# Pohled překladače na procesor

# Scheduling - volba uspořádání instrukcí

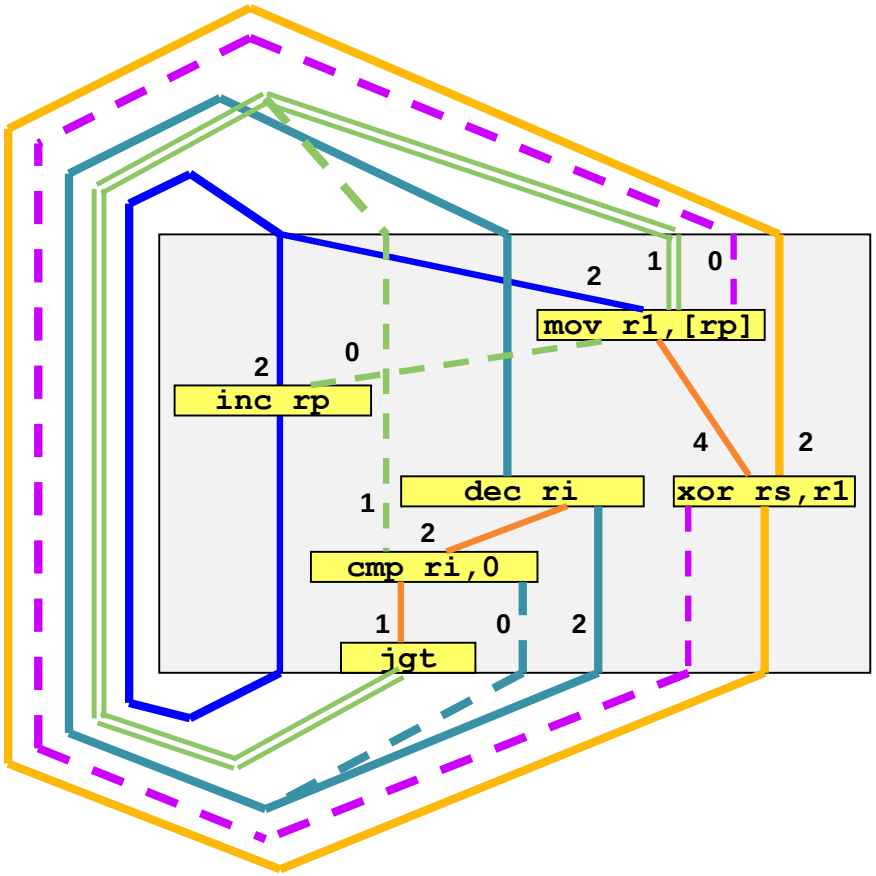
- Dříve nejpodstatnější fáze překladače z hlediska výkonu kódu
  - Dnes má podobný význam vektorizace
- Out-of-order execution v CPU řeší tentýž problém
  - Má více informací ale méně času
- Hledá se takové pořadí které je
  - Ekvivalentní z hlediska efektu/pravidel jazyka
    - Vyhovuje závislostem
  - Nejrychlejší
    - Model časování procesoru

- Speciální metody pro jednoduché cykly – Software pipelining
  - Základní blok zakončený podmíněným skokem na svůj začátek
  - Scheduling optimalizuje chování v cyklu
    - Zvládá přesouvání instrukcí přes hranice iterací
    - Loop unrolling – tělo cyklu se duplikuje pro odstranění některých závislostí
  - Základní principy je možné aplikovat i ručně
    - Pro lepší porozumění problému v kritických místech kódu
    - Pro ruční optimalizace tam, kde překladač selhal

# Závislosti

- Závislost (dependence)
  - Povinnost provést jednu operaci/instrukci po jiné
  - Částečné uspořádání operací/instrukcí v jednom BB
- Datová závislost (dependence)
  - Závislost producent-konzument v toku dat
- Antidependence
  - Read-Write: Čtení se musí stihnout před zápisem
  - Write-Write: Pořadí zápisů se nesmí změnit
  - Control-dependence: Operaci lze provést až po vyhodnocení podmínky
    - Týká se operací s efektem, který nelze odčinit (zápisy do paměti, chyby,...)
  - Jiné důvody, obvykle nízkoúrovňového původu

