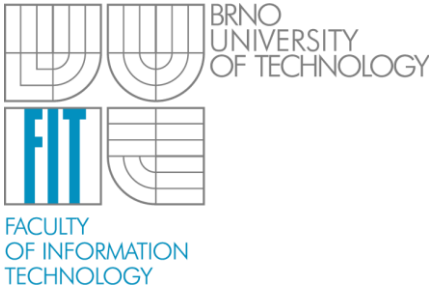
Pokročilé metody syntézy číslicových obvodů (High Level Synthesis)

Tomáš Martínek

Vysoké učení technické v Brně, Fakulta informačních technologií v Brně
Božetěchova 2, 612 66 Brno
ant@fit.vutbr.cz



INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Osnova

- Úvod
- Reprezentace obvodu
- Základní transformace
- Plánování
- Přiřazení
- Alokace
- Generování RTL
- Shrnutí

Motivace

- Vývoj na trhu s elektronikou
  - Zvyšující se stupeň integrace na čipu
  - Schopnosti návrhářů vyvíjejících obvody konvenční technikou (RTL úroveň) jsou limitovány
- => Velký tlak na zvýšení produktivity návrhářů

Rok	1991	1994	1996	1998	2000	2002	2004	2006	2008	2010
Technologie	0,7u	0,5u	0,35u	0,25u	0,18u	0,13u	90n	65n	54n	32n
Hradel/mm2	1k	5k	15k	30k	45k	80k	150k	300k	600k	1,2M
Hradel/čip (50mm2)	50k	250k	750k	1,5M	2,2M	4M	7,5M	15M	30M	60M
Hradel/návrháře na rok	4k	6k	9k	40k	56k	91k	125k	200k	200k	200k
Návrhářů/rok (50mm2)	~10	~40	~80	~40	~40	~43	~60	~75	~150	~300

Jak zvýšit produktivitu?

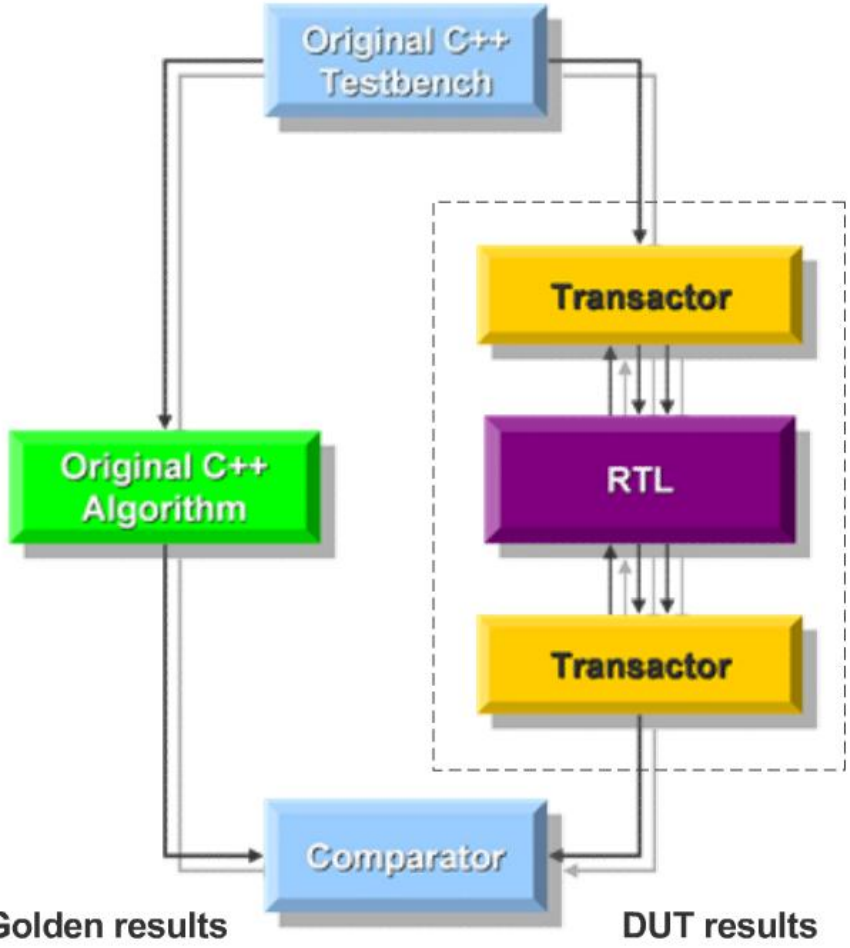
- Znovu-používáním existujících bloků (IP cores)
  - Dokáže zvýšit produktivitu přibližně na pětinásobek
- Změnou technik pro návrh obvodu
  - Přechod z úrovně RTL na obecnější popis algoritmů např. v jazyce C/C++
  - Přirovnáváno k přechodu od assembleru k vyšším programovacím jazykům
  - Techniky syntézy obvodu z jazyka na vyšší úrovni abstrakce na úroveň RTL nebo přímo na masku obvodu ASIC popř. konfiguraci čipu FPGA
  - Dokáže zvýšit produktivitu na desetinásobek a více

Jak HLS pomáhá zvýšit produktivitu?

