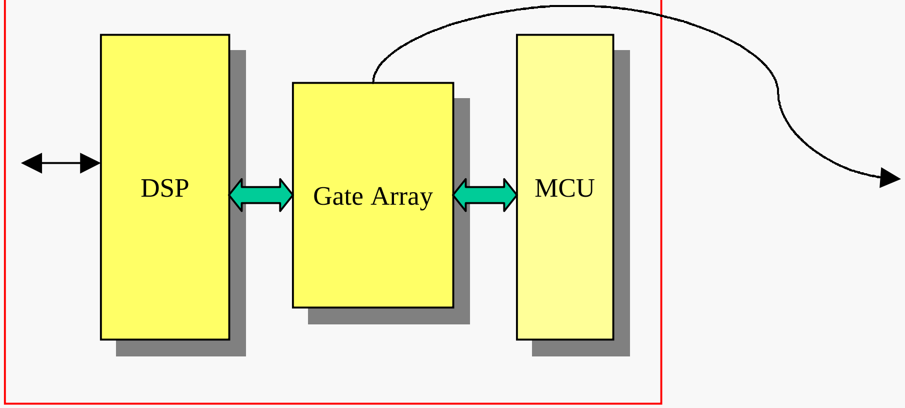
Best choice if the data transfer is decoupled !

We need a protocol !

Host Interface Technology







Gate-Array

- Bidirektional Fifo
- Bidirektional Latch
- Dual-Port
- Serial Interface (SSI,SPI)

- Multprocessing
- Industrial Interfaces
- Network and Protocol