# Příklad




```
char chksum(
  char * rp, int ri)
{
  char rs = 0;
  while ( ri > 0 )
  {
    char r1 = *rp++;
    rs ^= r1;
    --ri;
  }
  return rs;
}
```

- Datové závislosti
- Antidependence
  - důsledek opakovaného používání proměnných
- Control dependence
  - jgt -> mov

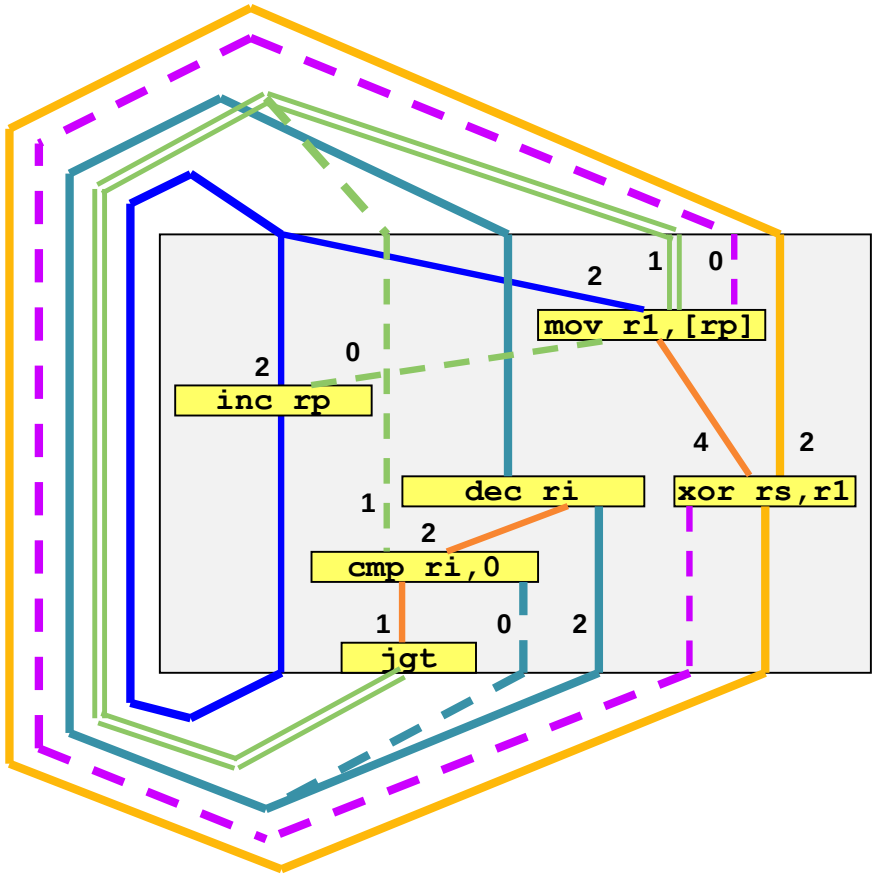
## Model procesoru

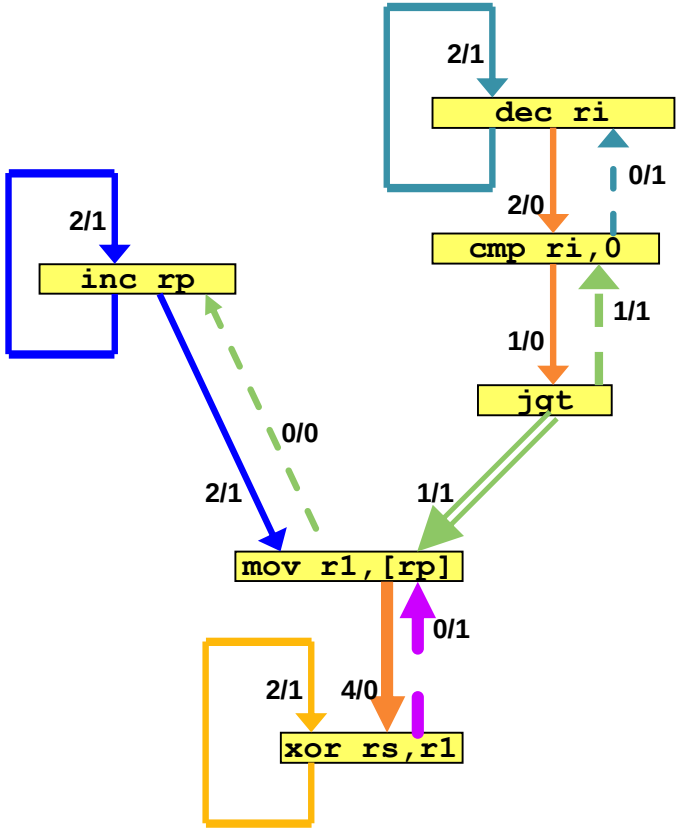
- Latence – časování závislých dvojic instrukcí
  - Počet cyklů procesoru, který musí proběhnout mezi referenčními body závislých instrukcí
  - U antidependencí a ve speciálních případech může být nulová
- Rezervační tabulky – schopnosti paralelního zpracování
  - Procesor je rozdělen na funkční jednotky různých druhů
  - Je určen počet jednotek každého druhu
  - Pro každou instrukci definována rezervační tabulka
    - Počet jednotek daného druhu, který instrukce potřebuje v daném čase
  - Velmi primitivní model ve srovnání s dnešními CPU
    - Přesto většinou řídí scheduling správným směrem

## Scheduling pouze odhaduje skutečné časování

- Skutečné časování je ovlivněno nepředvídatelnými jevy
  - Zbytky rozpracovaných instrukcí z předchozího kódu
    - Řešení: Trace-scheduling, řízení profilem
  - Paměťová hierarchie
    - Doba přístupu k paměti závisí na přítomnosti v cache
    - Obvykle se předpokládá nejlepší možný průběh
    - Speciální problém: Multithreaded aplikace na multiprocesorech
  - Fetch bandwidth
    - Instrukce nemusí být načteny a dekódovány včas
    - Zdržují skoky a soupeření o přístup do paměti
    - Přesné simulování fetch jednotky by neúměrně komplikovalo scheduler
- Scheduler nezná skutečné závislosti přístupů do paměti
  - Musí postupovat opatrně a zohledňuje i nejisté závislosti
  - Procesor zná skutečné adresy přístupů a detekuje pouze skutečné závislosti
    - Procesor dokáže v některých případech udělat roll-back – spekulativní provádění
    - Agresivně optimalizující procesor může zvolit zcela jiné pořadí instrukcí

