Lecture 1, Jan 9, 2024

CPU Selection

- What do we need to look for when selecting a CPU?
  - Speed: clock rate, CPI (cycles per instruction), instruction length and complexity
  - Interfacing: GPIO count, I/O peripherals, etc
  - Other considerations: power consumption, size, in-system upgrade, hardware design features, cost
- There are also criteria external to the processor itself:
  - Design process: tools (assembler, debugger, compilers, code generators, etc), code base, training, availability
  - Production: availability (supply chain considerations), cost, suppliers, product maturity and lifetime
  - Design for future: in-system upgrade, replacement, failure rates and modes

Lecture 2, Jan 16, 2024

Number Representation

- Ultimately everything is represented in binary; but a number in binary is meaningless without an attached interpretation!
- Representing a number in binary requires us to know the signedness, bit width, and target representation
  - Integers can be represented in a variety of ways including sign and magnitude (using one bit for the sign and remaining for magnitude), 1's complement, and 2's complement
    * 2's complement has the advantage of not having two representations for zero and allowing for subtraction with normal addition logic
- To convert to binary, successively divide by 2 and take the remainder each time as the bit value, from right to left
- For non-integers, we use either a *fixed-point* or *floating-point* representation
  - In a fixed-point representation, a fixed number of bits is allocated to represent the integer part and another fixed number of bits for the fractional part
    * To get the fractional part, we successively multiply by 2 and take any integer part as the bit value, from left to right
    * Example: 27.2
      - Using 8-bit unsigned for the integer part gives `0001 1011`
      - Fractional part:
        - $0.1 \times 2 = 0 + 0.2$
        - $0.2 \times 2 = 0 + 0.4$
        - $0.4 \times 2 = 0 + 0.8$
        - $0.8 \times 2 = 1 + 0.6$
        - $0.6 \times 2 = 1 + 0.2$
        - $\dots$
      - The fractional bits would repeat forever as `0.0001100011...`
  - In a floating-point representation the position of the decimal point is not fixed
    * The bits are split into a sign bit, a signed exponent, and an unsigned *mantissa*
    * The number is recovered by multiplying the mantissa by a base raised to the power of the exponent, times the sign
    * Compared to fixed point, this can store a much wider range of numbers
    * However doing math on them requires them to be *denormalized* first, essentially converting this back to fixed point
      - This is slow, can lose precision, and can lead to bugs when the magnitudes of the floats differ by too much
- Note since the binary representation is a repeating decimal, if we truncate it at any finite digit count and convert back to decimal, we won't get the original number back

- Often by converting a decimal number to binary, we lose precision; when we do arithmetic operations on these numbers the precision loss is compounded
  - Both fixed and floating point are *lossy*
- The result of an arithmetic operation may not fit in the number representation of the operands
  - A *carry-in* can be used to implement addition at bit widths higher than a single instruction allows
  - A *carry-out* indicates that the result doesn't fit within the bit width allocated
  - If the carry-out is unhandled, it becomes an *arithmetic overflow*
    * A 1 in the carry-out position indicates an overflow
    * If a carry-out has occurred, it is usually put into a special register
  - An *arithmetic underflow* can also happen when we subtract a larger number from a smaller one when we're working only with unsigned numbers
    * This doesn't happen with 2's complement since everything gets converted to addition, so it uses overflow for both numbers that are too small or too big
- Bits can be *shifted* or *rotated*
  - An arithmetic shift left shifts the MSB into the carry and 0 into the LSB
  - An arithmetic shift right shifts duplicates the MSB and shifts the LSB into the carry
- Two's complement representation is a common way to represent negative numbers
  - The MSB acts as a sign bit; a 1 in MSB indicates a negative number
  - The remaining bits are used to store the magnitude
  - If the number is negative, the magnitude is complemented (inverted) and has 1 added to it
  - This allows us to unify the addition and subtraction logic
  - Underflow or overflow is indicated by the carry-out being different from the carry-on into the MSB (sign bit) of the result
- When we move a signed value into storage that has more bits, we perform *sign extension*, i.e. duplicating the MSB to fill any unfilled bits
  - e.g. $0b11100000$ becomes $0b11111111'11100000$ upon sign extension to 16 bits
- For multiplication/division, most modern RISC systems will do this in a number of steps:
  1. Check the signs of the operands
  2. Take 2's complement of any negative operands
  3. Multiply/divide as unsigned
  4. If operands had different signs, take 2's complement of the result
- Number representations affect our choice of microcontroller – if we need to work with a lot of large numbers or floating-point numbers, we need to choose processors that have the corresponding features, otherwise performance will be very poor
- Data can be stored in multiple different locations:
  - Data memory (often but not always RAM)
  - Instruction memory (often non-volatile, non-runtime-modifiable memory)
  - Registers
  - Memory inherent to instruction
- Going down this list, the amount of memory available gets smaller but the memory is faster

Important

Key takeaways:
1. We need to be aware of our data representation limits
2. Any calculations using data from non-controlled sources must be assertion-checked to avoid errors
3. Use assertions to be more aware of potential overflow causes in intermediate math
4. Use completley specified typing for variables; do not assume anything about variable sizes, etc
5. Avoid non-deterministic stop conditions (i.e. have fallbacks), especially when external data is involved

Lecture 3, Jan 23, 2024

Storage and Variables

Code Memory and Instructions

- Slower and larger than registers, but usually faster than data memory
- Usually not modifiable when the program is running
- This can be a region of a larger set of memory (*unified memory model*) or physically separate (*separate memory model*)
  - We need to be aware of this because it may have performance implications
- Code is compiled/assembled into *instructions*, which are stored here
- Each instruction is an atomic operation that may take multiple clock cycles to complete (measured in *cycles-per-instruction*, CPI)
  - Particularly in CISC architectures, CPI for some instructions can be much greater than 1
  - Typical goal of RISC architectures is to limit CPI to 1 for most instructions
  - CPI can also be less than 1 (e.g. multiple cores, superscalar architecture, etc)
- Actual execution time is the sum of the CPIs of all instructions multiplied by the clock period
- *Pipelining* helps achieve CPI of exactly 1 by overlapping instructions to avoid having idle hardware
  - If two instructions have multiple sub-parts that use different internal areas, we can start the second instruction while the first one is still executing
  - e.g. fetch the second instruction while the first is executing
  - This is akin to an assembly line
- Such features are mostly to the programmer, but not the hardware designer
  - These affect our ability to relate real execution time to CPI, since it adds unpredictability to execution time
  - Other features can include caching, branch prediction, out-of-order execution, etc

Registers

- The *program counter* (PC) keeps track of either the current instruction executing or the next instruction to be fetched
  - The CPU would fetch the instruction at the position, interpret it, and execute it, which modifies the program counter
  - This is a *special function register* (SFR), which are registers that have specific uses and must be accessed in specific ways
- All microcontrollers provide *general purpose registers* or *accumulators*
  - They are very limited in number, but very fast
- The register size typically matches the microcontroller word size, but occasionally divided into half-words or bytes
  - If we're working with a lot of math that exceeds the bit width of the microcontroller, this can be very slow
  - Most execution time is spent moving data between registers and slower memory
- Some instructions may concatenate multiple registers to form a larger operand
- Some older microcontrollers (e.g. 8051) implement registers in RAM, which allows for *register banking* (having multiple sets of registers that can be switched)
  - This is mostly obsolete in newer chips due to speed
- *Accumulators* are general purpose registers that are often dedicated to arithmetic
  - These are inherently used by math instructions
  - To use them as e.g. an address, they often have to be copied to another register first
  - This simplifies the instruction set since math instructions will only use the accumulators
  - Mostly a feature in modern RISC chips; CISC designs directly use general purpose registers

Memory Layout

- Most memory in CPUs will be byte-addressable, even if the bit width of the microcontroller is more than 8
- Since a word can span multiple memory addresses, we need a convention on how to store a word in multiple bytes
  - *Big endian* systems store the most significant bytes first
  - *Little endian* systems store the least significant bytes first
    - Note the individual bits are not reordered, only the byte order is
  - This is normally handled by the compiler, but may become important when we do type casting or accessing a specific bit in a mask

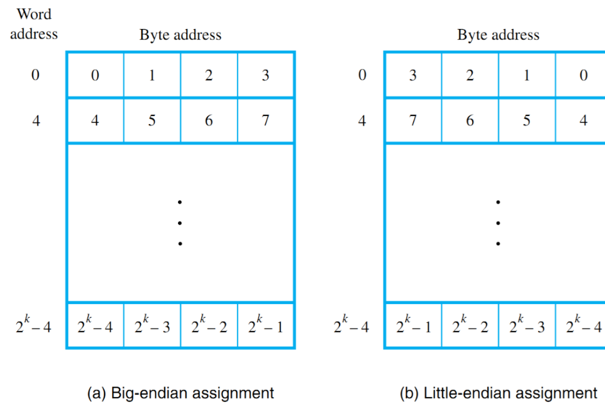


Figure 1: Big vs. little endian system.

