Software pipelining

- Speciální metody schedulingu pro jednoduchý cyklus
  - Základní blok zakončený podmíněným skokem na svůj začátek
- Unroll-and-compact
  - Téměř běžný scheduling aplikovaný na rozvíjející se cyklus
- Modulo scheduling
  - Rozvrhování modulo N – cílový počet taktů procesoru
- Obě metody vyžadují detekci závislostí a latencí přes hranice iterací

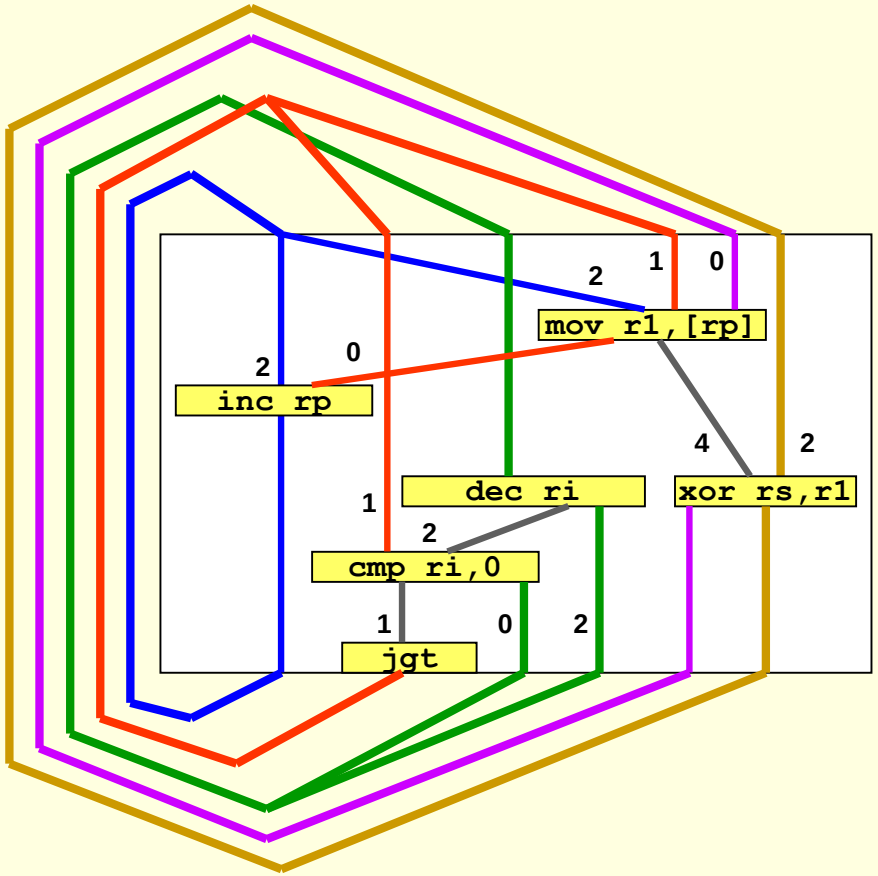
Příklad – software pipelining

- Závislosti uvnitř BB

z instrukce	do instrukce	čas
`cmp ri,0`	`jgt`	1
`mov r1,[rp]`	`xor rs,r1`	4
`dec ri`	`cmp ri,c`	2
`mov r1,[rp]`	`inc rp`	0

- Závislosti přes hranice BB (loop-carried dependences)

z instrukce	do instrukce	čas
`inc rp`	`inc rp`	2
`inc rp`	`mov r1,[rp]`	2
`dec ri`	`dec ri`	2
`cmp ri,0`	`dec ri`	0
`xor rs,r1`	`xor rs,r1`	2
`xor rs,r1`	`mov r1,[rp]`	0
`jgt`	`cmp ri,0`	1
`jgt`	`mov r1,[rp]`	1