- Jednodušší specifikace úlohy
- Snazší realizace/integrace systému složeného z softwaru a hardwaru (HW/SW Codesign)
- Efektivnější verifikace/validace obvodu
  - Simulace na úrovni C/C++ je výrazně rychlejší než na úrovni RTL
  - Úroveň popisu v C/C++ je obecně méně náchylnější k tvorbě chyb
  - Menší počet chyb = méně vývojových cyklů => kratší doba pro dosazení výrobku na trh
- Efektivnější ladění obvodu
  - Lze snadno využít existujících nástrojů vyvinutých pro C/C++
- Rychlý průzkum prostoru možných mikro-architektur obvodu

Verifikace/validace obvodu

- Pro validaci obvodu je potřeba napsat testbench v jazyce C/C++
- Testbench = program, který otestuje požadovanou funkci dostatečnou sadou vstupních hodnot
- Testbench je aplikován na původní zdrojový kód (spuštěný na běžném proceosoru) a současně na vygenerovaný RTL obvod (simulovaný např. v ModelSIMu)
- HLS syntezátor automaticky zajistí vytvoření rozhraní mezi testbench souborem a prostředím ModelSIM
- Výsledky obou testů jsou porovnány a v případě neshody je chyba dohledána a zobrazena na časovém diagramu ModelSIMu



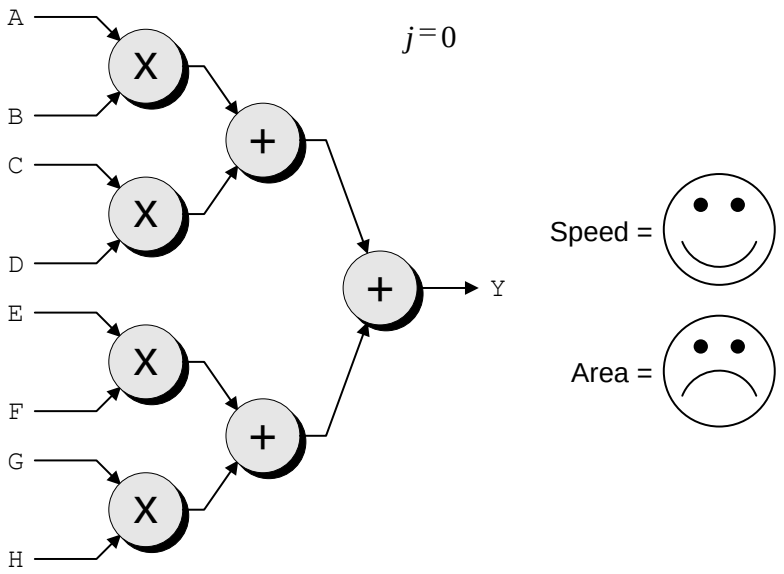
Průzkum mikro-architektur

- HLS vs. RTL
  - Technikou RTL se obvykle vytvoří jedna architektura, u které se ladí frekvence
  - HLS vytvoří hned několik různých architektur s různým poměrem množství zdrojů vs. Doba výpočtu
- Příklad MAC
  - Použití v DSP aplikacích, operace konvoluce, filtrace obrazu

```
y[n] := 0;
for j=0 to M-1 loop
   y[n] := y[n] + h[j]*x[n-j];
end loop;
```

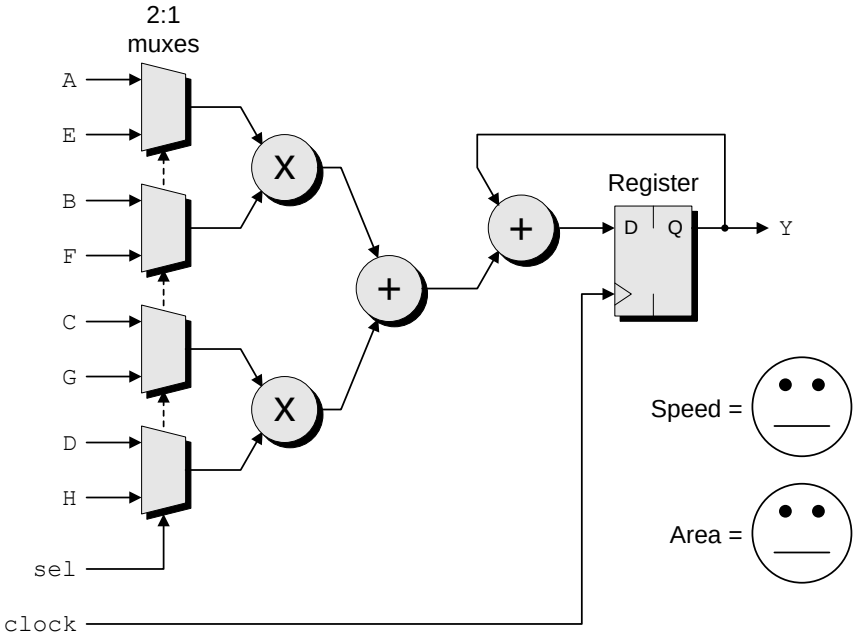
- Plně paralelní implementace
- Výpočet proveden v jednom kroku
- Řešení náročné na spotřebu zdrojů