# Přehlednější abstrakce téhož problému



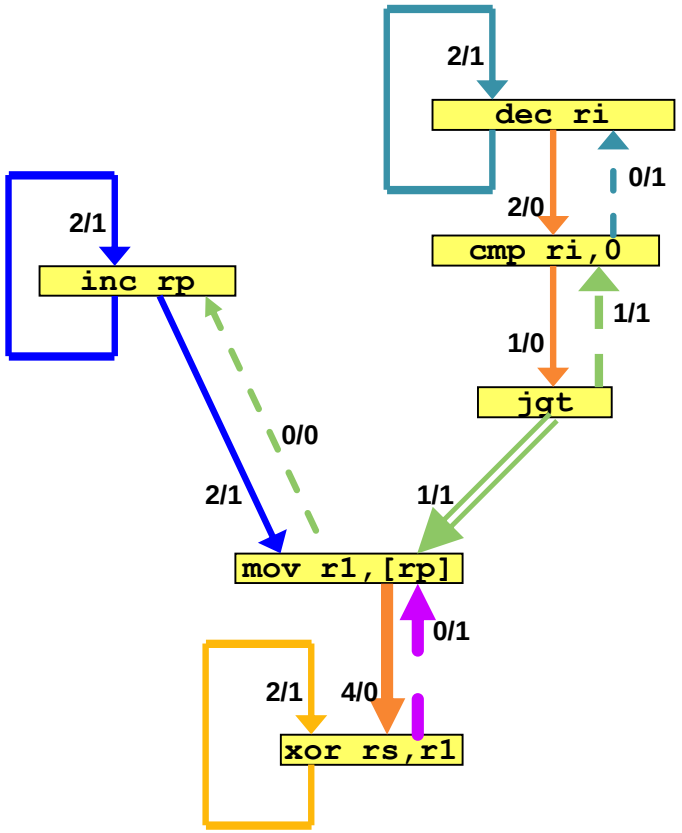




- Číslo před lomítkem = latence
- Číslo za lomítkem = počet překročení hranice iterací

- Kritická smyčka
  - Největší podíl součtu latencí a součtu iterací

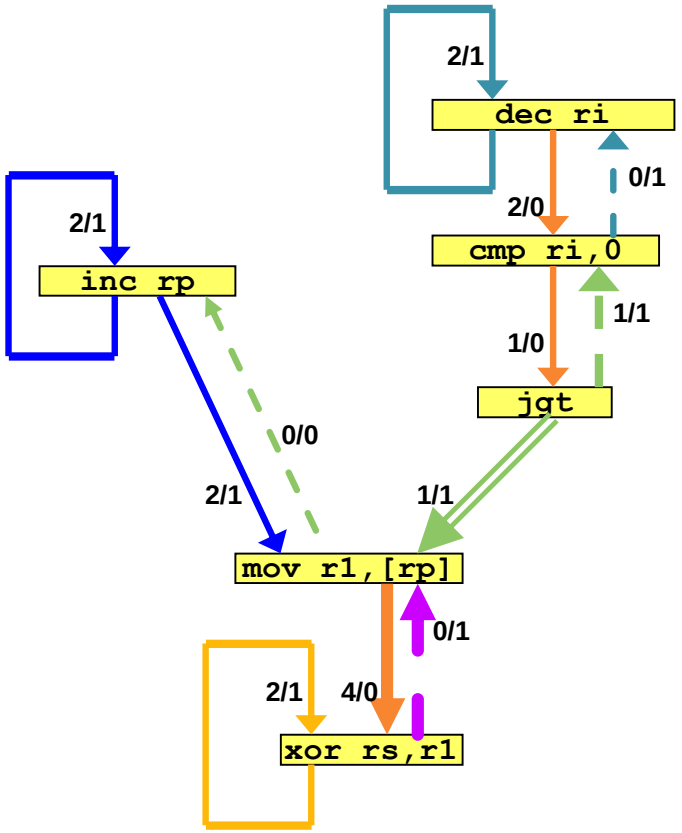
`[4/0]+[0/1] = (4+0)/(0+1) = 4/1`

  - Určuje maximální možnou výkonnost cyklu
    - 4 takty CPU na 1 iteraci
  - Zohledňuje pouze latence
    - Neřeší kapacitu procesoru