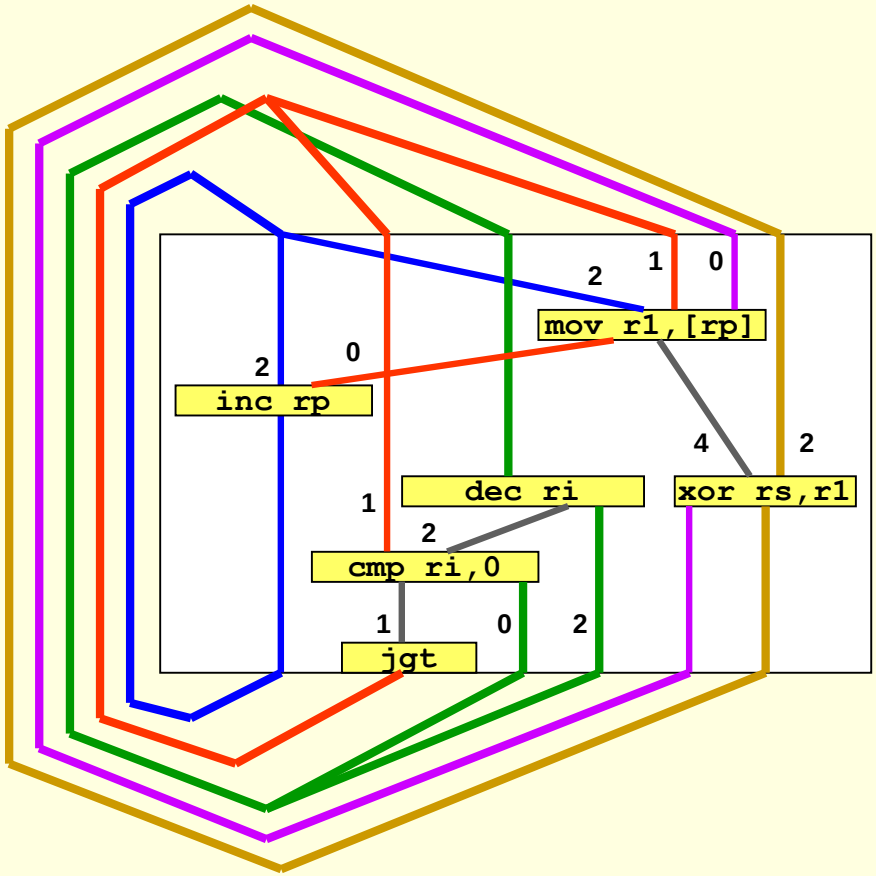


Příklad – unroll-and-compact

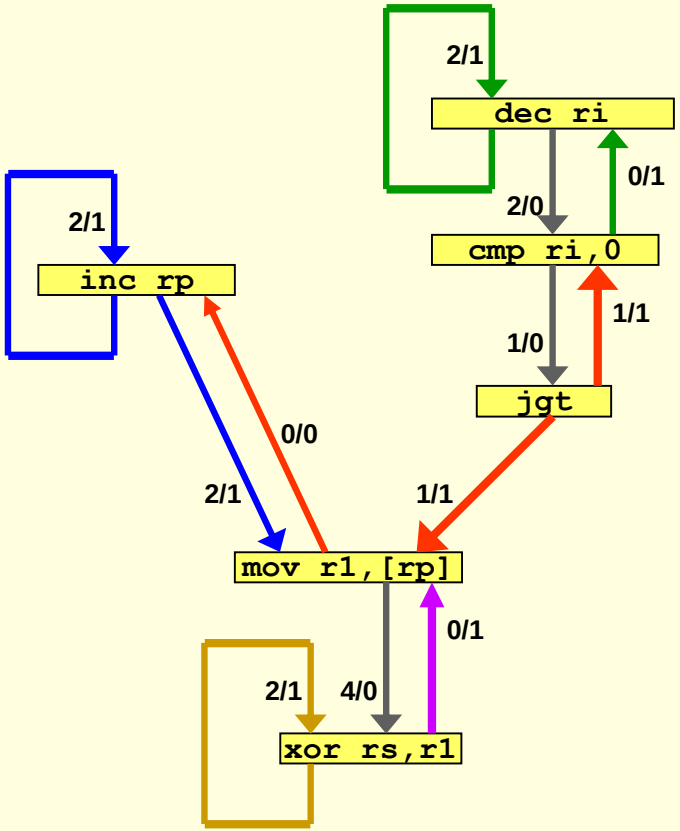
- **Software pipelining**
  - "Unroll-and-compact"
    - Rozvrhuje se rozvinutý cyklus tak dlouho, dokud nevznikne opakující se vzorek
      - Perioda opakování může odpovídat více iteracím
    - Problém: Jak určit prioritu při rozvrhování
      - Rozvinovaný cyklus nemá konec – není kritická cesta
      - Řeší se asymptotické chování - kritická smyčka

čas	R	MEM	ALU	W
0	mov1, dec1			
1	inc1	mov1	dec1	
2	cmp1	mov1	inc1	dec1
3	dec2	mov1	cmp1, jgt1	inc1
4	xor1, mov2		dec2	mov1, cmp1
5	inc2, cmp2	mov2	xor1	dec2
6	dec3	mov2	inc2, cmp2	xor1
7		mov2	dec3, jgt2	inc2, cmp2
8	xor2, mov3			mov2, dec3
9	inc3, cmp3	mov3	xor2	
10	dec4	mov3	inc3, cmp3	xor2
11		mov3	dec4, jgt3	inc3, cmp3
12	xor3, mov4			mov3, dec4
13	inc4, cmp4	mov4	xor3	
14	dec5	mov4	inc4, cmp4	xor3
15		mov4	dec5, jgt4	inc4, cmp4
16	xor4, mov5			mov4, dec5
17	inc5, cmp5	mov5	xor4	
18	dec6	mov5	inc5, cmp5	xor4
19		mov5	dec6, jgt5	inc5, cmp5
20	xor5			mov5, dec6
21			xor5	
22				xor5

Příklad – software pipelining



- **Jiná abstrakce**
  - latence/iterace

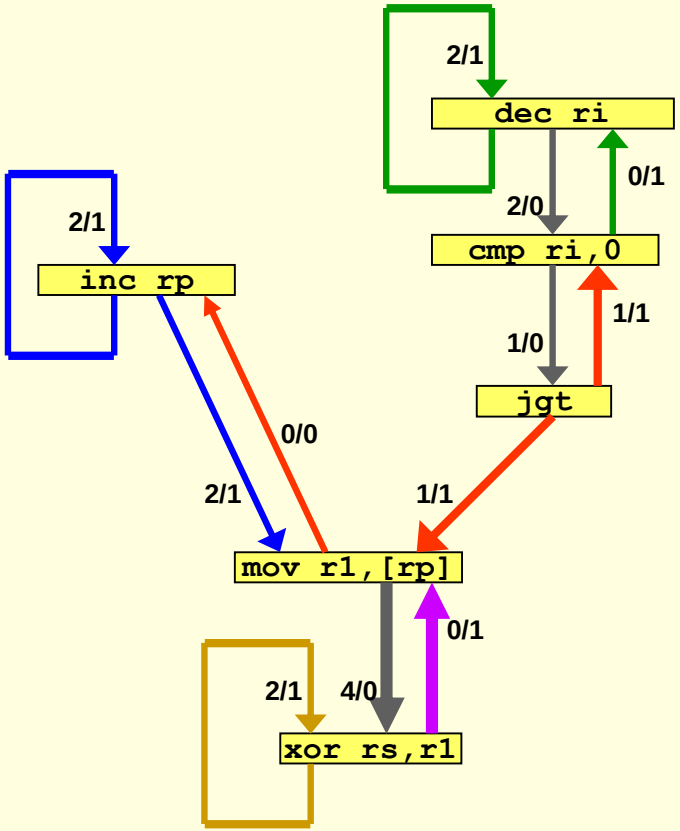


Software pipelining

- Kritická smyčka
  - Cyklus v orientovaném grafu závislostí s největším podílem součtu latencí a součtu iterací