$$y[n] = \sum_{j=0}^{M-1} h[j].x[n-j]$$

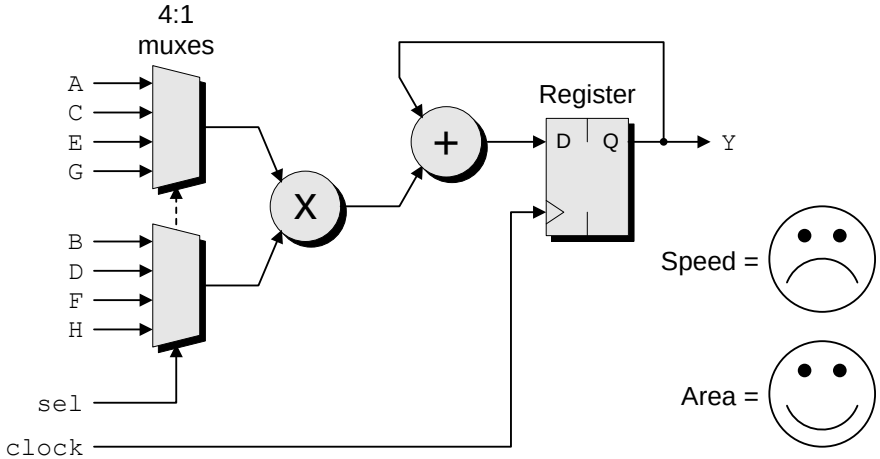


Průzkum mikro-architektur

- Sdílení zdrojů
- Pouze jedna větev stromu, redukce zdrojů cca na polovinu
- Výpočet ve dvou krocích, uložení mezivýsledku do pomocného registru



- Sdílení zdrojů
- Pouze jedna násobička a sčítačka
- Maximální úspora zdrojů
- Výpočet výsledku ve čtyřech krocích s využitím pomocného registru



Dostupné HLS nástroje

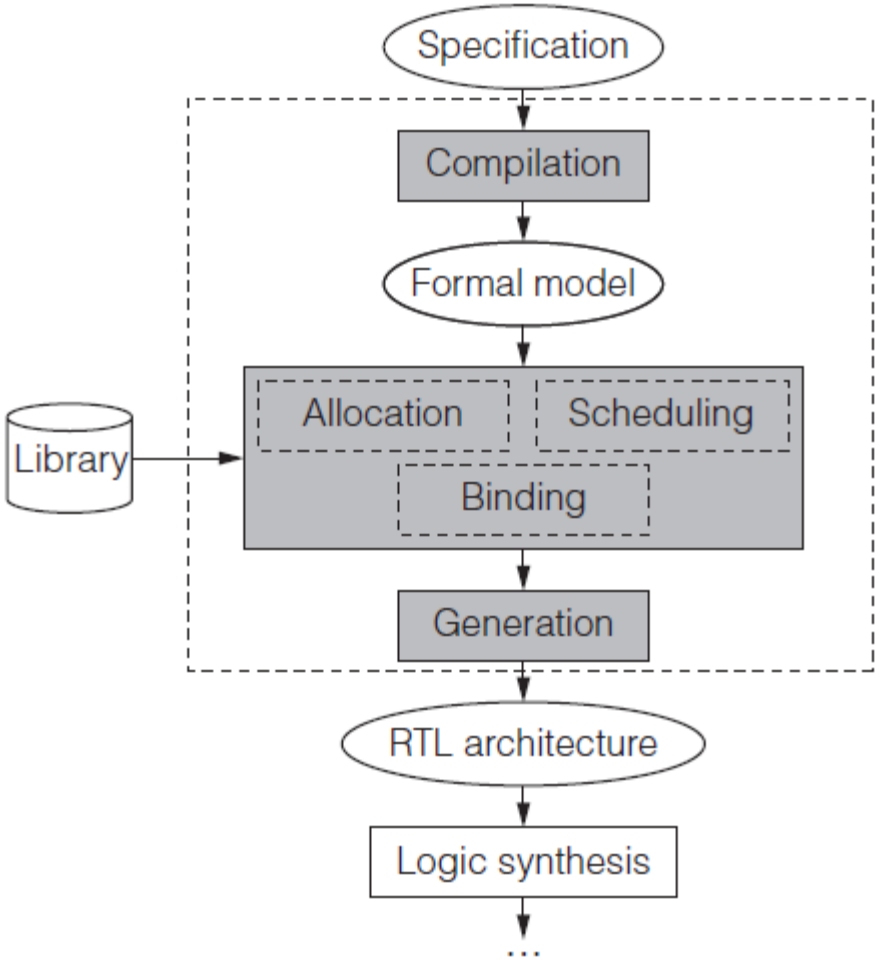
- Cataput C, Mentor Graphics, (ANSI C/C++)
- CyberWorkBench, NEC (C/C++)
- Forte Cynthesizer, Forte Desing Systems (SystemC)
- AutoPilot, AutoESL Design Technologies, (SystemC, ESL)
- Agility Compiler, Celoxica, (SystemC)
- PICO Express, Synfora, (C/C++)
- C-to-Silicon (C2S) compiler, Cadence, (C/C++)
- …

"Designer that moved to C-level design usually don't want to came back to RTL level".