Stack Pointer

- The *stack* is a special region of memory inherently accessed through special instructions
  - This has a variety of uses such as subroutines and temporary variable storage
- The stack and program memory may share the same address space or physical memory, or may be entirely separate
- The *stack pointer* (SP) is another SFR that tracks where the top of the stack is in memory
- What exactly the stack pointer points to varies between implementations
  - Some platforms (e.g. 8051) have the SP point to the next free space, while some others (e.g. HC12) point towards the last filled space
  - Some stacks grow downwards (SP decreases, e.g. 68HC12), while others grow upwards in memory
- Since the stack doesn't know where the data came from and how big it was, if we push a 16-bit value onto the stack and pop it into an 8-bit register, we would only get the first 8 bits while the rest stays on the stack
- Note that in most architectures, popping the stack only copies the value to a register and moves the stack pointer, without ever clearing the value in the stack
- Growing the stack past the designed limit causes a *stack overflow*, which could overwrite data or even code
- Many stack implementations are very limited, and some platforms may even have a fixed upper limit to the call depth

Code Compilation

- Each instruction is compiled into an *op-code*, which is decoded by the microcontroller
- Every variant of the instruction has its own distinct encoding
  - e.g. the `MOV` instruction can have many different op-codes depending on where it's reading from and writing to

- Instruction encoding is usually handled by the assembler/compiler and not too much of a concern

Lecture 4, Jan 30, 2024

Addressing Modes

- *Inherent addressing*: address, data, or register is inherently specified by the instruction
  - Used by instructions that either don't take any operands, or operands are fixed by the instruction
  - e.g. `NOP`
- *Immediate addressing*: a constant value is explicitly stored within the code and used by the instruction
  - e.g. `MOV R0, 0xAF`
  - For word sizes above 1 byte, this will have to respect endianness
  - Best used to implement constants, since it is faster but not modifiable
- *Register addressing*: operands are registers
  - e.g. `MOV R0, R1`
  - Generally accumulator based or RISC architectures restrict where accumulators and general purpose registers can appear as operands
  - Best used to implement transfers or variable assignment
  - Limits us to the width of our registers
- *Direct addressing*: specifies a memory location to access (i.e. a constant pointer)
  - e.g. `MOV [0xA000], 0xAF`
  - The addresses themselves are stored within the instruction and are constant
  - Best used to implement global variables, constant loading, memory-mapped I/O, etc.
  - If we only use direct addressing, we need to make all variables global in scope and pre-assign fixed addresses to every variable
- *Indirect addressing*: the memory location in the operand contains the address of the data to be used by the instruction (i.e. a true pointer)
  - e.g. `MOV R0, @[0xA000]`
    * The memory at `0xA000` is read, and interpreted as a memory address, and then data from that address is put into `R0`
  - Best used to implement pointers, arrays, etc, to implement data structures and pass data to subroutines
  - However, this uses 2 or more memory accesses per instruction, which are slow
- The below addressing modes are outside the minimum required set, so some microcontrollers might not support them, especially RISC
- *Register indirect addressing*: the register in the operand contains the address of the data to be used
  - e.g. `MOV R0, @R1` or `MOV R0, @@R1` (dereference twice, usually only on CISC)
  - Allows us to dynamically create and destroy new pointers
  - Allows doing math on pointers for e.g. array access with variable indices
  - Address space is again limited to register width, so we might not be able to access all the memory
  - Can become a vector for attack if a malicious actor can put data into the register
- *Indexed addressing*: a family of addressing modes to used to add indexing many of the previous modes
  - Relative addressing: relative to current program counter, e.g. `BRR 0x05`
    * Useful for flow control
  - Base register: adding offset to register value, e.g. `MOV R0, @[R1 + 5]`, `MOV R0, @[R1 + R2]`
    * This is very useful for array dereferencing and data structures in general
- Other exotic modes (these are much less common)
  - Base plus index plus offset: effective address is the sum of all 3 (for supporting 2D arrays)
  - Scaled: effective address is the base address plus a scale times an index (for addressing arrays)
  - Register auto-increment/decrement

Example 1: Factorial

- Example: compute the factorial of a number stored in `R0`, assume 3 general purpose registers `R0` to `R2` are available and ignore checking pre-conditions
- Assume for the purposes of this example that we cannot modify `R0`
- Translate from C code:

```
int n;
int fact = 1;
for (int i = 1; i <= n; i ++) {
    fact *= i;
}
```

- Create variable assignment table:

Variable	Assignment	Scope
`n`	`R0`	Global
`fact`	`R1`	Global
`i`	`R2`	Local to `for`

- Code generation:

```
    MOV     R1, 0x01    ; int fact = 1; // note assumption that registers are 8-bit
    MOV     R2, 0x01    ; int i = 1;
loop:
    CMP     R2, R0
    BGT     end         ; if (i > n) { break; }
    MUL     R1, R2      ; fact = fact * i;
    INC     R2          ; i ++;
    BRA     loop
end:
```

- Note we used labels here for the relative branch instructions; if not, we would have to compute the offsets ourselves
  - Assuming each instruction takes 2 bytes, the `BGT` would have an offset of `0x06` while the `BRA` would have `0xF6` (-10)

Example 2: Square Root

- Example: calculate the square root of a number S in `R0` by the following procedure:
  - Set $x_n = S/2$
  - Iterate $x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$
- Code generation:

```
    MOV     R1, R0     ; int x_n = S;
    DIV     R1, 2      ; x_n = x_n / 2;
loop:
    MOV     R2, R0     ; int temp = S;
    DIV     R2, R1     ; temp = temp / x_n;
    ADD     R1, R2     ; x_n = x_n + temp;
    DIV     R1, 2      ; x_n = x_n / 2;
    BRA     loop
```

Lecture 5, Feb 6, 2024

Branching

- A branch instruction is an instruction that modifies the program counter, changing the flow of code execution
- We will make a distinction between *branch* and *jump*:
  - *Branch* has a limited range of movement, usually relative to the current program counter
  - *Jump* can access the full range of memory and set the program counter anywhere
- A *direct branch* is the simplest variant that directly loads a value into the PC
  - e.g. `BRA 0xF5`
  - Typically a simplified or a low-cost version of a full jump since it addresses a smaller range of memory
  - Some platforms use it to jump to interrupt code
  - Essentially a `goto`
- A *relative branch* increments or decrements the PC by some value, relative to the current PC
  - e.g. `BRR 0x05`
  - The offset can be negative to jump back to earlier code
  - Note the offset usually has less bit width than the address, so we cannot access addresses that are too far away
  - We also need to perform sign extension when adding this offset to a PC with larger bit width
- Relative branching allows relocatable code, i.e. we can move a chunk of code anywhere and it will execute the same since it does not use any hard-coded addresses within itself
  - This allows the code to be compiled once and dynamically linked
  - Also allows for self-modifying code, which allows the existence of bootloaders, OSes, etc
- To calculate the offset for relative branching, subtract the current PC (after reading the branch instruction) from the destination PC
  - The offset is the address of the destination instruction, minus the address of the branch, minus the size of the branch instruction
- A *jump* simply loads the PC with the direct value in the instruction
  - This allows accessing the entire memory range of the processor
  - Typically we don't do math with jump instructions
- *Conditional* branches and jumps only occur if a condition is true
  - The test usually involves a subtraction and testing if the result is positive/negative/zero
  - The conditional branch usually relies on values in the CCR (status register)
    - This means we can branch off not just subtractions but anything that sets the CCR
  - Depending in ISA we may have branching instructions for result equal to zero, positive/negative, overflow, etc
  - Some ISAs offer a compare instruction, which is a specialized subtraction that may have benefits such as not modifying the operands, restoring the original CCR bits after branching, etc

Lecture 6, Feb 13, 2024

Code Reuse

Inline Functions and Macros

- *Inline functions* or *macros* are code blocks defined once and duplicated by the assembler or compiler each time it's used
- Since no function call is involved, they have no overhead, can be easily optimized, and is simple to implement
- However duplicating them each time means inefficient code memory usage, so we should keep them short
- Usually they will make explicit assumptions about the program state (e.g. operands are in certain registers)

    - Should only contain re-locatable code (i.e. no jumping to fixed addresses within the code block)
    - Also cannot contain recursion!
- Best used for code that are executed often (e.g. inside a loop), but called/used in few other places

Subroutines

- *Subroutines* are defined and compiled once; to execute them, code execution jumps to the memory they occupy and jumps back after they're done
    - The code does not strictly need to be relocatable
    - Reduces code memory usage since only one copy is needed
    - However, calling and passing arguments introduces some overhead, which could amount to a significant amount if the subroutine itself is short
    - Also harder to optimize since the subroutine code needs to be as generic as possible, so each call cannot be optimized by itself
- A call instruction saves the current PC on the stack and jumps to the subroutine
    - A return instruction loads the saved PC and jumps to that location, returning to the point before the subroutine call
- A safe subroutine has to back up any registers it uses on the stack as to not interfere with any calling code
    - Some CISC microcontrollers might do this automatically on call instructions
    - This is uncommon today, since it introduces large overhead and backs up all registers, even the ones that aren't used in the subroutine
        * This is still done commonly for interrupts
    - At the start of the function, we push any registers we need to use, then at the end of the function we pop in opposite order to restore them
        * Sometimes we may back up SFRs as well
- Variable passing can be accomplished in a number of ways:
    - Using registers can work if we need few parameters, and saves overhead
    - Using memory (fixed locations) require agreement on locations, and struggles for variable-length data and recursion
    - Using the stack is the best and most general option, which is preferred for an automatic implementation by compilers
        * To pass variables or return values on the stack, we need a special addressing mode that allows us to access earlier points in the stack, since the top of the stack will store the PC