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

  - Limituje asymptotické chování jakéhokoliv rozvrhu

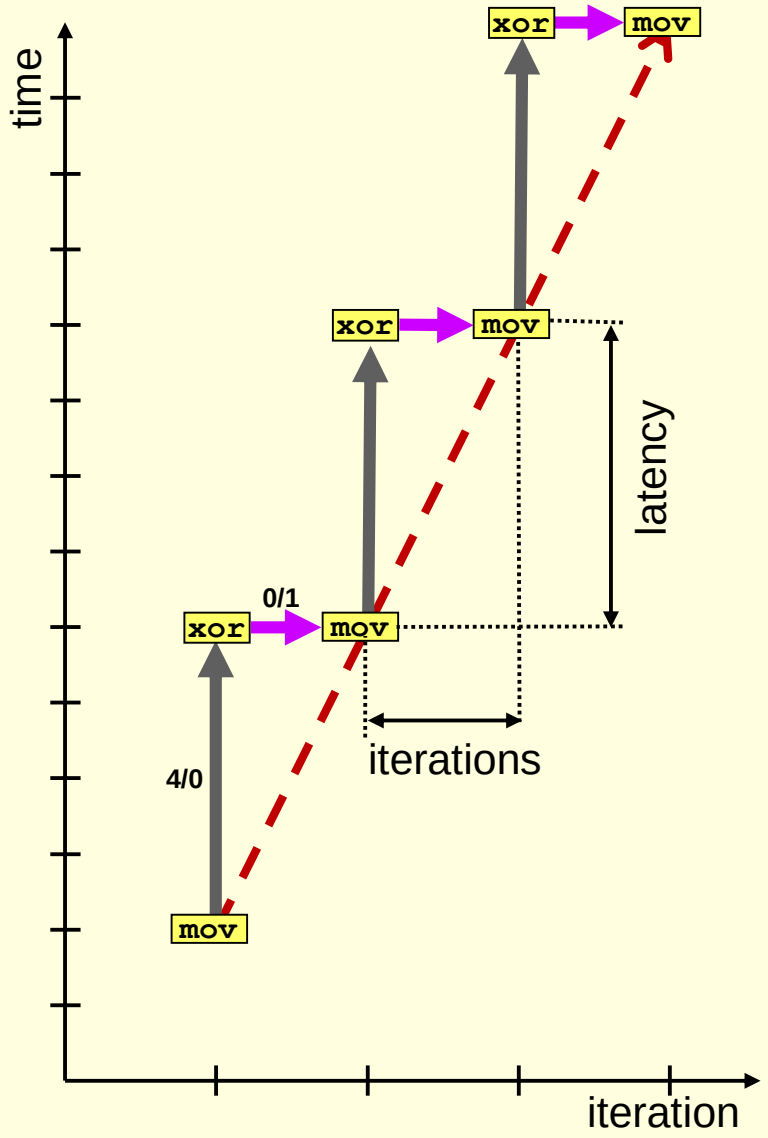


Software pipelining

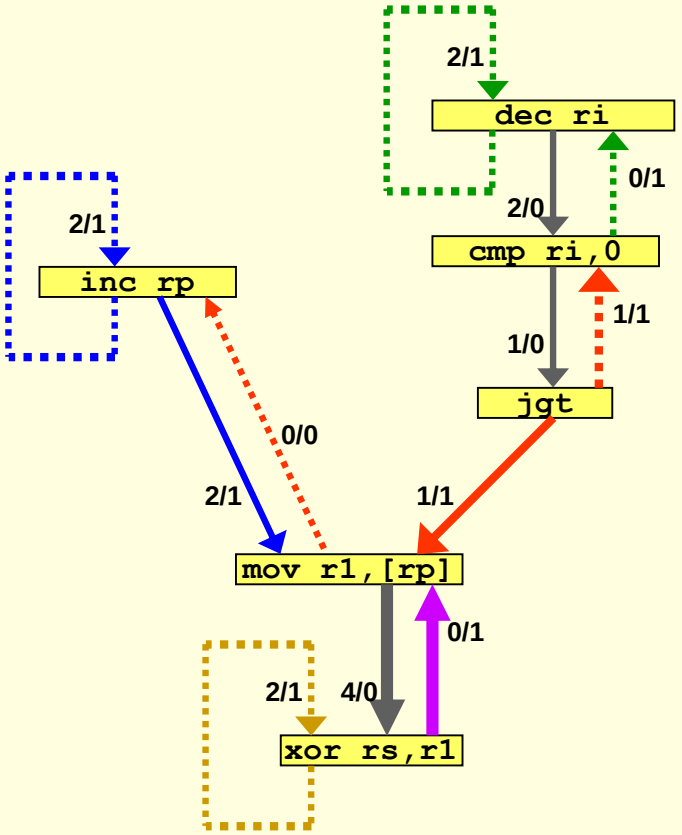
- Kritická smyčka
  - Cyklus v orientovaném grafu závislostí s největším podílem součtu latencí a součtu iterací

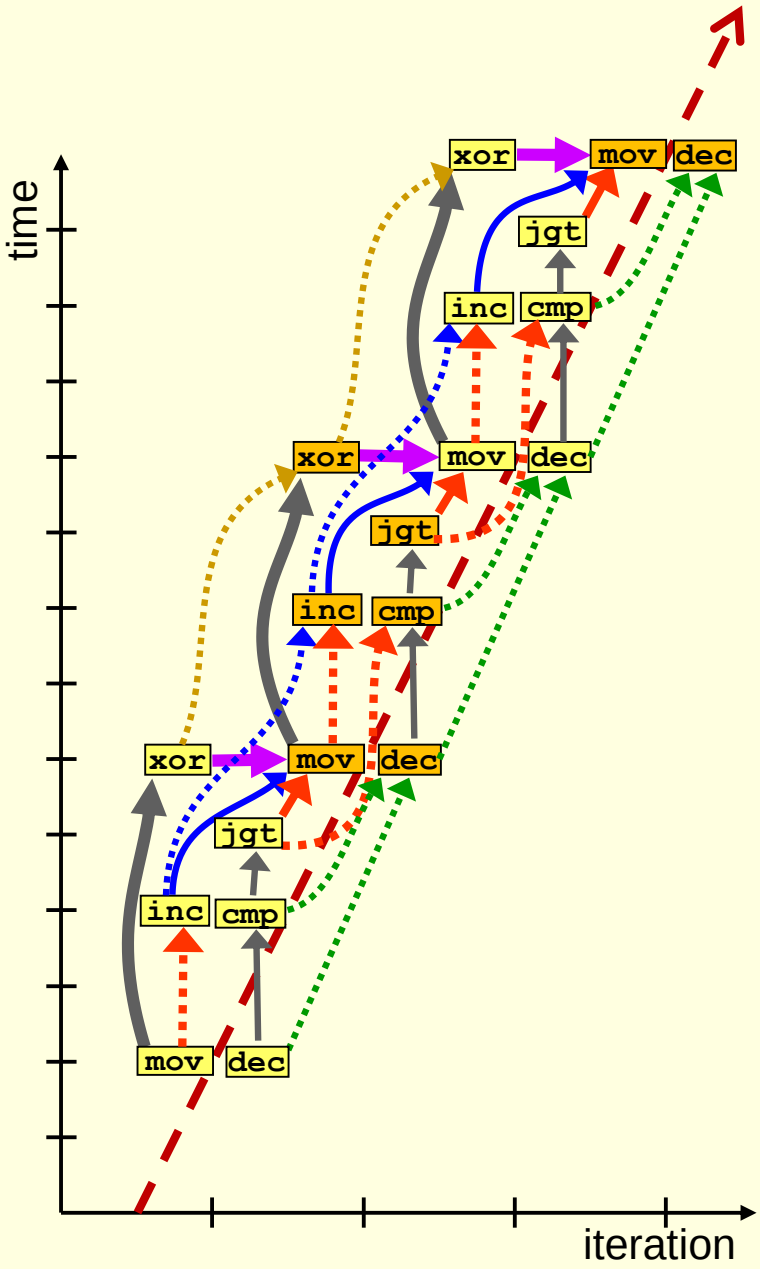
`[4/0]+[0/1] = (4+0)/(0+1) = 4/1`

  - Limituje asymptotické chování jakéhokoliv rozvrhu
  - Ostatní části rozvrhu se připojují před nebo za kritickou smyčku



Unroll-and-compact





- Instrukce jsou v grafu umístěny na poslední čas umožňující dodržení rozvrhu kritické smyčky
  - Kapacita procesoru se neřeší
- Rozhodují hrany směrem ke kritické smyčce
  - Ostatní budou automaticky dodrženy