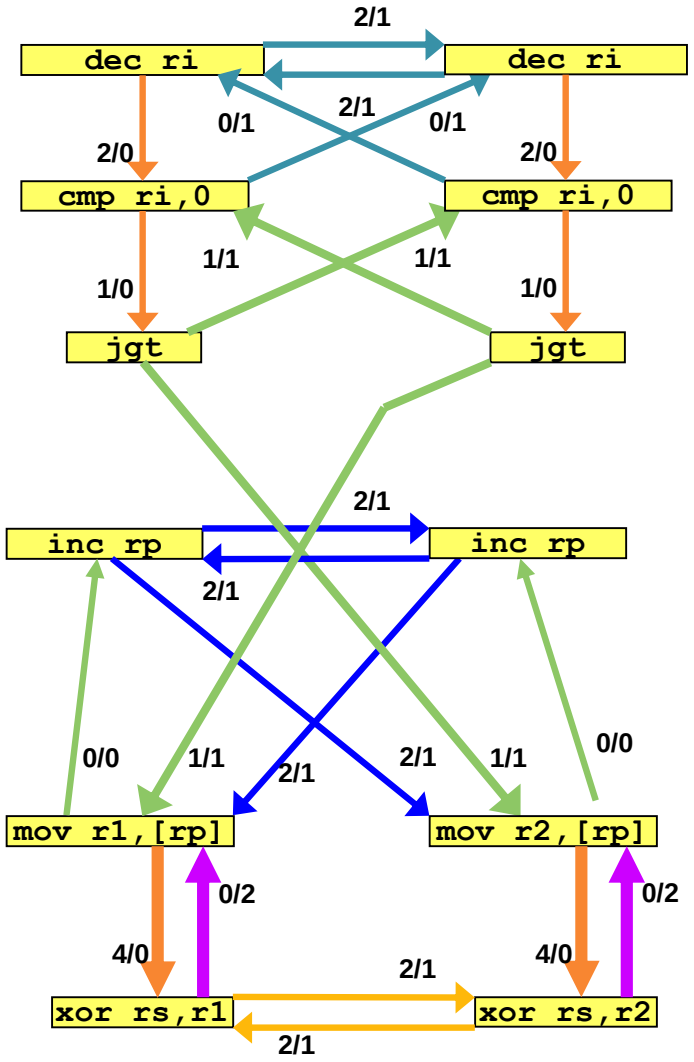
# Duplikace kódu a proměnných

- Duplikace proměnných
  - Antidependence lze odstranit
    - U jednoduchých proměnných
  - Střídání použité proměnné
    - Vyžaduje duplikaci kódu
    - Vede k vyšším nárokům na počet registrů
      - Mnohonásobná multiplikace nebývá efektivní
  - Duplikaci může provést i programátor
    - V případech, kdy překladač neumí nebo odmítá provést duplikaci
    - Typicky u složitějších datových typů, kde překladač neodhalí správně aliasy
  - Duplikace proměnných fakticky dělá totéž, co renaming v OOO procesorech
    - Překladač nedokáže CPU renaming vynutit, nahrazuje se duplikací kódu