Example: Call/Return Compilation Example

```c
int max(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}
void main(void) {
    int c = 1, d = 6;
    int e = max(c, d);
}
```

- Assume:
    - `main()` starts at `0x0100`, `max()` starts at `0xF000`
    - Instructions with register-only operands are one byte long, while others are one byte plus any immediates
    - We have 3 registers `R0`, `R1`, `R2`
    - `int`s and registers are 16-bit
    - Branching instructions use 8-bit offsets

  - Stack grows upwards, SP points to next available address
  - Big-endian system
  - Stack-based call and return
    * Note: in reality the compiler will likely use registers for argument passing, or optimize this into an inline function/macro due to its short length

- Variable assignment:
  - In `main()`:
    * `c` →`R0`
    * `d` →`R1`
    * `e` →`R2`
    * All variables have scope ending only when program terminates
  - In `max()`:
    * `a` →`R0`
    * `b` →`R1`
    * Both variables have scopes that exist through the entire function, and only in the function
    * Since `c` and `d` are still in scope at this point, we need to backup the registers

- Code for `main()`:

```
0x0100      MOV     R0, #0x0001
0x0103      MOV     R1, #0x0006
0x0106      PUSH    R0           ; First arg
0x0107      PUSH    R1           ; Second arg
0x0108      ADD     SP, #0x0002 ; Space for retval
0x010B      CALL    0xF000
0x010E      POP     R2           ; Retrieve return value
0x010F      SUB     SP, #0x0004 ; Clear args from stack
```

- Code for `max()`:

```
0xF000      PUSH    R0              ; Back up registers
0xF001      PUSH    R1
0xF002      MOV     R0, [SP - 12]   ; Load arg a
0xF005      MOV     R1, [SP - 10]   ; Load arg b
0xF008      CMP     R0, R1          ; if (a <= b) goto else;
0xF009      BLE     #0x0008
0xF00B      MOV     [SP - 8], R0    ; Return a (note 2 bytes for offset)
0xF00E      BRA     #0x0003
0xF010      MOV     [SP - 8], R1    ; Return b
0xF013      POP     R1              ; Restore registers
0xF014      POP     R0
0xF015      RET
```

- Stack contents are shown in the table below
  - By the start of the function call SP is at `0x2008`, right after PC
  - By the end of the call the SP is back to `0x2005`, exactly where it was before the call
  - At the end of the program, the stack pointer should be back to `0x2000` if we cleaned up properly

Address	Data	Comment
0x200B	...	R1 backup, low byte
0x200A	...	R1 backup, high byte
0x2009	...	R0 backup, low byte
0x2008	...	R0 backup, high byte
0x2007	0x0E	PC, low byte

Address	Data	Comment
`0x2006`	`0x01`	PC, high byte
`0x2005`	?	Space for return
`0x2004`	?	Space for return
`0x2003`	`0x06`	`R1`, low byte
`0x2002`	`0x00`	`R1`, high byte
`0x2001`	`0x01`	`R0`, low byte
`0x2000`	`0x00`	`R0`, high byte

Lecture 7, Feb 27, 2024

Memory Mapping

- In a *unified memory space*, the code, data, and stack are together, and the compiler can arrange these easily to optimize
- In embedded systems, we often have *separated memory spaces*
  - Compilers have a hard time understanding and optimizing these, because some sections of memory can support different types of access, have different speeds, etc
  - We may need to e.g. explicitly tell the compiler to put something in non-volatile memory so it doesn't take up RAM
- Typically external memory cannot be utilized automatically by the compiler
  - Some microcontrollers may have DMA to access these, but compilers doesn't understand these
  - The way to do this typically varies per-chip and per-compiler
- Therefore we need to understand how the compiler does low-level optimization (what it does and doesn't optimize) to write good high-level code

Optimization and Compilation

- For embedded systems, compiler toolchains are often a lot less developed than for desktop architectures, so we need to be mindful of writing optimized or easy-to-optimize code
- We usually optimize for one of two metrics: execution time and program space
  - Often faster code is shorter code, but this is not always true (e.g. unrolling a loop)
- Optimizing code too heavily will lead to unreadable and unmaintainable code
  - Only optimize timing-critical or heavily reused/iterated code
- There are 3 common forms of optimization:
  - *Global optimization*: making algorithmic changes to the code over multiple lines
    * This is high-level (before code generation) and often difficult for compilers
  - *Local optimization*: making optimizations within a single line/expression/statement
    * Even basic compilers generally do this well
    * We are not expected to do this by hand
  - *Keyhole/peephole optimization*: runs on the final assembly code and optimizes a few instructions at a time
    * Uses a sliding window and tries to match known optimizations
    * No human intervention
    * When two templates meet, there can be inefficiencies, e.g. storing a register into memory only to load it back again; this will be optimized away by keyhole optimization
    * Note that we can't always apply an optimization, due to e.g. interrupts, SFRs, DMA, etc
- Global optimization techniques:
  - *Loop unrolling*: expanding a fixed-length loop into repeated code
    * Increases code size but avoids overhead of looping
    * Unrolled code is less readable and harder to maintain
    * Compilers are often good at this, but only if the loop length is well-known (constant)
  - *Code motion*: factoring out unchanging code from inside a loop

    * e.g. if we're indexing a constant index of a constant array in a loop, we can move the indexing outside the loop and only do it once
    * Certain embedded compilers will be able to do this
    * Note that this uses extra temporary storage
  - *Strength reduction*: using loop structures to convert more complex/slower operations into simpler ones
    * e.g. if we're assigning `x[i] = c * i` in a loop over i, we can convert the multiplication to successive additions
    * Array operations are often a common source of these optimizations
    * Whether this is worth doing depends heavily on the platform and how fast each type of instruction executes
  - *Common sub-expression elimination*: factoring out common sub-expressions that are used multiple times, and only doing it once
    * Compilers do this pretty well
  - *Lookup tables*: using pre-computed lookup tables instead of computing everything
- Local optimization techniques:
  - *Coalescence*: using the instruction set (side effects) to compile multiple operations
    * e.g. `x = x + y` would normally be 4 instructions (2 loads, 1 add, 1 store) but can often be optimized to just one
  - *Constant folding*: pre-calculating constant values
  - *Local strength reduction*: strength reduction within a line
    * e.g. multiplication by powers of 2 to shifting, exponentiation to repeated multiplication, etc
  - *Machine idioms*: making use of specialized instructions on the microcontroller
    * e.g. counting down instead of up in a loop if comparison with zero is faster

Interrupts

- *Interrupts* are events that asynchronously affect the program flow
  - Calls to the *interrupt service routine* (ISR) are done automatically in response to the *interrupt source*
  - The ISR is called like a regular function, but it can be called at any time in the program flow
- Interrupt sources are often external, but we can have internal software-generated interrupts (SWI) as well
- The ISR is a special subroutine executed on an interrupt; since it can be called at any point during program execution, it must:
  - Make no assumptions about program or microcontroller state (e.g. register contents)
  - Make no (unexpected) modifications to the microcontroller state on exit
    * Back up all registers and SFRs that we use
    * Some architectures back up registers automatically on the stack
    * Returning is usually done via a specialized return from interrupt instruction
  - As fast and short as possible, and have a deterministic exit condition (i.e. won't hang)
    * Avoid:
      - Extensive shared resource use
      - Calls to additional subroutines (which can use a lot of stack)
      - Waiting for hardware, polling, delays
    * Use timeouts or failsafes
- Interrupts are typically not enabled by default, so we have to set them up first
  - In ASM we define or set jump points through an *interrupt vector table* or set of SFRs
  - In high-level code this is often done with function pointers, pragmas, etc
- The vector table is used by the CPU to look up the address of the ISR to jump to
  - Often we don't have a one-to-one mapping from interrupt source to ISR due to hardware cost
  - In the ISR we need to scan through and identify the source of the interrupt
    * This can be costly for heavily overloaded ISRs, especially external interrupts
- Interrupts can come from a number of sources:

    - External interrupt lines (IRQ lines)
      * Triggering can occur on positive or negative edge, level triggering, or user-defined
    - Peripheral events (e.g. timers, ADC, protocol peripherals)
    - Software interrupts (SWI)
      * This can be used for exception handling, multi-tasking, debuggers, etc
- All platforms provide a method to *mask* interrupts, selectively enabling or disabling interrupts
    - However, even when interrupts are masked, they often still accumulate in hardware and will trigger once we restore interrupts
    - We usually disable interrupts of the same source while in the ISR for that source, since the ISRs themselves can be interrupted
    - This is done using a `CLI` instruction to clear masks (allow interrupts) and `SEI` instruction to set masks (disable interrupts)
      * Modern systems will have SFRs per interrupt source
- Some critical interrupts cannot be masked (*non-maskable interrupts*, NMIs)
    - This is used for critical tasks like bootloaders, watchdogs, e-stops, etc
- If two interrupts occur simultaneously, or we have multiple interrupts waiting after clearing the mask, *interrupt priority* is used to determine which gets handled first
    - This is often configurable in modern microcontrollers through a table; older platforms have a fixed table
    - Note this completely ignores order of arrival; higher priority interrupts are always handled before lower ones
    - We need to be more careful with high-priority interrupts so they don't monopolize the CPU

Lecture 8, Mar 5, 2024

Interrupts – Data Sharing

- Transfer of data to and from ISRs is done through shared global variables
- Since interrupts can occur at any time, this can lead to *race conditions*
- The interrupt may occur and write to a variable during a sequence of operations in which the variable is assumed constant
    - e.g. comparing two variables that are set in an ISR; the interrupt can occur after the main code reads one of the values, and update the other value before it is read from main, leading to inconsistency
    - This can occur at the end of any assembly instruction, and not just high-level code constructs
    - An interrupt can occur in the middle of a line of code that gets translated to multiple instructions!
- *Critical sections* access important shared resources/variables and thus must prevent multiple access to those resources
    - Disable interrupts at the beginning and re-enable them at the end to prevent multiple access
    - Critical sections must be as short as possible, since disabling interrupts directly increases system latency
      * Critical sections that are too long can lead to comatose states
    - To reduce their size, make local copies of shared data, e.g. for a comparison, load the two variables into temporaries in a critical section and compare the temporaries outside the critical section
- Some compilers have the ability to disable certain unsafe optimizations
    - e.g. `volatile` in C tells the compiler that the variable's value may change at any time

Interfacing Sensors

- Sensors can be categorized by output type broadly into either analog or digital
    - Within analog output, we can classify sensor output into either DC/low rate-of-change (ROC) or a general continuous-time signal (higher frequency)
    - Within digital output, we can classify sensor output as word-based, encoded into something like PWM, or some other communications protocol

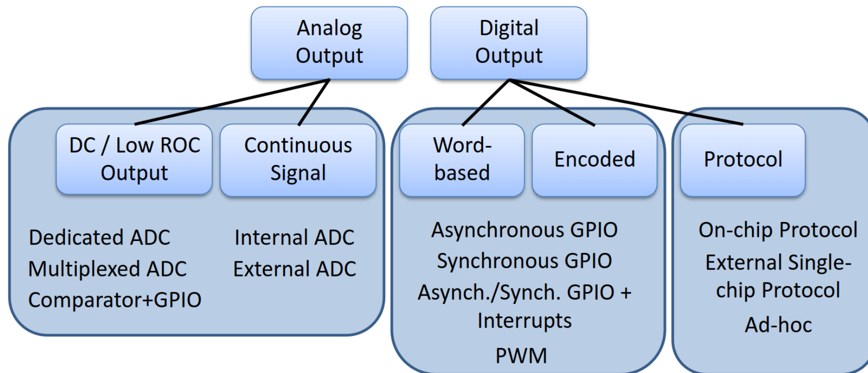


Figure 2: Types of sensor output along with preferred handling methods.

- Sensors can also be classified by output data frequency (i.e. how often the data changes)

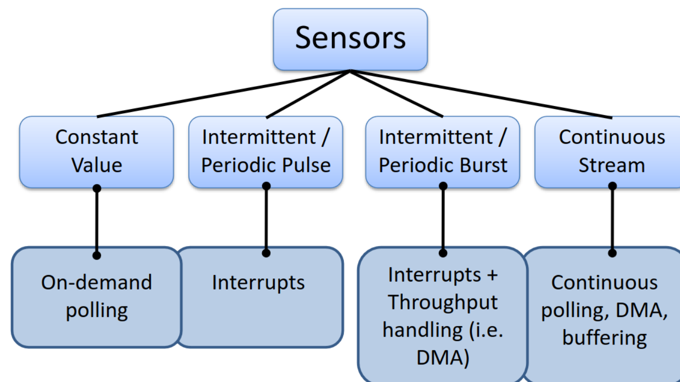


Figure 3: Types of sensor frequency along with preferred handling methods.

1-Bit A/D Conversion

- This would be used for a signal that changes slowly or takes on only a few fixed values
- We want to compare the analog voltage level with a threshold to generate a 1 or 0
- This can be done using a *comparator*, which is like an op-amp without a feedback loop, so it outputs either 1 or 0 depending on which input voltage is higher
  - This allows for high-speed comparison
  - If we input a triangle wave and a reference voltage to a comparator, we can implement PWM by changing the reference voltage
- Many modern platforms have built-in comparators, the inputs of which can be mapped to GPIO pins via SFRs
- For input voltages near the threshold, we could have the result change rapidly between 1 and 0 due to oscillations, caused by noisy data, control lag, etc

- This can be addressed with *hysteresis*: split the reference into an upper and lower bound; if the input is over the upper reference, output 1; if it's under the lower reference, output 0; if it's between, hold the previous output
  - This can be implemented with a changing reference signal; when output is high, set reference to low bound, and when output is low, set reference to high bound
  - This is good for dealing with noise in either the signal or the reference itself
  - Another application is quasi-digital signals such as button presses, encoder outputs, etc
  - In analog, hysteresis is implemented using a positive feedback from output to reference, or via

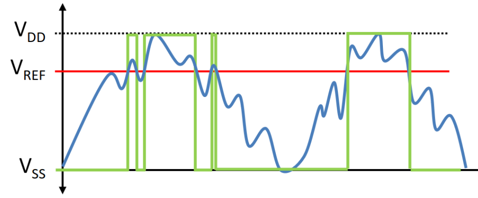


Figure 4: Incorrect triggering of output due to noise in the signal.

dedicated triggers such as *Schmitt triggers*

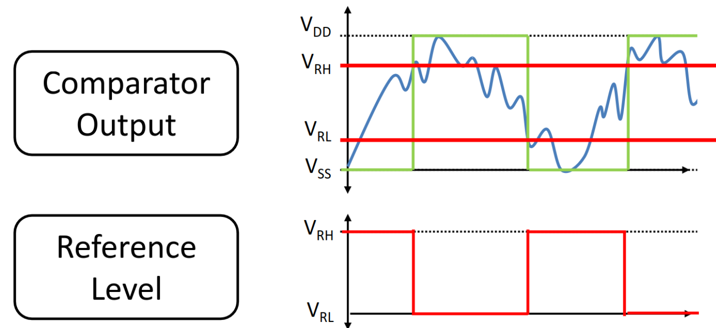


Figure 5: Illustration of hysteresis.

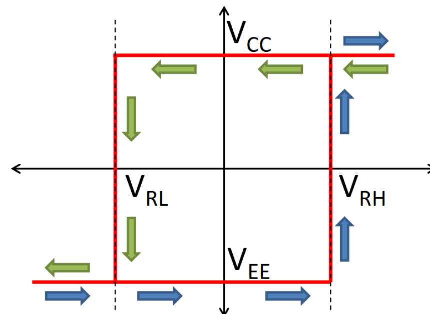


Figure 6: Illustration of hysteresis.

- The effects of hysteresis on a slow-responding control system is to increase the oscillation period (so the output switches less frequently), but the magnitude will increase
- Filters can be used on noisy data
  - These can be implemented in hardware or software depending on tradeoff of added parts/circuit complexity vs. added code complexity/load on CPU, and sampling rate requirements
  - Noisy signals can be smoothed out, but this introduces information loss and a phase shift
  - Best applied for high-frequency noise that is distinct from the frequency of the signal
  - However, this often can't eliminate all noise and incorrect switching by itself
- Another method is to use triggering methods, such as *multivibrator circuits*
  - Switch debouncing is one common application
- Multivibrators come in 3 variants:
  - Astable: an oscillator (not useful for us)
  - Monostable: a single stable state that the multivibrator will stay in; the state switches to an unstable state on some signal input, and switches back after a set amount of time
    * The time parameter should be tuned based on actual hardware, long enough to ignore temporary noise but short enough to not miss data that comes after

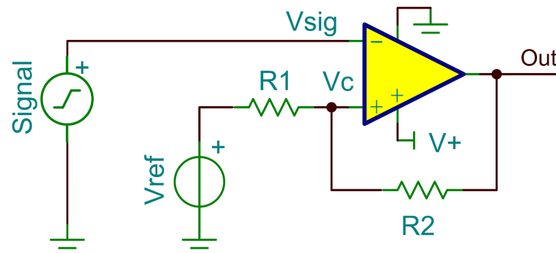


Figure 7: Analog comparator with hysteresis.