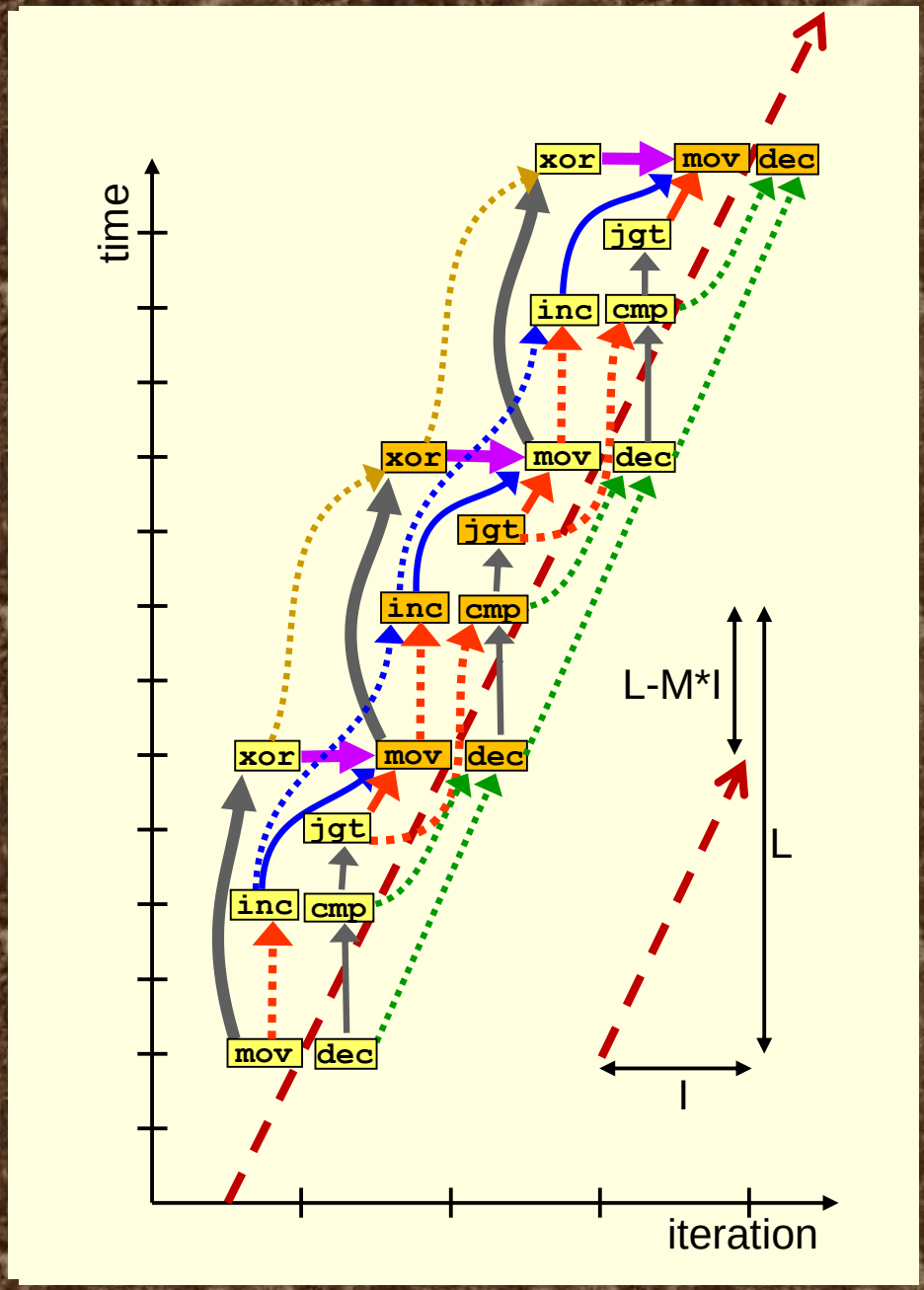
Unroll-and-compact

- Graf pouze vysvětluje princip, ve skutečnosti stačí spočítat součty latencí a iterací na nejvýznamnější cestě ke kritické smyčce
  - Významnost se měří vzorcem

  L-M*I
  - L,I – součty latencí, iterací cesty
  - M – podíl latencí a iterací kritické smyčky
  - Významnost měří vzdálenost v grafu od přímky kritické smyčky

- Priorita instrukce
  - Vyjadřuje nejmenší možné vertikální posunutí přímky kritické smyčky, pokud bude instrukce rozvržena na čas, ve kterém je připravena
  - Součet významnosti cesty od instrukce ke kritické smyčce a vzorce

  T-M*I
  - T,I – čas připravenosti a iterace instrukce
  - Je-li kritických smyček více, počítá se maximum

