* Vyšší programovací jazyky pro mcu

# Programování pro MCU x PC

## Jazyk C

* + Přímý přístup k HW
* + Instrukce přímo manipulující s jednotlivými bity: SET, CLEAR, TEST,… (PC řeší softwarově)
* - Omezené hardwarové prostředky (redukována velikost paměti, hloubka implementace zásobníku)
* - Absence operačního systému (absence dynamické alokace paměti, řešení multitaskingu, ukládání kontextu při přerušení)
* - Omezené nebo zakázané rekurzivní volání

## Datové typy

* Většina podporována
* Velikost integeru je závislá na platformě (16/32bit, nemá fixní velikost jazyka C)
* Float a double většinou nejsou podporovány (nutnost softwarové emulace)
* Výkonnější ARMy (Cortex M4) mají hw podporu pro plovoucí desetinnou čárku

## Assembler vs C

* Assembler:
  + Složitejší
  + Vyšší úroveň optimalizace
  + Delší výroba programu
  + Nepřenosný mezi platformami
* C:
  + Rychlejší, čitelnější
  + Přenosný mezi platformami (většinou nejsme závislí na konkrétním hardwaru)

# Požadavky vyšších programovacích jazyků na architekturu MCU

* =Jaké vlastnosti by měl hardware mít, aby byla kódová nadbytečnost co nejmenší
* Více pracovních registrů (akumulátorů; charakteristika architektury RISC)
* Krátký instrukční cyklus (charakteristika architektury RISC)
* Indexování polí (instrukce obsahuje adresu začátku pole a posun vůči začátku)
* Šíření příznaku nuly (instrukce zahrnující výsledek předchozí operace)
* Bitové proměnné (slouží ke zjištění hodnoty na portech => true/false)
* Instrukce včetně operandů uložena v paměti na jedné adrese (není nutno do paměti přistupovat opakovaně)
* Hardwarový stackpointer
* Podpora aritmeticko-logických instrukcí s vyšší šířkou, než je ta nativní (u ATmega64 word – zdvojené registry)

# Optimalizace kompilátoru

* Kompilátor volí, jak bude implementovat kód vyššího programovacího jazyka
* => minimální velikost/co největší rychlost programu
* Kód bychom měli psát taky, aby měl kompilátor s optimalizací co nejméně práce

## Optimalizace závislé na hardwaru

* Podpora ze strany architektury
* Registrové proměnné – klíčové slovo, kompilátor se pokusí proměnnou uložit do registrů místo do RAM
* Optimalizace jednoduchým přístupem – pokud je to možné, tak se kompilátor snaží využít bitové operace
* Změna typu/směru smyčky – pro procesory je jednodušší testovat na nulu => vzestupný for změněn na sestupný

## Optimalizace nezávislé na hardwaru

* Aplikovatelné vždy
* Zpracování konstant
  + Výpočty obsahující konstanty jsou předpočítány v době kompilace
* Vyloučení opakovaných výpočtů (výrazů)
  + Uložení do registru (tuto optimalizaci ruší klíčové slovo volatile, které říká, že se hodnota proměnné může změnit)
* Optimalizace skokových příkazů
  + Vícenásobné skoky (vnořené ify) možno nahradit přímým skokem
  + Volba absolutního/relativního skoku
* Vyloučení mrtvého kódu (většinou za return nebo break)
* Náhrada opakujících se úseků programu skoky
* Negace skoků
  + Znegování podmínky
* Optimalizace plnění
  + Nadbytečné příkazy plnění se odstraňují
  + Například deklarace proměnné, do které se posléze uloží výsledek výpočtu
* Překrývání dat
  + Sdílení statických proměnných v rámci několika funkcí (pokud to neovlivní správný průběh programu)
* Optimalizace jednoduchých cyklů
  + Místo smyčky „nakopírování“ kódu několikrát za sebou
  + Zabere více paměti x zrychlí průběh programu
* Rotace smyček
  + Zaměnění pořadí provádění instrukcí, pokud na sobě nejsou závislé
* Optimalizace řídícího toku
  + Náhrada za switch-case
  + Switch-case se převádí na if
  + Lze „skočit“ přímo na správný case a netestovat předchozí