      * This effectively makes all pulses at least a certain duration wide
    - Bistable: both states are stable (needs an external reset trigger signal to reset the state)
- While hysteresis needs to be implemented in the comparator itself, a multivibrator can be attached after the comparator to have the same effect
    - This is useful when we have comparators in hardware that we cannot modify
    - This also makes more sense to do in software, as we do not need true digital filtering

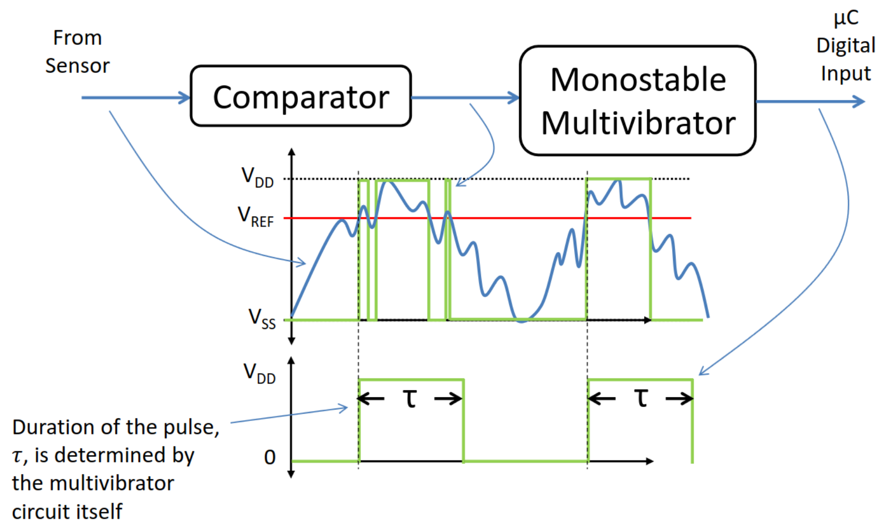


Figure 8: Response of a monostable multivibrator trigger.

Lecture 9, Mar 12, 2024

General A/D Conversion

- Generally ADCs consist of two components: *sample & hold* and *quantizer*
    - The sample & hold takes a snapshot of the input voltage level (on sample clock) and holds it constant, even if the input changes
        * This is needed because quantization takes time, so the input to the quantizer cannot change during this time
    - The quantizer maps the input voltage to an n-bit signal and put into an SFR
        * More modern processors with DMA can write this directly to a location in shared memory
- The input voltage in the range of 0 to V_{ref} is mapped to the entire digital output range
    - Having a stable reference voltage is important
    - Internal ADCs often have reference voltages that can vary due to temperature changes, etc, causing the ADC output to drift over time
    - External ADCs do a lot to make V_{ref} constant

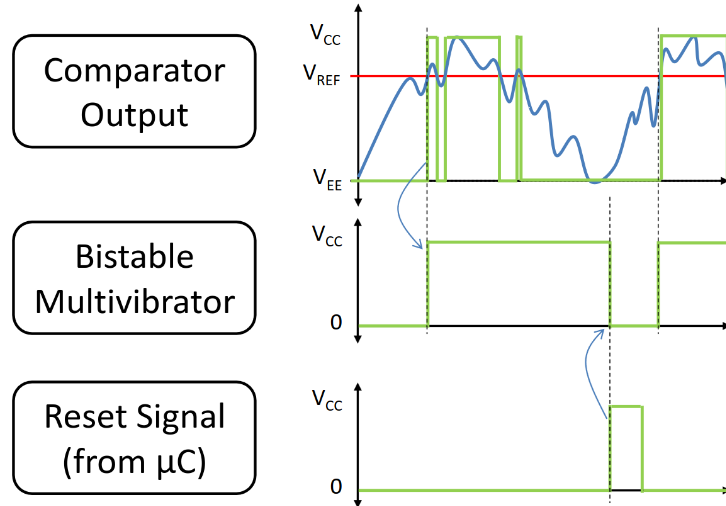


Figure 9: Response of a bistable multivibrator trigger.

- A naive quantization scheme would simply divide the full range into the 2^n possible output values and maps input to output evenly
  - However, for the max and min input levels, the quantization error is equal to $\frac{V_{ref}}{2^n}$ since we can't go below zero or above max
- A more common quantization scheme will use only half a division at the top and bottom of the input range
  - Now each digital unit maps to a range of $\frac{V_{ref}}{2^n - 1}$
  - The worst-case quantization error is now $\frac{1}{2} \cdot \frac{V_{ref}}{2^n - 1}$
  - More modern hardware will do this, because the precision lost by doing this becomes negligible at higher bit widths

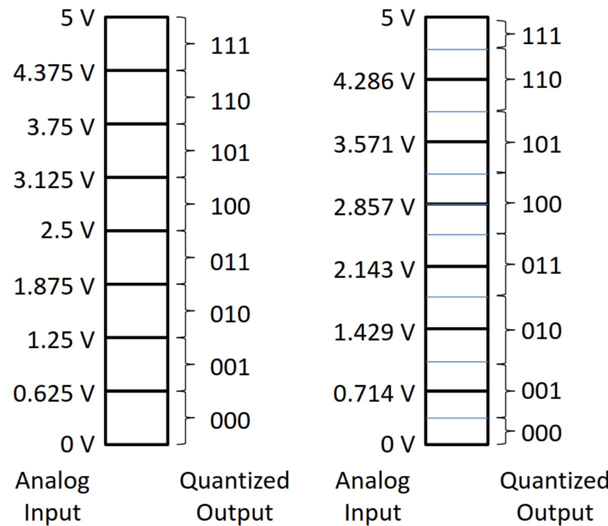


Figure 10: Different quantization schemes. Left: naive quantization, right: better quantization.

- Most ADCs can only sample at fixed intervals; they require setup time, hold time, and output time,

which is often significant compared to the CPU clock
  - Many features can increase this time, e.g. hardware oversampling/averaging
  - ADC data will lag behind real time!
- The *Nyquist-Shannon Sampling Theorem* states that we must sample at at least twice the frequency of the highest component of the input, otherwise we will get aliasing
  - Some ADCs will have Nyquist filters built-in to avoid this (the input will be passed through a low-pass filter)
  - If the ADC does not have one built-in, we need additional hardware to do the filtering, since it's not possible to do this in software

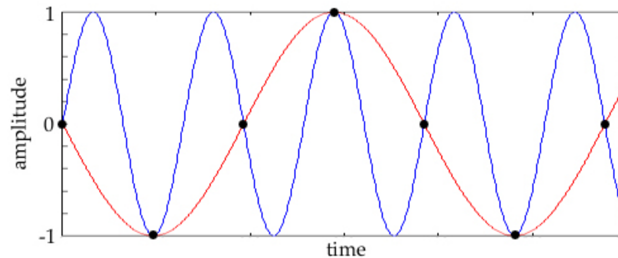


Figure 11: Illustration of aliasing.

External ADCs

- Most microcontrollers have internal ADCs, however these often have low specs when it comes to bit width, V_{ref} stability, accuracy, speed, etc
- External ADCs can send the data in parallel, serial, serial shift register, or some other protocol
- Depending on the ADC implementation/structure, we can get very different conversion speeds, precisions, etc
- A *flash ADC* uses a ladder of 2^n comparators to directly quantize the input
  - The output is generated almost instantly
  - Highly impractical for large bit widths
- *Single-slope ADCs* use a triangle/ramp signal (from a counter + DAC) and a comparator to find the signal level
  - The counter counts up and raises the reference; whenever the reference exceeds the input, we take the current value of the counter as the output
  - However this is very slow, so true single-slope ADCs are mostly obsolete
- A *dual-slope ADC* counts up as well as down, giving essentially twice the conversion speed
- An *integrating/multi-slope ADC* tries to keep the reference near the input signal so we don't waste too much time counting
  - These work the best on slowly-changing signals but bad on rapidly changing signals
  - Integrating/multi-slope ADCs are complex and expensive but much faster

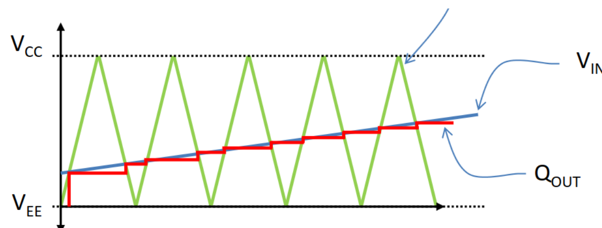


Figure 12: Dual-slope ADC.

- *Successive approximation ADCs* (SARs) essentially performs a binary search to find the correct output value

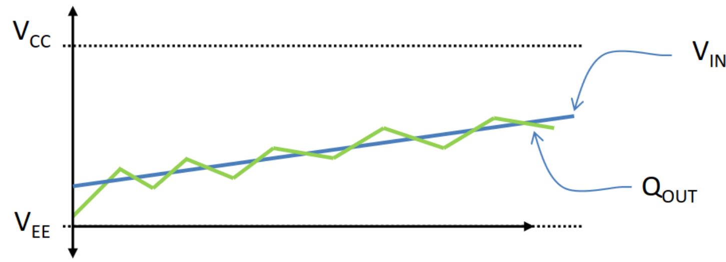


Figure 13: Integrating/multi-slope ADC.