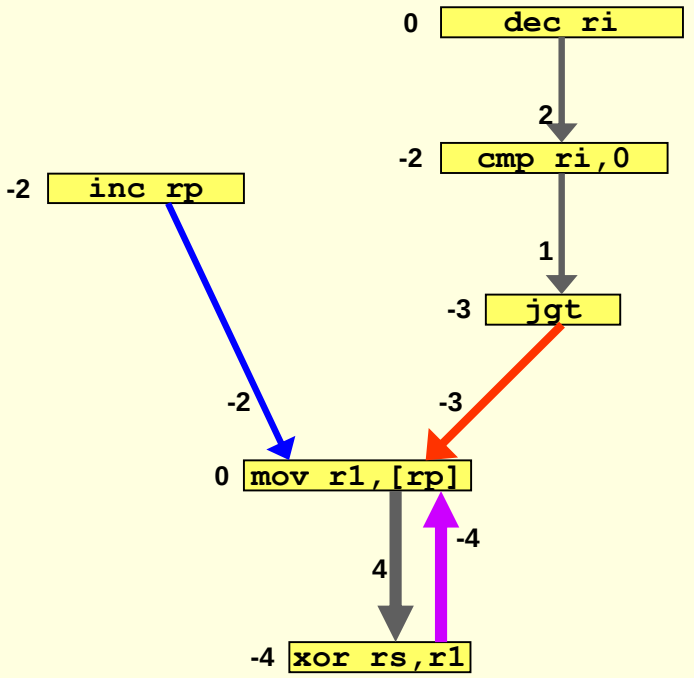


Unroll-and-compact

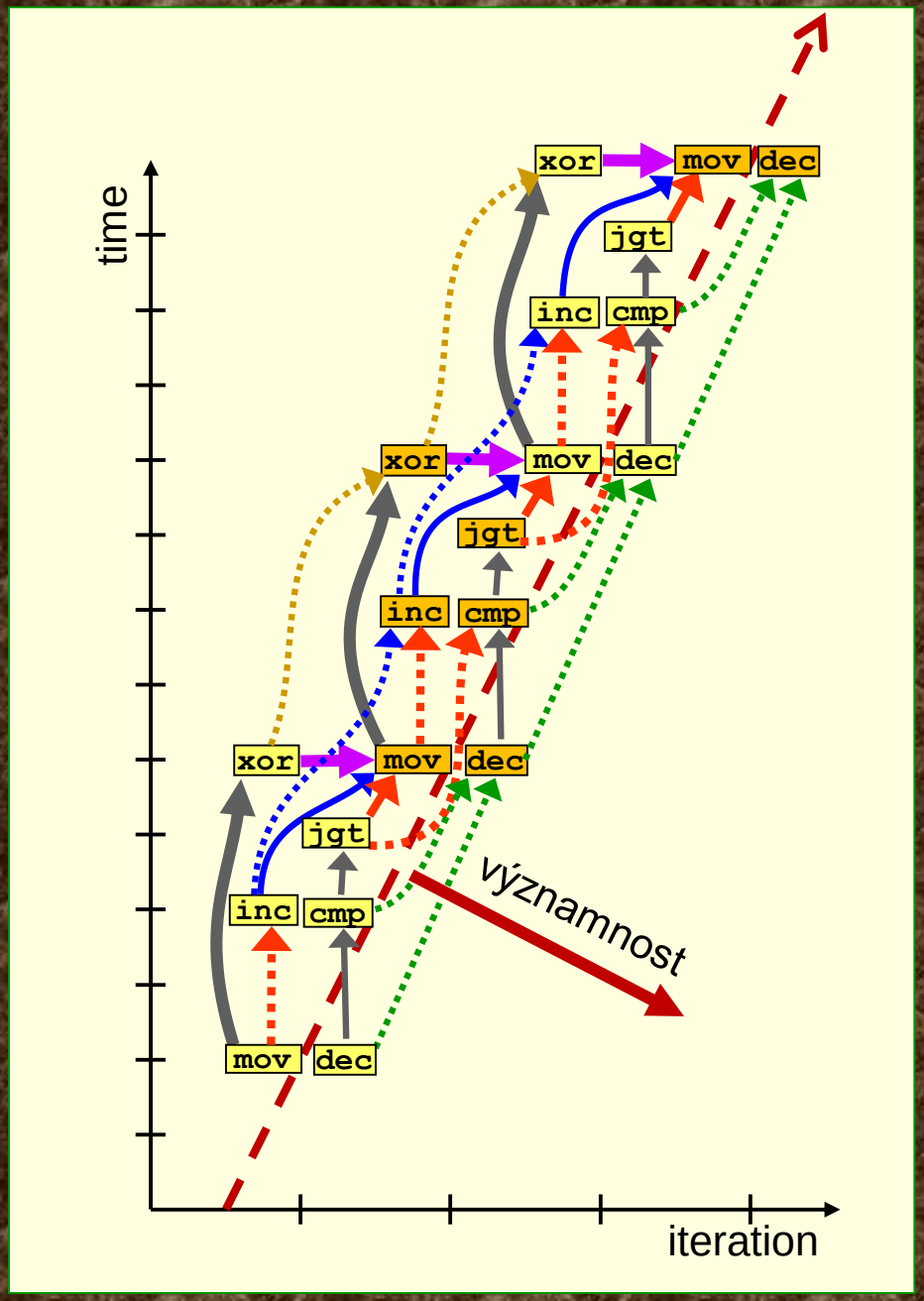
- Graf pouze vysvětluje princip, ve skutečnosti stačí spočítat součty latencí a iterací na nejvýznamnější cestě ke kritické smyčce
  - Významnost se měří vzorcem

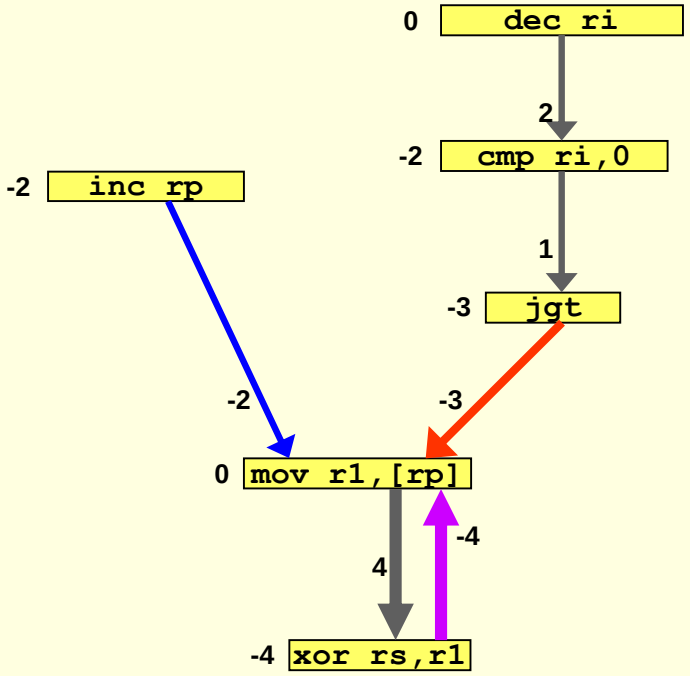
  L-M*I
  - L,I – součty latencí, iterací cesty
  - M – podíl latencí a iterací kritické smyčky
  - Významnost měří vzdálenost v grafu od přímky kritické smyčky
- Priorita instrukce
  - Vyjadřuje nejmenší možné vertikální posunutí přímky kritické smyčky, pokud bude instrukce rozvržena na čas, ve kterém je připravena
  - Součet významnosti cesty od instrukce ke kritické smyčce a vzorce