Pascal Urard, STMicroelectronics

Syntéza obvodů na vyšší úrovni

- Vstup:
  - Popis algoritmu ve vyšším programovacím jazyce (C/C++, popř. varianty)
- Výstup:
  - Maska obvodu ASIC nebo konfigurace obvodu FPGA
- Základní schéma návrhu:
  1. Kompilace vstupního kódu a převod do vhodné formální reprezentace (Control-Data Flow Graph)
  2. Plánování, alokace a přiřazení prostředků
  3. Generování RTL schéma
  4. Logická syntéza pomocí konvenčních nástrojů

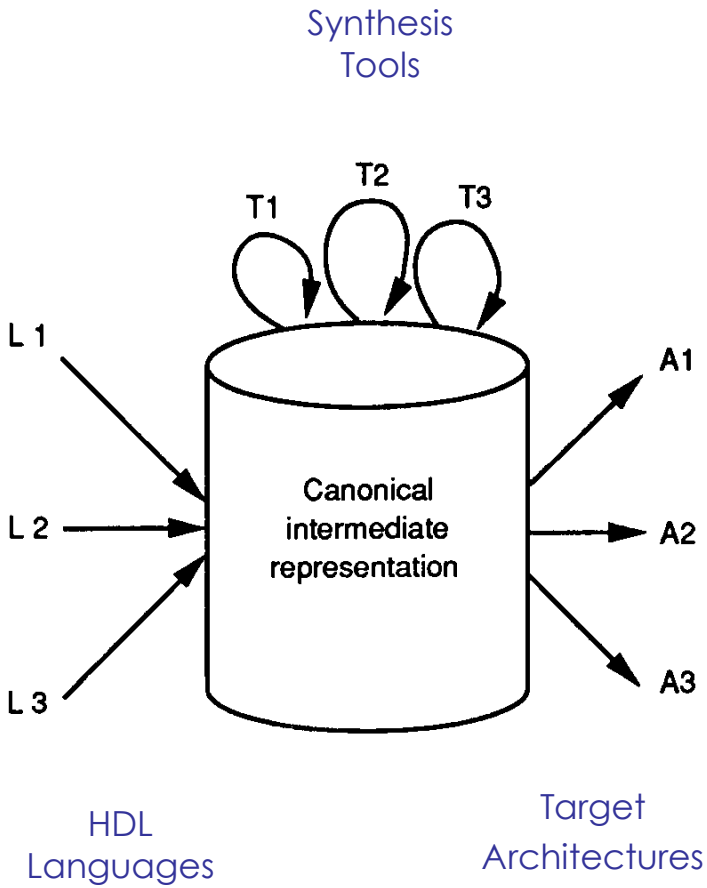


Osnova

- Úvod
- Reprezentace obvodu
- Základní transformace
- Plánování
- Přiřazení
- Alokace
- Generování RTL
- Shrnutí

Požadavky na reprezentaci obvodu

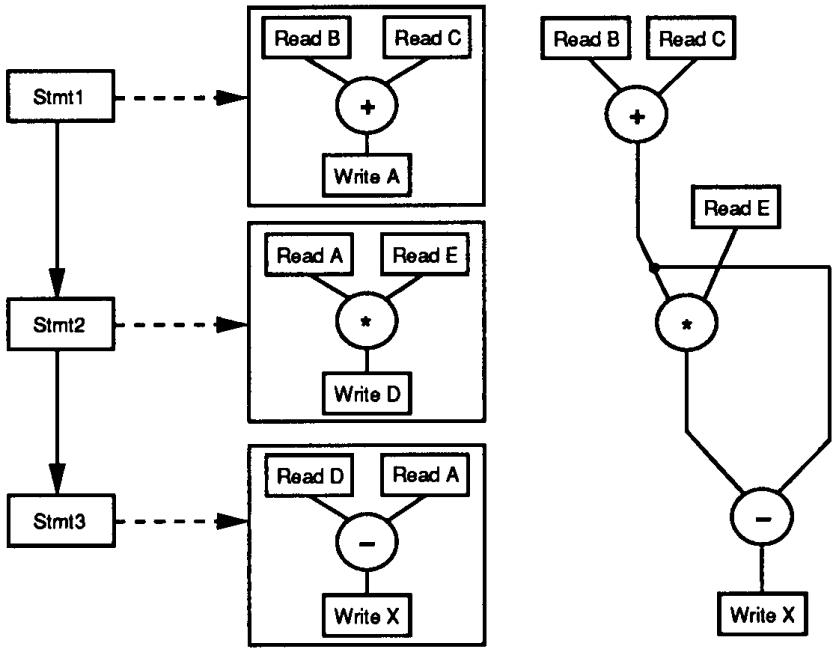
- Cílem je vytvořit vhodnou reprezentaci obvodu na úrovni mezi HDL a cílovou architekturou
- Základní požadavky:
  - Uniformní z pohledu uživatele i nástroje
  - Nezávislá na použitém HDL jazyku
  - Dostatečně silná pro podporu různých cílových architektur
- Podobně jako u kompilátoru SW (intermediální kód)
- Nad obvodem lze provádět optimalizace nezávisle na cílové architektuře