  - This is used by the lab HCS12s
  - This doesn't scale well with larger bit widths and faster speeds, but they are still good and common in older hardware
- The more precise the output, the more time and complexity the ADC will take

Type of ADC	Conversion Speed (Comparative)	Cost/Hardware Complexity per Bit of Precision (Comparative)
Flash	Very High	Very High
~~Single-Slope~~	*Mostly obsolete*	
Dual-Slope	Low	Low
Multi-Slope / Integrating	Moderate to High(1)	Moderate to Low
Successive Approximation	Moderate to High	Moderate to High

Figure 14: Comparison of ADCs.

Digital I/O

- Some CPUs have dedicated instructions for I/O
  - This is uncommon because this ties the ISA to the actual pin count of the chip
- Most CPUs use SFRs to control I/O; often one SFR is used per *port*
- Pin configuration (input or output, open drain, etc) is also set via SFRs
- In older hardware, each physical pin on the microcontroller has only a small number of functions or a single fixed function
  - This is convenient for the programmer, but hard to expand, is wasteful of pin count, and hard to design the wiring
- In modern hardware each pin often has many selectable functions; each pin is multiplexed
  - e.g. modern PIC chips have a peripheral pin select (PPS) system that allows near-complete remapping of pins
  - This allows much easier layout of hardware

    - Often accomplished through a matrix mapping
- When interfacing with digital I/O, we must be mindful of:
    - Voltage ranges of signals (e.g. 3.3 or 5 volt logic)
    - Noise margins of different parts (i.e. the ranges that are considered high/low; this can be considered different even within 3.3 or 5 volt logic)
        * When one device outputs a logic high, it could be anywhere within the noise margin
        * We need to ensure that the noise margins match or the margin of the source device is contained within the margin of the destination device
        * Many different standards exist for noise margins and logic thresholds
    - Fan-in/fan-out (i.e. loading effects when driving multiple things)

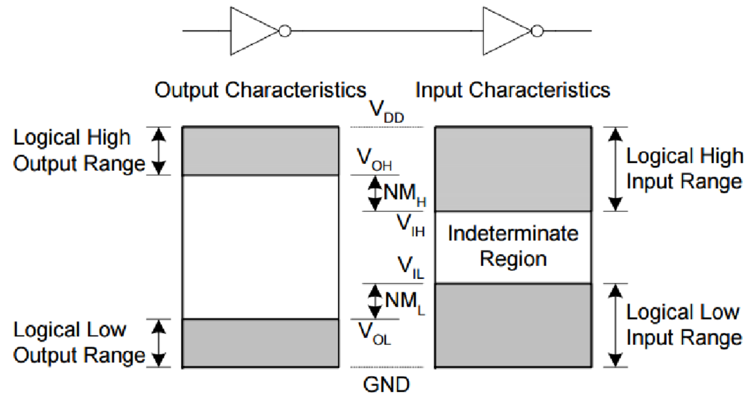
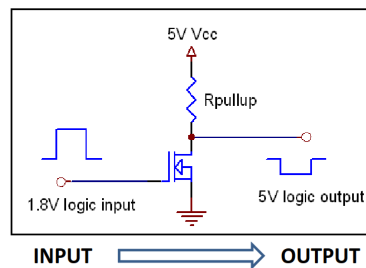
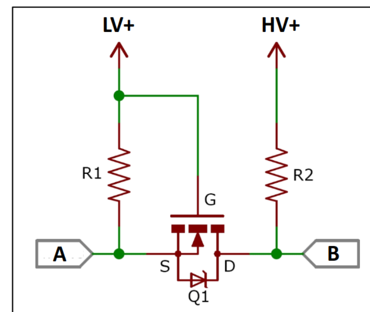


Figure 15: Noise margin compatibility.

- Some devices have compatibility with different logic levels; a lower voltage device can sometimes interpret a higher voltage input and withstand it without damage
    - This does not guarantee that it works the other way around!
- If logic levels and noise margins are incompatible, we need hardware logic level shifters; these are often simple and easily implemented with MOSFETs
    - However, these add a slight propagation delay



Example 1 – Discrete, unidirectional 1.8V-to-5.0V level shifter



Example 2 – Discrete, bidirectional level shifter; supports multiple voltage standards, as given by supplies LV+ and HV+

Figure 16: Example designs for logic level shifters.

Lecture 10, Mar 19, 2024

General-Purpose I/O – Additional Considerations

- Pins will sometimes have diodes to V_{DD} and V_{SS} for purposes of ESD protection, so the voltage on the pin is clamped to a certain range
  - Note this can cause problems if other parts of the system start up first, they may unintentionally power the microcontroller through a GPIO and put it in an indeterminate state
- Some pins are able to be configured as input, output, and a third, high-impedance mode; these are known as *tri-state outputs*
  - In the high-impedance mode, the pin is not actively driven to a logical high or low
  - This is very useful for shared data lines such as in buses, so we don't have multiple drivers interfering
- Tri-state outputs are usually configured through an SFR; when the tri-state control SFR is enabled, the pin will be high impedance, and the internal driving circuitry is entirely disconnected
- We can also achieve a similar effect with open-collector/open-drain outputs, where the pin is connected to ground through a MOSFET or BJT, and an external pull-up makes the pin high when it's disconnected from ground
  - More common in older hardware
  - This has a tradeoff between speed and efficiency
  - If we want to go faster, we need a smaller pullup to charge the line capacitance faster; but smaller pullups are less efficient
  - This was used originally in applications such as the I2C bus
  - In an open-drain configuration, the bus state will be low if any device outputs a low, so driver conflicts are impossible
- Whenever we have uncontrolled input to a pin, we should add protection circuitry
  - Common types include external protection diodes, purpose-built transient/ESD suppression diodes (often two Zener diodes back to back), or a transceiver/buffer/filter to act as a sacrificial device

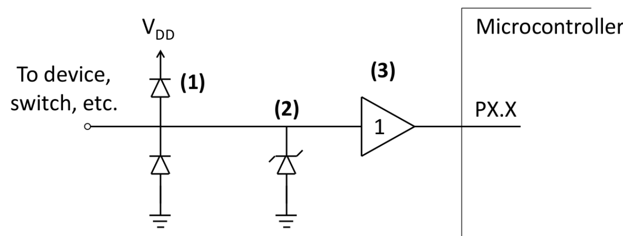


Figure 17: Pin protection circuits.

- Additional external multiplexing can be used if we run out of GPIOs
  - This is better than simply choosing a chip with a larger number of pins
  - Introduces a propagation delay
  - A multiplexer takes an input and a number of address bits, and connects the input to one of the outputs depending on the address
  - A decoder takes a binary encoded number, and outputs a pattern of bits depending on the input number
- If we need a very large number of I/O pins, there are dedicated I/O expansion chips that communicate over protocols
  - This offers many more advanced features such as tri-state outputs, interrupts, etc that you would expect in normal GPIO

Bus-Based I/O

- On older systems, the I/O ports are connected as devices on a shared data bus, which may be exposed externally

  - This requires adding a lot of external hardware, so this is rare today
- This is almost infinitely expandable (as long as we have free memory addresses), but very complex to implement
  - To the microcontroller this is indistinguishable from a normal read/write to any other memory, so it's easy to use in software
- This makes the I/O memory mapped instead of SFRs – reading/writing to a range of memory addresses will get picked up by the devices on the bus, and respond with the appropriate action
- Memory is often divided into multiple banks
  - Each bank starts its addressing from 0, so we need a decoder that takes the high bits and enables the correct memory bank for the address
  - The lower bits are sent directly to the memory bank
  - Any external devices on the bus also need a decoder
- Since the bus is shared, only one device can drive it at a time
  - If we have bus conflicts, we can irreversibly corrupt the internal state of the microcontroller

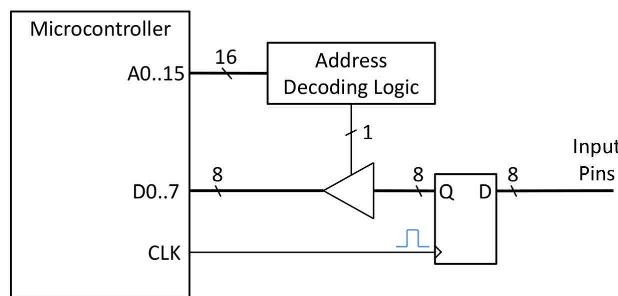


Figure 18: Example layout for an external input port.