  T-M*I
  - T,I – čas připravenosti a iterace instrukce

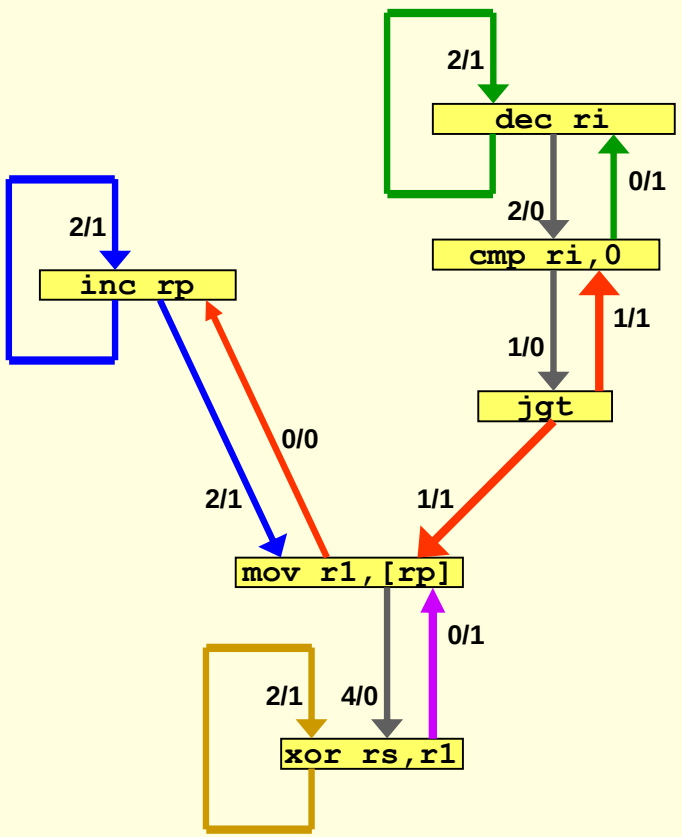


Unroll-and-compact





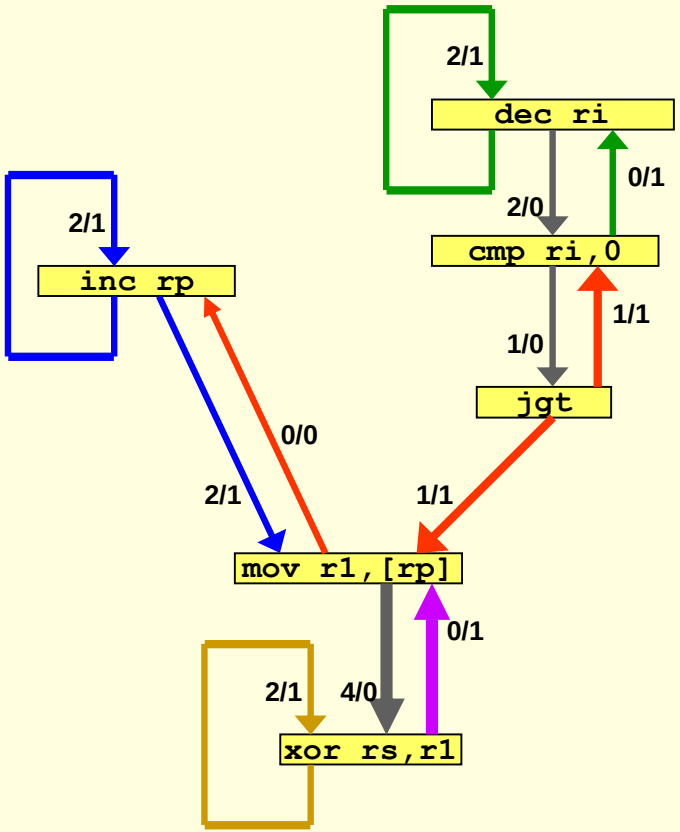
Příklad – unroll-and-compact



čas	R	MEM	ALU	W
0	mov1, dec1			
1	inc1	mov1	dec1	
2	cmp1	mov1	inc1	dec1
3	dec2	mov1	cmp1, jgt1	inc1
4	xor1, mov2		dec2	mov1, cmp1
5	inc2, cmp2	mov2	xor1	dec2
6	dec3	mov2	inc2, cmp2	xor1
7		mov2	dec3, jgt2	inc2, cmp2
8	xor2, mov3			mov2, dec3
9	inc3, cmp3	mov3	xor2	
10	dec4	mov3	inc3, cmp3	xor2
11		mov3	dec4, jgt3	inc3, cmp3
12	xor3, mov4			mov3, dec4
13	inc4, cmp4	mov4	xor3	
14	dec5	mov4	inc4, cmp4	xor3
15		mov4	dec5, jgt4	inc4, cmp4
16	xor4, mov5			mov4, dec5
17	inc5, cmp5	mov5	xor4	
18	dec6	mov5	inc5, cmp5	xor4
19		mov5	dec6, jgt5	inc5, cmp5
20	xor5			mov5, dec6
21			xor5	
22				xor5

Software pipelining

- „Modulo scheduling“
  - Analýzou grafu závislostí se odhadne počet cyklů na iteraci
    - Sčítají se ohodnocení hran ve smyčkách dvojicemi latence/iterace
    - Rozhoduje nejvyšší podíl součtů na smyčce, např. 4/1
  - Hledá se rozvrh s daným počtem cyklů na iteraci
    - Nemusí existovat – opakuje se pro větší počet
  - Složitější verze zvládají necelé počty cyklů na iteraci
    - Pokud je v grafu závislostí kritická smyčka se součtem ohodnocení např. 5/2

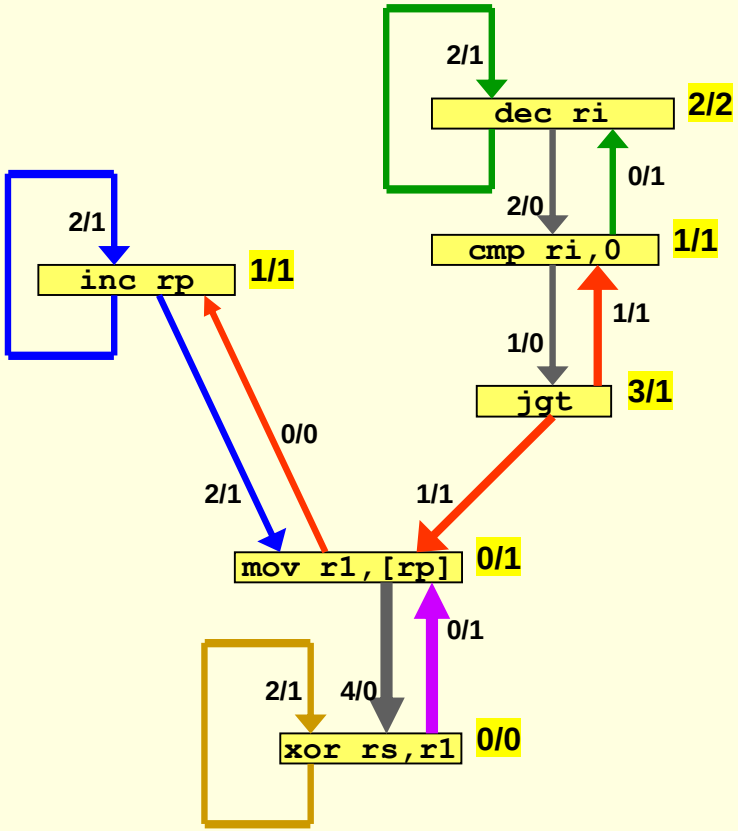


Modulo scheduling

čas	R	MEM	ALU	W
T+0	xor[N], mov[N+1]			mov[N], dec[N+1]
T+1	inc[N+1], cmp[N+1]	mov[N+1]	xor[N]	
T+2	dec[N+2]	mov[N+1]	inc[N+1], cmp[N+1]	xor[N]
T+3		mov[N+1]	dec[N+2], jgt[N+1]	inc[N+1], cmp[N+1]

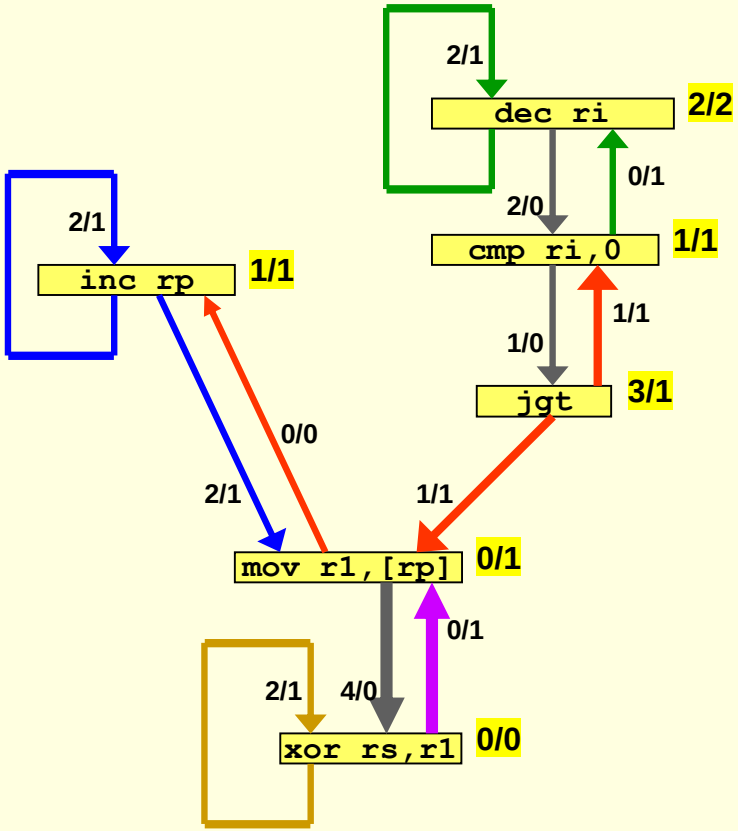
- Zvolená perioda M
  - Rezervační tabulky jsou používány modulo M: časy 0..(M-1)
  - Instrukce jsou rozmisťovány do dvojrozměrného prostoru čas/iterace
  - Rozvrh vyhovuje závislosti

pokud
$(T_B-T_A)-M*(I_B-I_A) \geq L-M*D$



Modulo scheduling

čas	R	MEM	ALU	W
0	xor2, mov3			mov2, dec3
1	inc3, cmp3	mov3	xor2	
2	dec4	mov3	inc3, cmp3	xor2
3		mov3	dec4, jgt3	inc3, cmp3
4	xor3, mov4			mov3, dec4
5	inc4, cmp4	mov4	xor3	
6	dec5	mov4	inc4, cmp4	xor3
7		mov4	dec5, jgt4	inc4, cmp4