Přechod od HDL k DFG

- Data Flow Graf (DFG) jako vhodná reprezentace
- Základní vlastnosti:
  - lze vidět paralelizaci výpočtu
  - hrany grafu označují datové závislosti
  - výška grafu ukazuje počet kroků nutných pro dokončení výpočtu
- Při kompilaci se vytvoří syntaktický strom na základě jednotlivých příkazů (statements) a následně se na základě datových závislostí sestrojí DFG

```
A := B + c;
D := A * E;
X := D - A;
```



Parse tree

Data Flow Graph

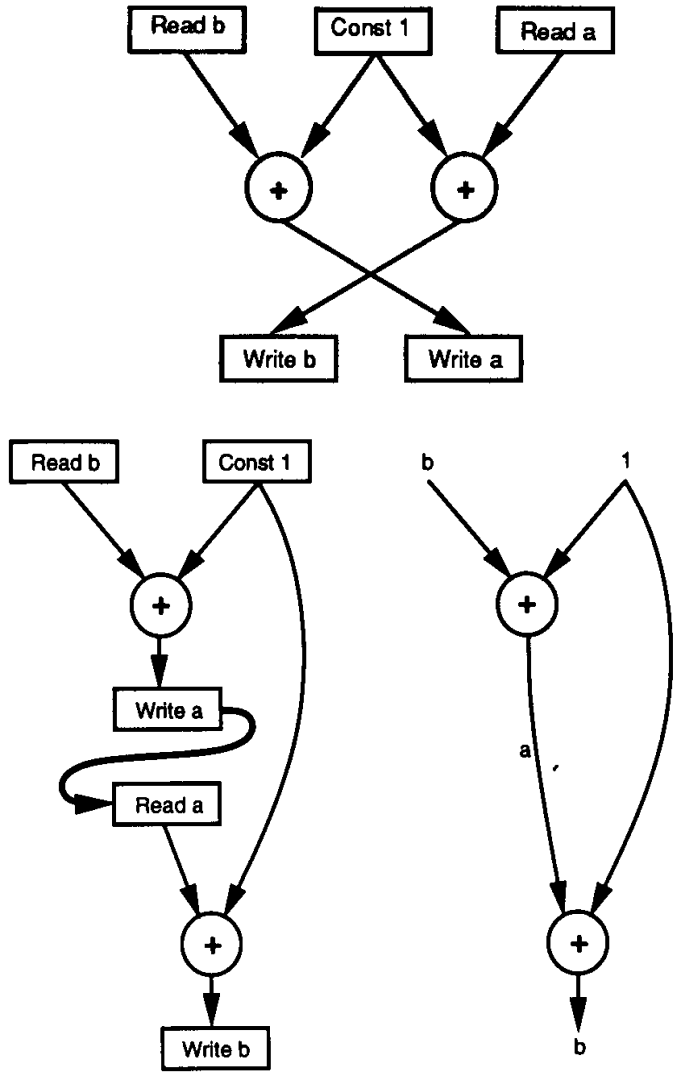
Vliv sémantiky jazyka

- Podobné konstrukce jazyka mohou mít odlišný sémantický vliv
- Přiřazení signálů
  - paralelní zpracování

```
A <= B + 1;
B <= A + 1;
```