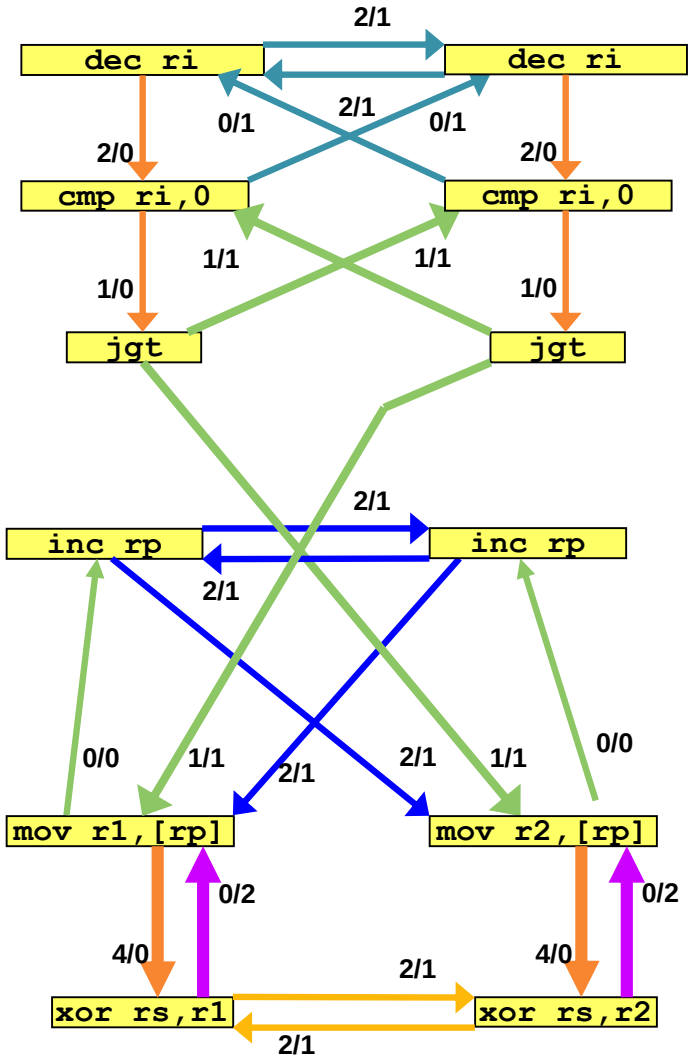


# Duplikace kódu a proměnných

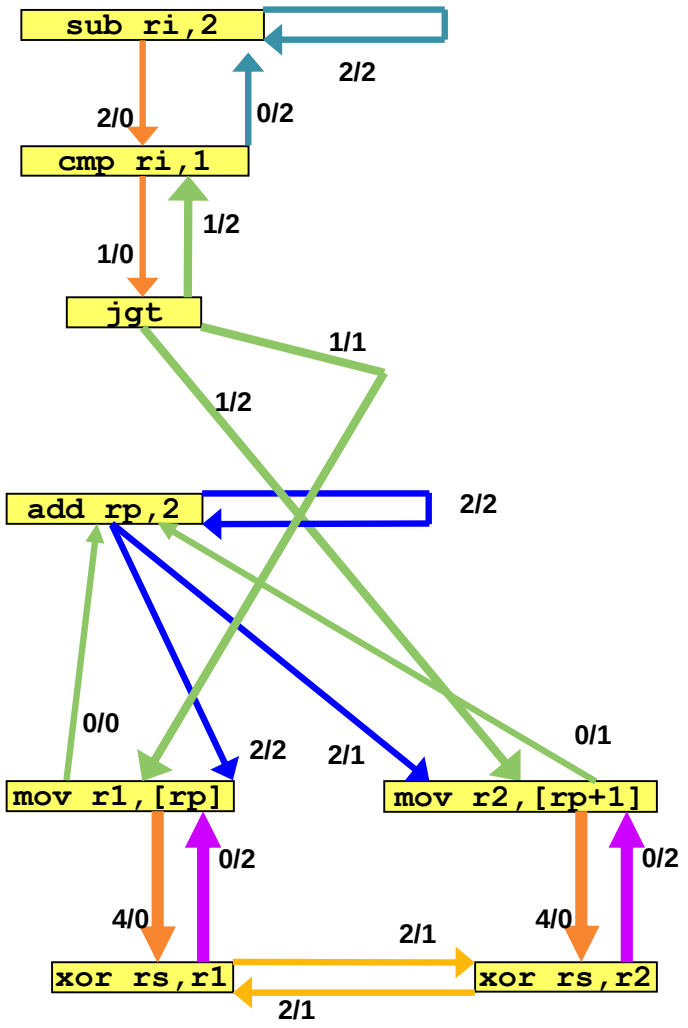
- Duplikace proměnných
  - Antidependence lze odstranit
    - U jednoduchých proměnných
  - Střídání použité proměnné
    - Vyžaduje duplikaci kódu
    - Vede k vyšším nárokům na počet registrů
      - Mnohonásobná multiplikace nebývá efektivní
  - Duplikaci může provést i programátor
    - V případech, kdy překladač neumí nebo odmítá provést duplikaci
    - Typicky u složitějších datových typů, kde překladač neodhalí správně aliasy
  - Duplikace proměnných fakticky dělá totéž, co renaming v OOO procesorech
    - Překladač nedokáže CPU renaming vynutit, nahrazuje se duplikací kódu



- Algebraické triky
  - Obvykle spojené s duplikací kódu
  - Instrukce příčítající konstantu
    - Náhrada dvou přičtení příčtením dvojnásobku
    - Ušetří instrukci
    - Zmenší kritičnost cyklu
  - Lineární podmínky
    - Jedním porovnáním lze vyřešit dvě původní podmínky
      - Konjunkce dvou po sobě jdoucích podmínek while-cyklu
      - Po ukončené upraveného cyklu je nutno otestovat, která z původních podmínek způsobila ukončení
    - Ušetří instrukce a kritičnost
    - Vrátí duplikovaný cyklus do podoby jednoduchého cyklu