Příklad – software pipelining

Výsledek pro příklad

- Výkon: 1/4 iterace/cyklus
  - Zlepšení o 25%
- Využití jednotek:
  - R: 5/8
  - MEM: 3/4
  - ALU: 5/8
  - W: 5/8
- Poslední opakování vzorku provede zbytečně instrukci dec
  - To není chyba

Vytvoření kódu z rozvrhu

- Prolog-smyčka-epilog
- Dokončení pro odbočky

čas	R	MEM	ALU	W
0	mov1, dec1			
1	inc1	mov1	dec1	
2	cmp1	mov1	inc1	dec1
3	dec2	mov1	cmp1, jgt1	inc1
4	xor1, mov2		dec2	mov1, cmp1
5	inc2, cmp2	mov2	xor1	dec2
6	dec3	mov2	inc2, cmp2	xor1
7		mov2	dec3, jgt2	inc2, cmp2
8	xor2, mov3			mov2, dec3
9	inc3, cmp3	mov3	xor2	
10	dec4	mov3	inc3, cmp3	xor2
11		mov3	dec4, jgt3	inc3, cmp3
12	xor3, mov4			mov3, dec4
13	inc4, cmp4	mov4	xor3	
14	dec5	mov4	inc4, cmp4	xor3
15		mov4	dec5, jgt4	inc4, cmp4
16	xor4, mov5			mov4, dec5
17	inc5, cmp5	mov5	xor4	
18	dec6	mov5	inc5, cmp5	xor4
19		mov5	dec6, jgt5	inc5, cmp5
20	xor5			mov5, dec6
21			xor5	
22				xor5

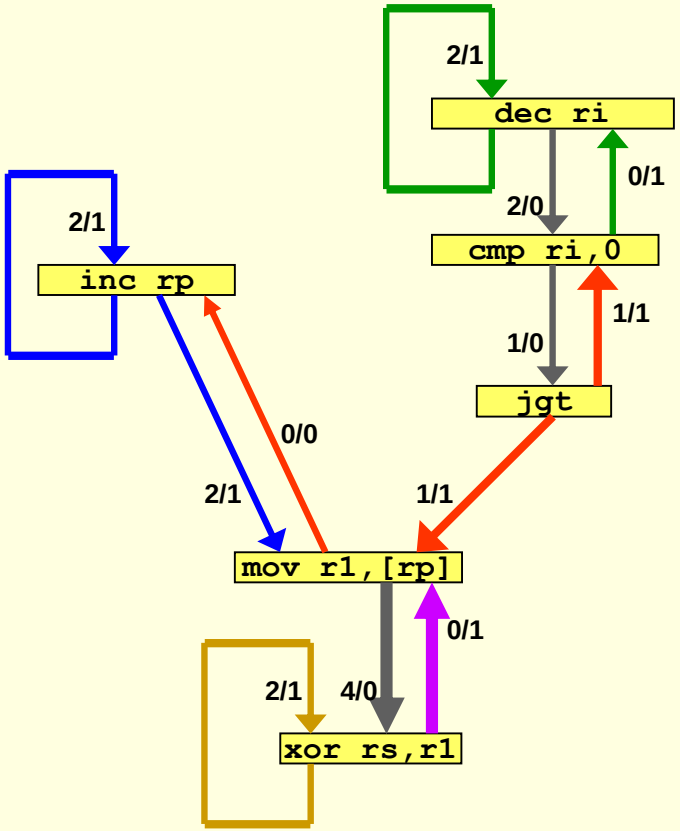
Příklad – software pipelining


```
mov r1,[rp]
dec ri
inc rp
cmp ri,0
jle l2
dec ri

l1:
xor rs,r1
mov r1,[rp]
inc rp
cmp ri,0
dec ri
jgt l1

l2:
xor rs,r1
```

čas	R	MEM	ALU	W
0	mov1, dec1			
1	inc1	mov1	dec1	
2	cmp1	mov1	inc1	dec1
3	dec2	mov1	cmp1, jgt1	inc1
4	xor1, mov2		dec2	mov1, cmp1
5	inc2, cmp2	mov2	xor1	dec2
6	dec3	mov2	inc2, cmp2	xor1
7		mov2	dec3, jgt2	inc2, cmp2
8	xor2, mov3			mov2, dec3
9	inc3, cmp3	mov3	xor2	
10	dec4	mov3	inc3, cmp3	xor2
11		mov3	dec4, jgt3	inc3, cmp3
12	xor3, mov4			mov3, dec4
13	inc4, cmp4	mov4	xor3	
14	dec5	mov4	inc4, cmp4	xor3
15		mov4	dec5, jgt4	inc4, cmp4
16	xor4, mov5			mov4, dec5
17	inc5, cmp5	mov5	xor4	
18	dec6	mov5	inc5, cmp5	xor4
19		mov5	dec6, jgt5	inc5, cmp5
20	xor5			mov5, dec6
21			xor5	
22				xor5