- Přiřazení proměnných
  - dodržení sekvence

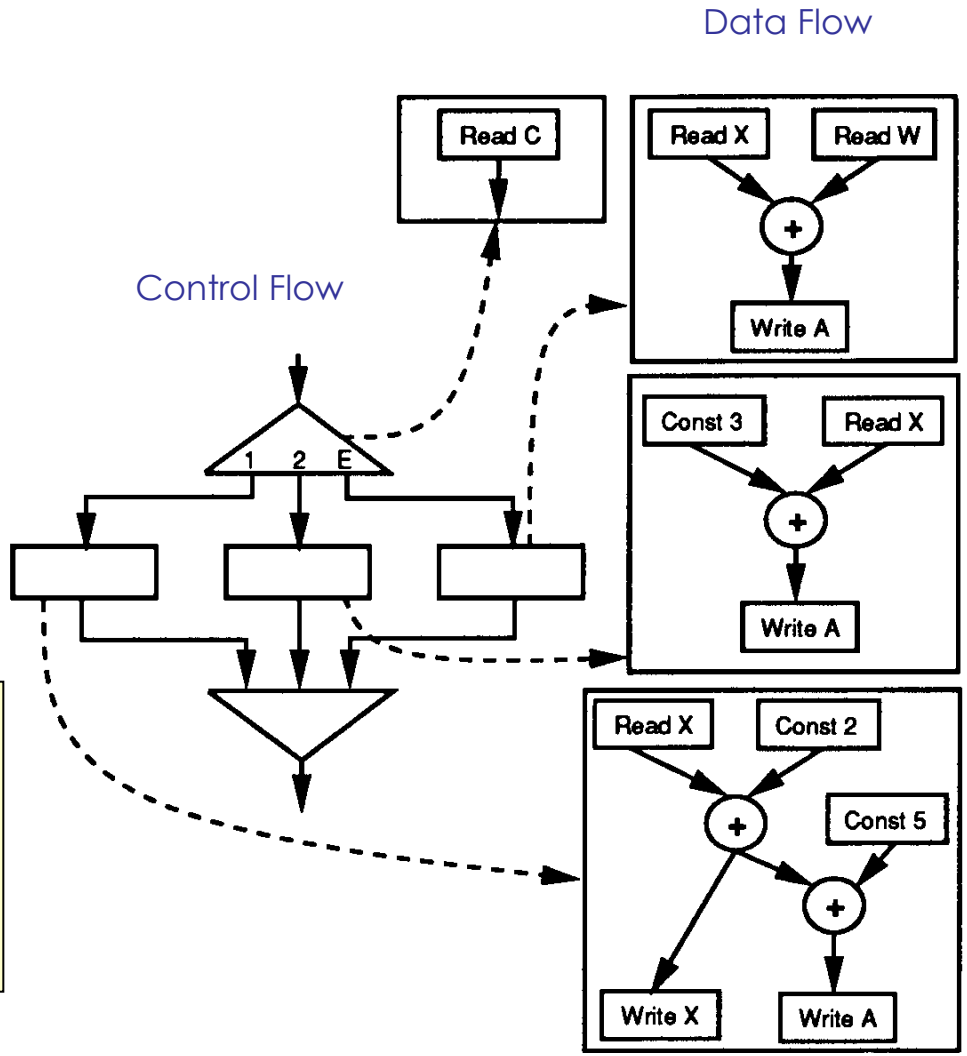
```
A := B + 1;
B := A + 1;
```



Control Data Flow Graf

- V průběhu výpočtu je nutné uvažovat kromě datových i řídící konstrukce – podmíněné vyhodnocení, cykly apod. – Control Flow Graf (CFG)
- Popis obvodu tvoří spojení obou těchto grafů do Control Data Flow Graf (CDFG)
- Příklad:

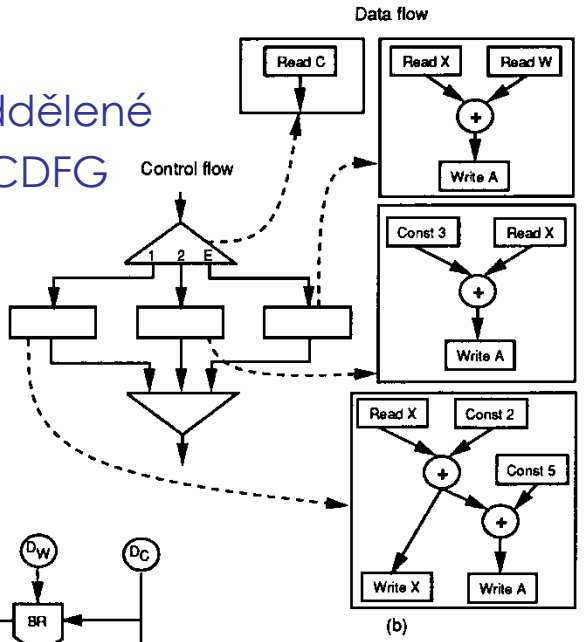
```
Case C is
   when 1 => X := X + 2;
             A := X + 5;
   when 2 => A := X + 3;
   when others => A := X + W;
end case;
```

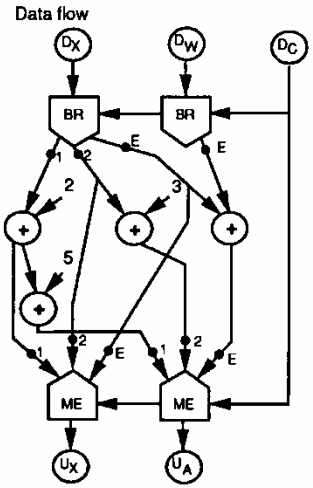


Různé reprezentace CDFG

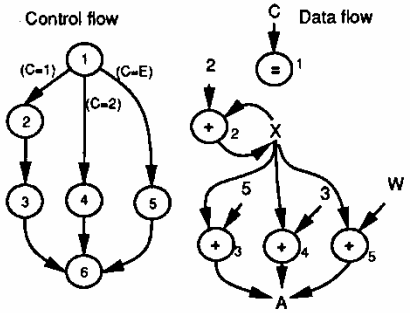
- Existují různé reprezentace Control Data Flow grafu:
  - Oddělené CDFG
    - Control a Data flow grafy jsou oddělené
  - DeJong's graf
    - Kontrolní informace jsou uloženy explicitně uvnitř Data Flow grafu
  - SSIM graf
    - Control Flow graf je kopií Data Flow grafu a ukazuje pořadí provádění jednotlivých operací