Communication Protocols

- *Serial interfaces* send out the data one bit at a time, usually synchronized by an additional clock signal
- *Parallel interfaces* send out a group of bits at a time, usually an entire word
- Parallel interfaces used to be prevalent due to their higher speed (back then), but they have issues with noise immunity, driving, and bus size, so modern devices are usually serial
  - Serial interfaces today are often faster than parallel ones due to their simpler logic and hardware

Name	Sync?	Type	Duplex	Max. Nodes	Typical Max. Speed	Comparative Max. Distance	Wire Count
RS-232	Async	Peer	Full	2	115200 bps	Long	2+
RS-422	Async	Multi-drop	Half	10+	10 Mbps	Long	1+
RS-485	Async	Multi-point	Half	32+	10 Mbps	Long	2
I^{2}C	Sync	Multi-Master	Half	-	5.0 Mbit/s	Short	2
SPI	Sync	Multi-Master	Full	-	N/A (10MHz+)	Short	(3+1)+
Microwire	Sync	Master/Slave	Full	-	N/A (MHz+)	Short	(3+1)+
1-Wire (Dallas)	Async	Master/Slave	Half	-	16.3 kbps	Short	1

Figure 19: Common serial interfaces.

- SPI and I2C are two of the most common serial protocols used today
- I2C is designed for compatibility, so it is designed to be as general as possible

- However different hardware might have different electrical specs, so they might not be compatible on the same bus
  - I2C can often be bit-banged if needed
  - Much slower relative to SPI
- SPI is designed for speed and usually requires dedicated hardware
  - e.g. ESP32 uses 4 SPI buses to communicate with external memory
  - Does not make sense to bit-bang
- Many standards are based on UART (Universal Synchronous Receiver/Transmitter)
  - While I2C and SPI both have hardware expectations, UART does not
- Other high-level standards exist to handle specific applications:
  - CAN (Controller Area Network): usually used in automotive applications; usually layered over RS-485; designed for high noise immunity and range due to its differential voltages and CRC checksums
    * Looks more like a network than a bus protocol – data is broadcasted, and devices choose to respond based on the CAN frame
    * I2C and SPI are designed for very short buses (a few inches) while CAN can be much longer
  - Bluetooth/ZigBee/Xbee/etc: short-range wireless standards
  - USB (Universal Serial Bus): long-range, designed for high noise immunity and throughput
  - JTAG: standard for programming and debugging embedded devices
  - 802.3/11(a/b/g/n): wired/wireless networking standards

I2C (Inter-Integrated Circuit)

- A two-wire interface, with a shared, bidirectional data bus, and a clock line driven by the master; multi-slave (and multi-master in theory, but rarely in practice due to poor support)
- Data lines are open collector or open drain, as previously mentioned
  - Modern versions of the standard expand to support tri-state outputs
- Half-duplex (can send or receive, but not at the same time)
- Bitrates can include 0.1/0.4/1.0/3.4/5.0+ Mbit/s

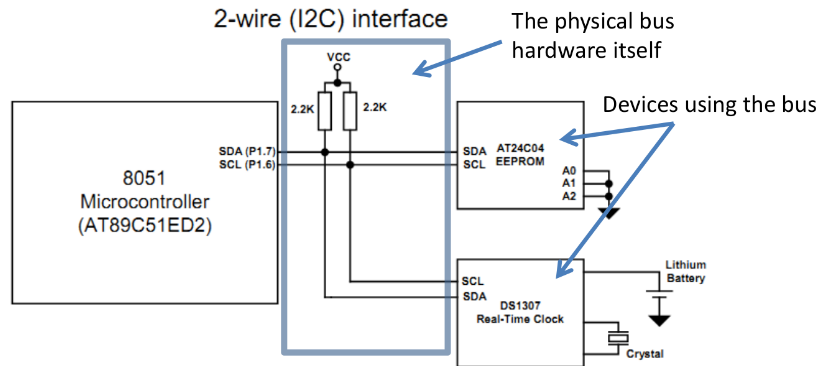


Figure 20: Example I2C bus setup.

- Communication process:
  1. Bus master sends the address of the destination device to the bus
  2. Slave devices monitor the bus for their address
     - Although in theory we can have many devices on the same bus, in practice we often have address incompatibilities from different manufacturers
  3. Handshake occurs once slave recognizes its address and communication starts
- An I2C bus handshakes using a *start condition*: holding SDA low for a certain amount of time while SCL is high
  - Note that since we have the pullups, the bus idles with both lines high
- Communication ends with a *stop condition*: holding SDA high while SCL is high

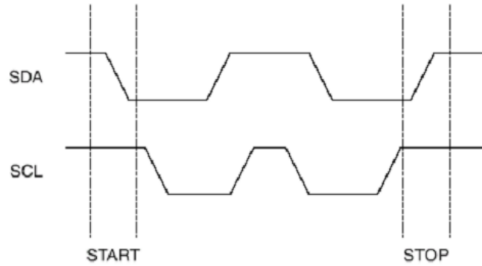


Figure 21: I2C start and stop conditions.

- Between the start and stop conditions, the clock is pulsed, and data is latched on rising edge
  - SDA should be set to the correct level before the rising edge and be held stable while SCL is high
  - Therefore SDA can only change when clock is low
- After the end of a word (or multiple words depending on device), the receiving device sends back an ACK
  - The sending device stops driving SDA and waits for the receiving device to assert SDA low for the ACK
  - The master will pulse SCL one more time for this
- Note for many I2C devices, if we send a malformed packet (i.e. not properly framed by start/stop), its state machine may not be able to recover
- Different I2C devices may have different word sizes, different number of words per transfer, etc

Lecture 11, Mar 26, 2024

Software I2C Implementation Example

- We want to write a software I2C implementation to interface a 24C02 EEPROM chip and write a single byte
- The payload consists of:
  - Address byte: first 4 bits are `0b1010`, next 3 bits are the I2C device address, final bit is R/W flag
    * Depending on the chip size, the 3 bits are divided differently into address bits and page number bits
  - Word address: 8-bit memory address of the byte to write to
  - Data: 8-bit data byte to write

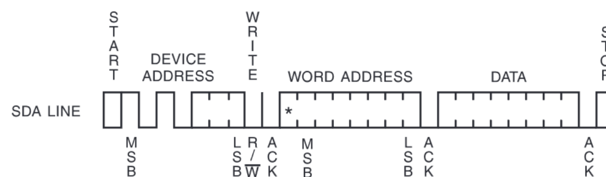


Figure 22: Payload format for a single byte write for the 24C02.

- Assume SDA is connected to P0.0 (alias `_SDA`), SCL is connected to P0.1 (alias `_SCL`), CPU with no hardware I2C support
- We're asked to:
  - Write a flexible, extensible software I2C implementation (i.e. a library) to send a single byte to a specified I2C address (low-speed mode)
  - Use the code to write 0x51 to the 2K version of the EEPROM memory at address 0xA2 at I2C device address 0x04 on the bus
- Generally we split up the code into several layers:

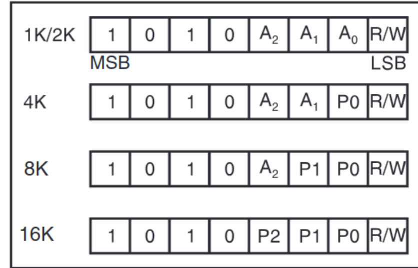


Figure 23: Device address format for the 24C02.

- *Application layer*: high-level user code that calls functions within the library or protocol layer; does not directly interface with hardware
- *Library/protocol layer*: defines common operations that are part of the core protocol specification, e.g. start/stop/send for I2C and read/write for the EEPROM; calls the hardware/physical layer
- *Hardware/physical layer*: code that interfaces with the exact hardware used, e.g. setting pin modes, specific timing, using SFRs

```
#define _SDA P0.0
#define _SCL P0.1

/***** Physical Layer *****/

void initPhysical(void) {
    // [Hardware setup, including pin mode configuration, pin speed, etc]
}

inline void setSDA(void) {
    // [Possibly (re-)set SDA as output]
    // [Wait for setup time]
    _SDA = 1;
    // [Wait for hold time]
}

// Omitted but similar to above
inline void clearSDA(void);
inline void setSCL(void);
inline void clearSCL(void);

/***** Protocol Layer *****/

void i2cInit(void) {
    initPhysical();
    // Set idle state of I2C pins
    // We do this here instead of initPhysical() since this is specified by I2C
    _SCL = 1;
    _SDA = 1;
}

void i2cStart(void) {
    // Pre-condition: SDA and SCL both high
    ASSERT(_SDA && _SCL);
    clearSDA();
}
```

```
void i2cSend(uint8_t data) {
    // Pre-condition: SDA low, SCL high
    ASSERT(!_SDA && _SCL);
    for (uint8_t i = 0; i < 8; i ++) {
        // Bring SCL low, so SDA can change
        clearSCL();
        // Send data
        if (data & 0x80)
            setSDA();
        else
            clearSDA();
        data <<= 1;
        // Bring SCL high again to get ready for the next bit
        setSCL();
    }
    // Ack: bring SCL low, set SDA to input, read ack, bring SCL high again
    clearSCL();
    // Should be added to physical layer
    releaseSDA();
    // Should be added to physical layer
    if (!readSDA())
        // Should be added to one of the layer depending on functionality
        // This depends on the hardware, the application, etc
        handleError();

    // Reset the pin and bus states so that we can send again
    // Should be added to physical layer
    driveSDA();
    clearSDA();
    setSCL();
}

void i2cStop(void) {
    // Pre-condition: SDA low, SCL high
    ASSERT(!_SDA && _SCL);
    setSDA();
}

/***** Application Layer *****/

void main(void) {
    i2cInit();
    i2cStart();
    // Address: 0b1010 to start, address of 0b100, write mode (0)
    i2cSend(0b10101000);
    // Word address
    i2cSend(0xA2);
    // Data
    i2cSend(0x51);
    i2cStop();
}
```