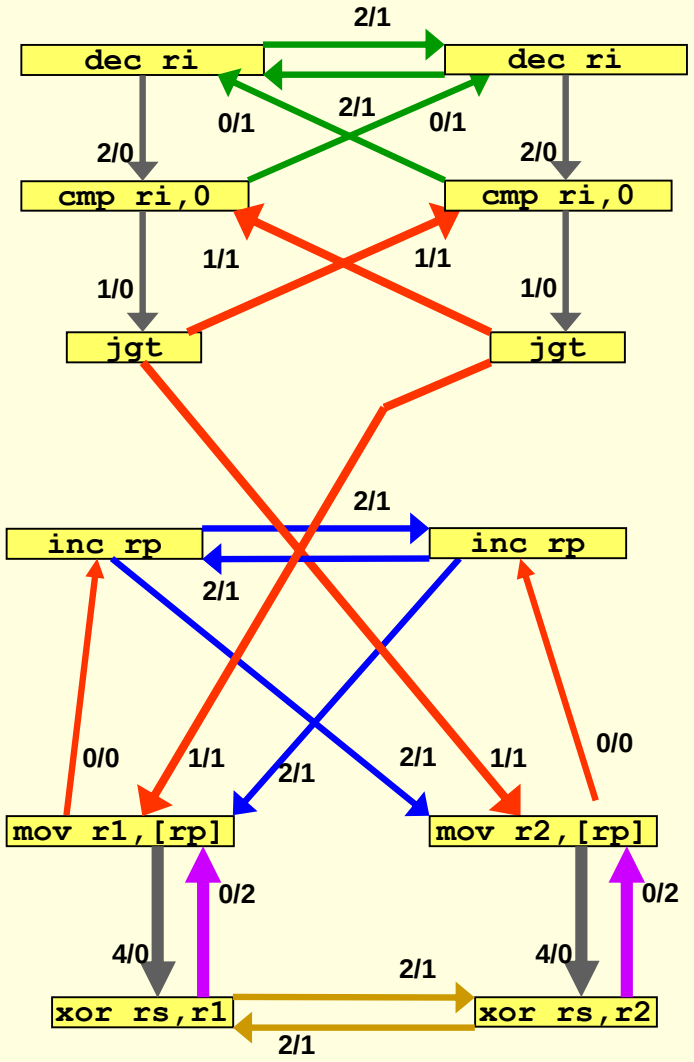
Software pipelining

- Duplikace proměnných (Variable expansion)
  - Duplikací proměnných lze odstranit některé antidependence
    - Teoreticky všechny registrové, roste však počet použitých registrů i velikost kódu
  - Duplikace proměnných se provede duplikací kódu a vhodným přeznačením



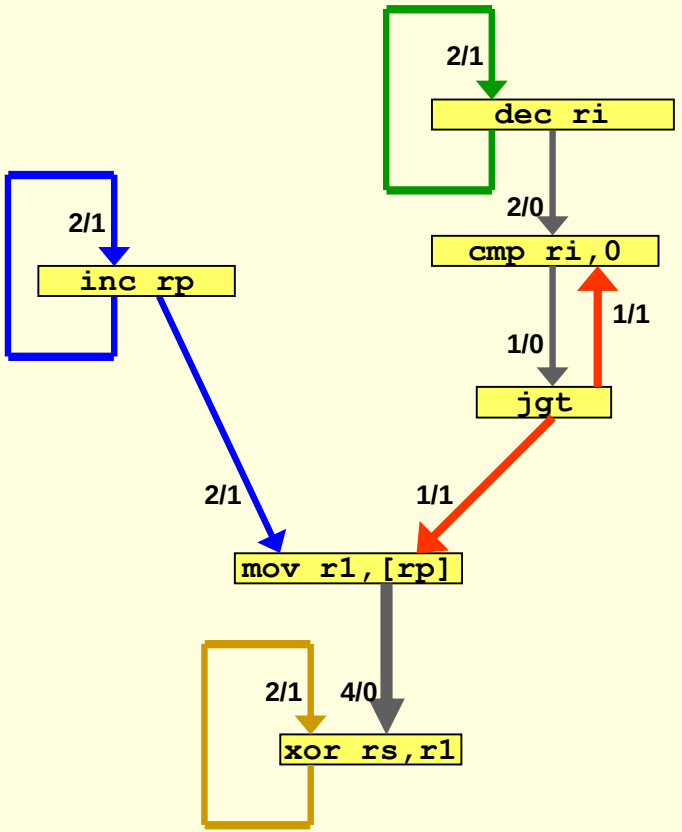
Variable expansion

- Duplikace proměnných (Variable expansion)
  - Duplikací proměnných lze odstranit některé antidependence
    - Teoreticky všechny registrové, roste však počet použitých registrů i velikost kódu
  - Duplikace proměnných se provede duplikací kódu a vhodným přeznačením
  - Příklad: Zdvojením proměnné r1 se kritický cyklus zkrátí z poměru 4/1 na 4/2



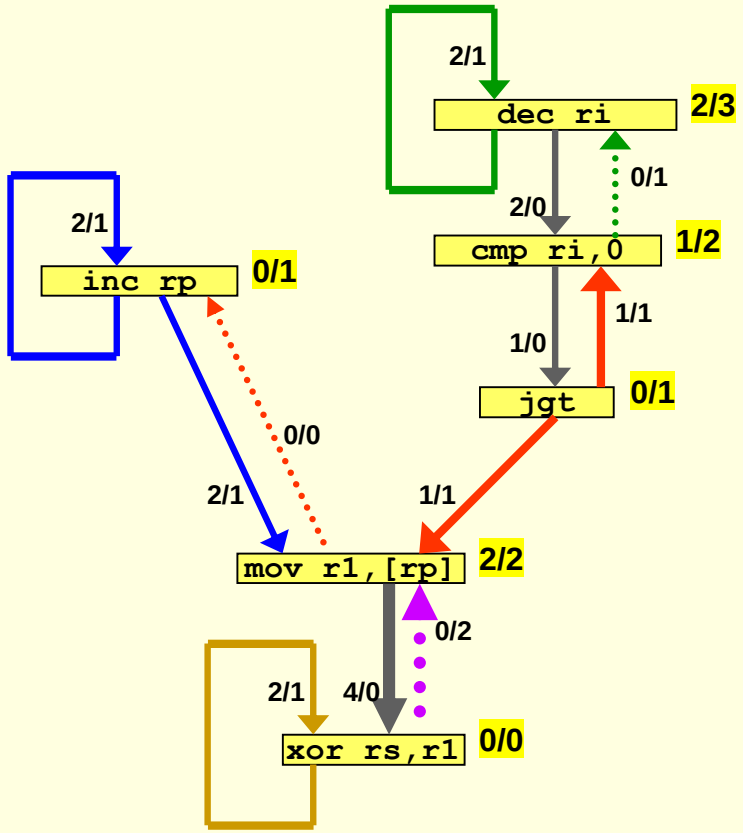
Software pipelining

- Duplikace proměnných (Variable expansion)
- Alternativní postup
  - Před schedulingem se odstraní všechny antidependence odstranitelné duplikací
  - Scheduling odhalí, u kterých proměnných je duplikace/multiplikace nutná
  - Na základě toho se provede potřebná multiplikace kódu
  - Tento postup fakticky dělá totéž, co renaming v OOO procesorech
  - Renaming nelze vynutit, nahrazuje se duplikací kódu



Příklad – modulo scheduling s duplikací


```
l1:
xor rs,r1        ; čas 0, iterace 2
inc rp           ; čas 0, iterace 3
cmp ri,0         ; čas 1, iterace 4
mov r1,[rp]      ; čas 2, iterace 4
dec ri           ; čas 2, iterace 5
jle l2           ; čas 3, iterace 4
xor rs,r2        ; čas 3, iterace 3
inc rp           ; čas 3, iterace 4
cmp ri,0         ; čas 4, iterace 5
mov r2,[rp]      ; čas 5, iterace 5
dec ri           ; čas 5, iterace 6
jgt l1           ; čas 6, iterace 5
l2:
```



čas	R	MEM	ALU	W
0	xor2, inc3	mov3	dec4, jgt3	mov2, cmp3
1	cmp4	mov3	xor2, inc3	dec4
2	mov4, dec5	mov3	cmp4	xor2, inc3

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 );

/* ... */
```