Oddělené CDFG





DeJong's flow graph



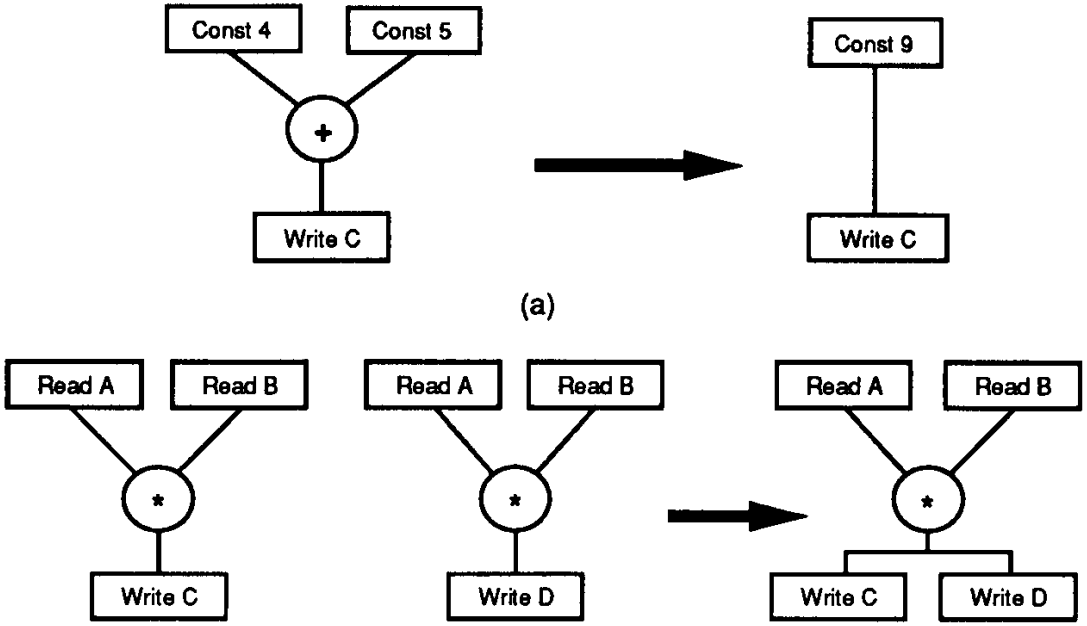
SSIM flow graph

Osnova

- Úvod
- Reprezentace obvodu
- Základní transformace
- Plánování
- Přiřazení
- Alokace
- Generování RTL
- Shrnutí

Základní transformace

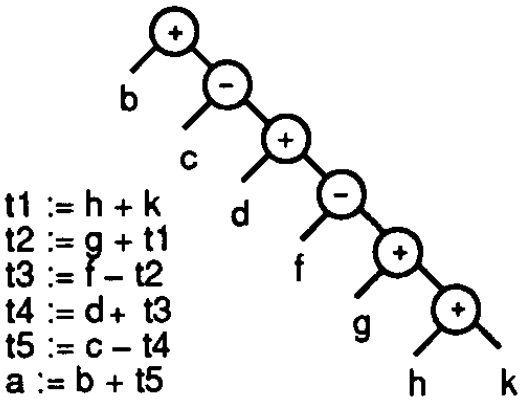
- Optimalizace na úrovni kompilátoru (podobně jako u SW)
  - Vyhodnocení výrazů s konstantami
  - Odstranění redundantních operátorů
  - Eliminace mrtvého kódu
  - Vytknutí společných částí před smyčku

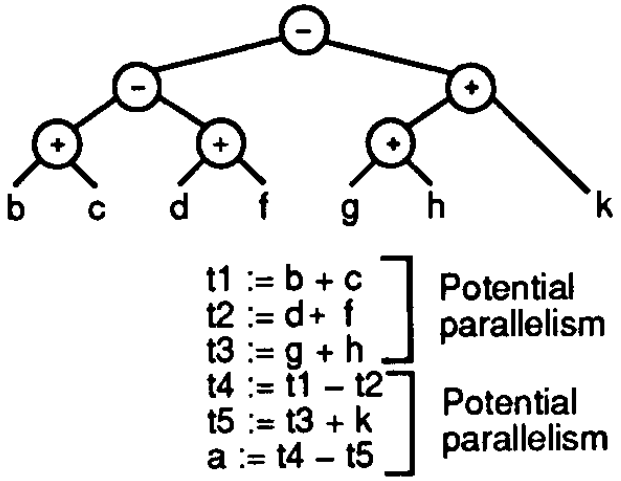


Základní transformace

- Redukce výšky grafu:
  - Lze využívat komutativní, distributivních a asociativní vlastnosti některých operátoru
  - Jejich přeuspořádání může vést na menší výšku stromu, využití paralelizmu a zrychlení výpočtu
  - Příklad:

```
a := b + c – (d + f – (g + h + k))
```