Lecture 12, Apr 2, 2024

SPI

- A 4-wire serial interface with one master and multiple slave devices (in theory can be multi-master, but never used)
  - SCLK (serial clock)
  - MOSI (master out slave in)
  - MISO (master in slave out)
  - SS (slave select), or CS (chip select)
    * Asserted when this device is being talked to
    * One per slave device
- Designed for much faster speed than I2C
  - Since MOSI and MISO are separate, the protocol is full-duplex (send and receive at the same time)
  - Using SS instead of an address also makes this faster
  - Can use hardware shift registers
  - Easily achieves 10 MHz+ speeds
  - No backwards compatibility
- Designed for short-distance transmission; no hardware flow control or ACK/error checking
  - For inter-board communication, this is often not practical

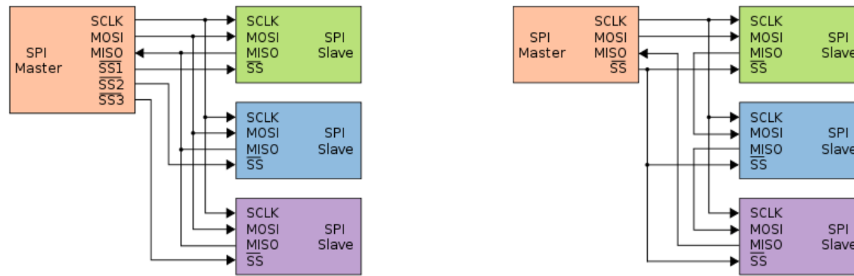


Figure 24: Typical vs. daisy-chained connection for SPI.

- Typically we have one SS line for each device, which is not very pin-efficient
- Some devices support daisy-chained connections
  - Like a ring buffer – when we want to send data to devices beyond the first one in the chain, we clock the data through all intermediate devices
  - This allows us to only use a single SS for all devices
  - This slows down the bus since data needs to go through all intermediate devices
- In a full-duplex implementation, it is expected that we would have hardware shift registers so that MISO and MOSI are simultaneously active
  - SPI expects hardware support; software implementation is rare and often impractical
- Often we need to set 2 SFRs: CPOL (clock polarity: is clock active high or low?) and CPHA (clock phase: is data clocked in on rising or falling edge?)
  - Once set up in hardware, this is usually very easy to use

Control Loops

- How do we set up the fundamental structure of a control loop while making use of as much hardware as we can?
- Our goal is to control a plant, which is possibly affected by a disturbance, and has its output measured by a sensor with added noise
- The controller has a filter which removes the sensor noise or compensates for its transfer function and tries to make the plant output track the reference input

- We need to perform the following tasks:
  1. Collect sensor input
     - We often do this in an interrupt handler
     - If we can calculate the control update quickly, we can do the entire control loop in the ISR
     - However putting the loop in the ISR means we often do not have control over when control updates occur (updates may be irregular)
     - Having the sensor read in the ISR but control loop outside the ISR breaks the timing between the control loop and sensor update, which can be an issue
  2. Store sensor input
     - Sometimes we need to store past inputs, e.g. for an integral controller
     - If only one or two past samples are needed, use extra variables
     - Otherwise we typically use a circular buffer, where the oldest data is replaced by the newest data
  3. Calculate reference signal
     - This depends on the purpose of the system; could come from user commands (from another sensor) or calculated internally
     - We may need to capture the reference signal over time just like the sensor data, with a circular buffer
     - However, the reference and sensor inputs are usually not synchronized
       * If we can guarantee that the input is always evenly spaced and roughly synchronized with feedback, we can often ignore timing differences
         - This only happens if we are polling at regular intervals
         - Interrupts, conditionals, or missed polls will introduce timing differences
         - Even if there are timing differences, many controllers are robust enough to deal with this
       * We can timestamp each input; maintain a system clock and record the time each sample is taken at
         - If no specialized hardware is available for timekeeping, we may need to do instruction counting to keep track of time
         - The degree of accuracy is inherently limited since we need very fast timers; often it's better to try to reduce the misalignment first
  4. Calculate error signal
     - The time that the feedback and reference input were taken can be different, leading to misalignment
     - Calculating the control output and applying the control update also takes time
     - If our control loop updates at a rate that is similar to the reference itself, we need to speed up the system
     - Using interpolation/extrapolation we can align data based on timestamps
       * We can bring everything back to a known past time to align the reference and feedback
       * We can also extrapolate to the time that we are expected to apply the control input
         - The quality of this prediction has a direct impact on the quality of the control
       * Interpolation or extrapolation should generally be avoided in favour of speeding up the loop if possible
  5. Calculate control output
     - To low-pass the input, we can use a (weighted) moving average
       * `Y[i] = K * (E[i] + E[i - 1] + ... + E[i - n]) / n`
       * By changing the number of samples we average, we can change the corner frequency of this filter
     - To high-pass the input, we take the difference between the last two samples and scale it up (i.e. a first order derivative)
       * `Y[i] = K * (E[i] - E[i - 1])`
     - An IIR (infinite impulse response, also known as non-windows low-pass) filter can be implemented with a recursive summation
       * `Y[i] = K * (k * Y[i - 1] + (1 - k) * E[i])`

        * `K` is a proportionality constant and `k` is a weight from 0 to 1 (typically 0.25 to 0.5)
        * This gives us finer control over the corner frequency and does not require a large buffer of past values to average over
        * Using the last value essentially maintains a long memory
    - To calculate integral and derivative terms for a PID controller we can use numerical methods to approximate
        * Integral can be calculated with a moving average between the last and current samples
        * Derivative can be calculated with a backward difference
        * This always introduces some sort of error to the system and possibly leads to instability
        * However PID controllers are often robust enough to work with some minor re-tuning
    - For an arbitrary continuous time transfer function, we can implement it in discrete time on a microcontroller using the Z-transform
6. Send control output to plant
    - This can be done synchronously or asynchronously
    - Synchronous output sending is often accomplished in a polling loop, while asynchronous output is sent on-demand by the plant (e.g. in an ISR)

Lecture 13, Apr 9, 2024

State Machines

- The *state machine* is a model of the dynamic behaviour of a system, represented by:
    - *States*: physical or logical states of the program, e.g. different ranges of variable values, program flow locations, etc that hold a certain meaning
    - *Transitions*: pre-define paths between states, triggered by specific actions or conditions
- State machines are an abstract but functionally equivalent representation of an underlying program or hardware
    - They can be used as a design pattern
    - Can be used as a diagnostic tool for existing code, e.g. automatic state minimization/optimization, consistency checking, etc
- A *finite state machine* (FSM) is a state machine with a finite set of states and allowable transitions between states, possibly with input required for each transition
    - FSMs can have *nondeterminism*, where the same input can lead to multiple different transition
    - We usually assume that our systems are fully deterministic (*deterministic finite automata*, DFAs)
- Each state also has a *payload* or output, i.e. the action taken by the program when that state is reached
- We can specify a state machine by the following:
    - List of states
    - List of outputs for each state (sharing indices with the state list)
    - List of transitions, as tuples of (source state, destination state)
    - List of inputs for each transition (sharing indices with the transitions list)
- *Dead ends* in the state machine are states that have no transitions out
    - These can be problematic
- Implementation of states and transitions can be explicit, but can also be implicitly defined using program flow itself
    - Explicit encoding derives directly from the mathematical representation; easy to modify, difficult to debug manually, has transition search overhead, hard to optimize by compiler
    - Implicit encoding is much more readable by a human; difficult to modify, easy to debug by a program, and has faster transitions
- State machine *minimization* is the process of removing *redundant* states
    - States are redundant if they have the same output and transition to the same set of states given the same input
    - We can search for redundant states, merge them, and then check states again and repeat
    - Given a state machine, we might want to rename/encode the states and write it in a mathematical representation, which makes the redundant states more obvious

    * Eliminate one of the redundant states and change all references to the eliminated state
    * Go through the list of transitions and eliminate duplicate transitions resulting from the change
  - To further simplify the state machine we may introduce variables
    * These variables are essentially state machines of their own
    * We control the transitions in this new state machine via the transitions of the original state machine; when we take transitions, we may choose to modify the variable, i.e. cause a transition in the variable's state machine
  - We can keep moving more of the payload onto transitions; eventually we reach a system with only a single state, with every action being a transition (input conditioned on a large number of variables)
    * This has diminishing returns
  - Since state machines are easy for code to understand, many automated systems exist for state machine minimization
    * These systems are capable of optimizing as much as we want, so we need to specify the correct level of optimization
- Note that state machine minimization does not necessarily get us faster code; the smaller state diagram can lead to faster execution, or less complexity and code size, etc