- Algebraické triky
  - Obvykle spojené s duplikací kódu
  - Instrukce příčítající konstantu
    - Náhrada dvou přičtení příčtením dvojnásobku
    - Ušetří instrukci
    - Zmenší kritičnost cyklu
  - Lineární podmínky
    - Jedním porovnáním lze vyřešit dvě původní podmínky
      - Konjunkce dvou po sobě jdoucích podmínek while-cyklu
      - Po ukončené upraveného cyklu je nutno otestovat, která z původních podmínek způsobila ukončení
    - Ušetří instrukce a kritičnost
    - Vrátí duplikovaný cyklus do podoby jednoduchého cyklu



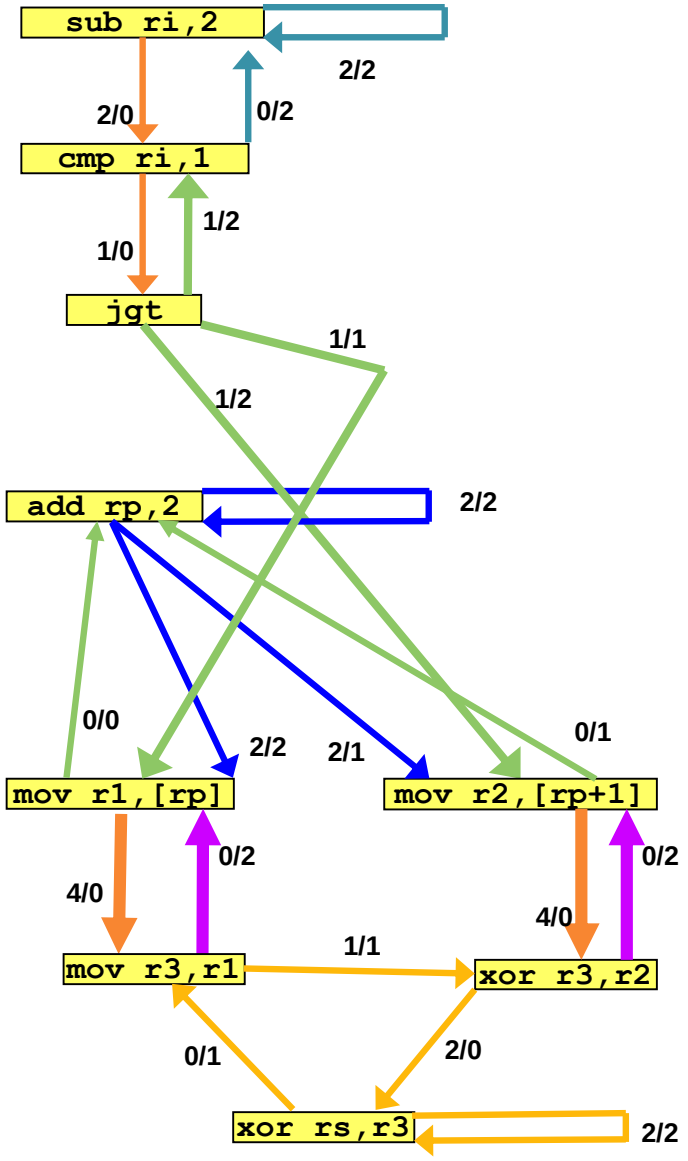
# Příklad – Intel compiler – x64

```
char chksum( char * p, int i)
{
  char s = 0;
  while ( i > 0 )
  {
    s ^= *p++;
    --i;
  }
  return s;
}

..B1.4:
        movsbq    (%rdi), %r8
        movsbq    1(%rdi), %r9
        xorl      %r8d, %eax
        xorl      %r9d, %eax
        addq      $2, %rdi
        addl      $1, %ecx
        cmpl      %edx, %ecx
        jb        ..B1.4
```

```
/*...*/

k = i >> 1;
j = 0;

do {
  r8 = *p;
  r9 = *(p+1);
  s ^= r8;
  s ^= r9;
  p += 2;
  j += 1;
} while ( j < k );

/* ... */
```

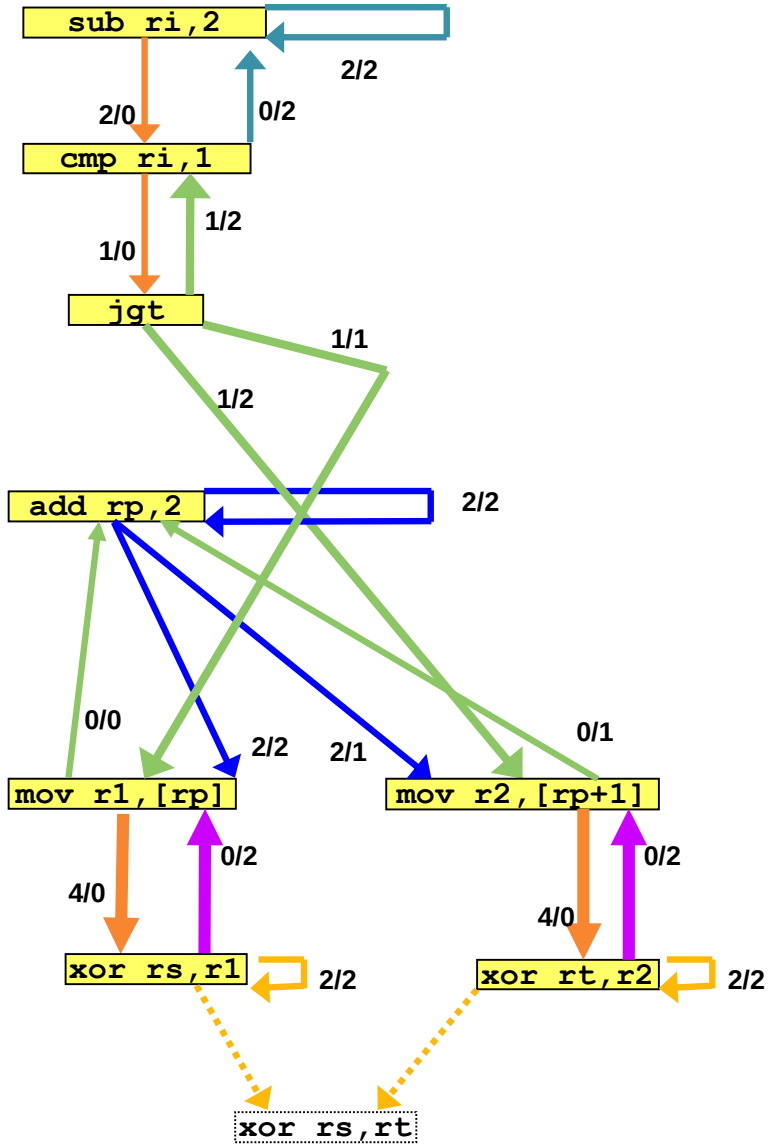
# Reasociace asociativních operací

- Algebraické triky
  - Reasociace operací
    - Vyžaduje asociativitu, někdy i komutativitu
    - Počet instrukcí zachován, někdy přidány přesuny
    - Zmenší kritičnost cyklu
  - Velmi často nutné pro vektorizaci



# Reasociace asociativních a komutativních operací

- Algebraické triky
  - Reasociace operací
    - Vyžaduje asociativitu, někdy i komutativitu
    - Počet instrukcí zachován, někdy přidány přesuny
    - Zmenší kritičnost cyklu
  - Velmi často nutné pro vektorizaci



# Scheduling - shrnutí

- Překladače obykle provádějí tyto fáze
  - Reasociace operací – obvykle na abstraktním mezikódu
  - Analýza závislostí
  - Speciální řešení jednoduchých smyček
    - Analýza podmínky a její případná transformace
    - Analýza kritické smyčky
    - Snížení kritičnosti duplikací kódu a proměnných
    - Slučování operací s konstantami
    - Software pipelining – vlastní scheduling smyčky
    - Doplnění prologu/epilogu pro nástup a výstup
  - Trace scheduling – scheduling zbytku procedury
- Pro programátora může mít smysl provést některé úpravy ručně
  - Tam, kde je situace pro překladač příliš nepřehledná
  - Provádět samotný scheduling ručně však nedává smysl