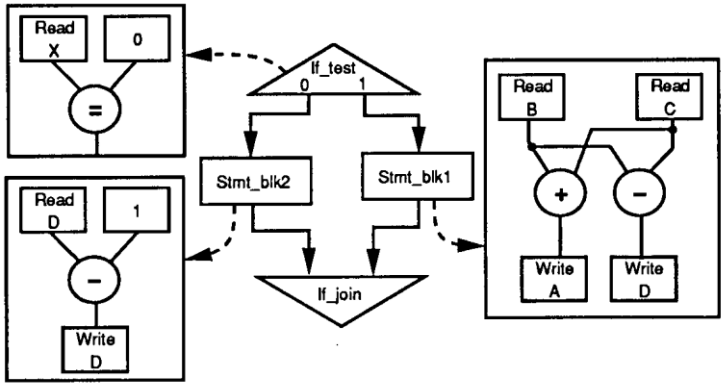


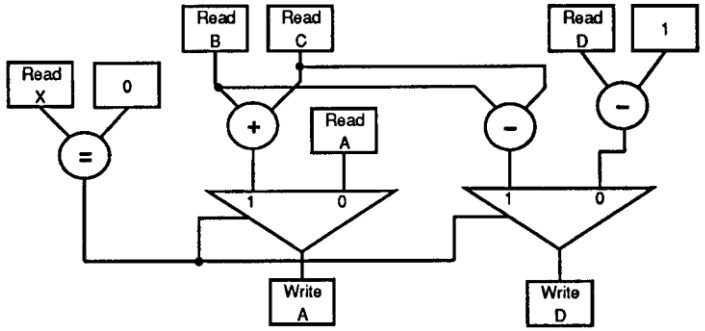


Základní transformace

- Transformace CF do DF (flattening):
  - Vyhodnocení podmínek z CF je promítnuto do DF
  - Všechny sekce se provádějí paralelně a na závěr se vybere výsledek pomocí multiplexoru
  - Podmínka pro výběr se může počítat v průběhu výpočtu (menší výška grafu)
  - Zdroje resp operátory nelze sdílet mezi oddělenými větvemi – náročnější na zdroje
  - Ztratí se informace o oddělených sekcích

```
if (X = 0) then
   A := B + C;
   D := B - C;
else
   D := D - 1;
end if;
```

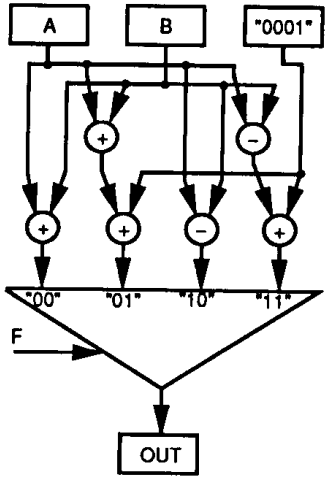




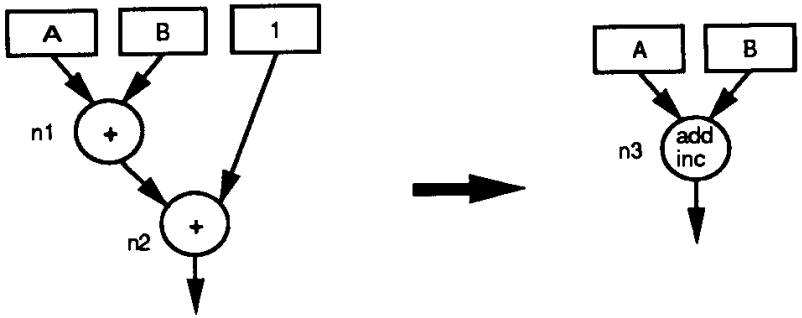
Základní transformace

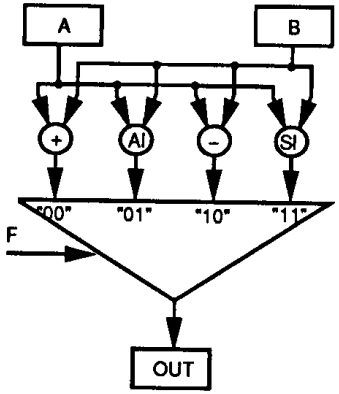
- Sdružování operátorů:
  - Pokud se v obvodu často vyskytuje skupina stejných operací, lze je sdružit
  - Sdružování může probíhat až do jediného uzlu – obecná ALU

```
Case F is
 when "00" =>
    OUT <= A + B;
 when "01" =>
    OUT <= A + B + "0001";
 when "10" =>
    OUT <= A - B;
 when "11" =>
    OUT <= A - B + "0001";
end case;
```

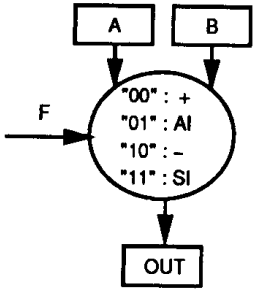


(b)





(c)



(d)

Osnova

- Úvod
- Reprezentace obvodu
- Základní transformace
- Plánování
- Přiřazení
- Alokace
- Generování RTL
- Shrnutí

Plánování (Scheduling)

- Plánování je úloha, která rozděluje vstupní CDFG do pod-grafů, které jsou vykonávány v jednom kontrolním kroku (taktu hodinového signálu)
- Počet operací v pod-grafu určuje počty a typy funkčních jednotek, které musí být v daném kontrolním kroku k dispozici
- Funkční jednotky lze sdílet mezi více kontrolními kroky
- Proces plánování tak může ovlivnit jak celkové množství potřebných funkčních jednotek, tak celkovou dobu výpočtu

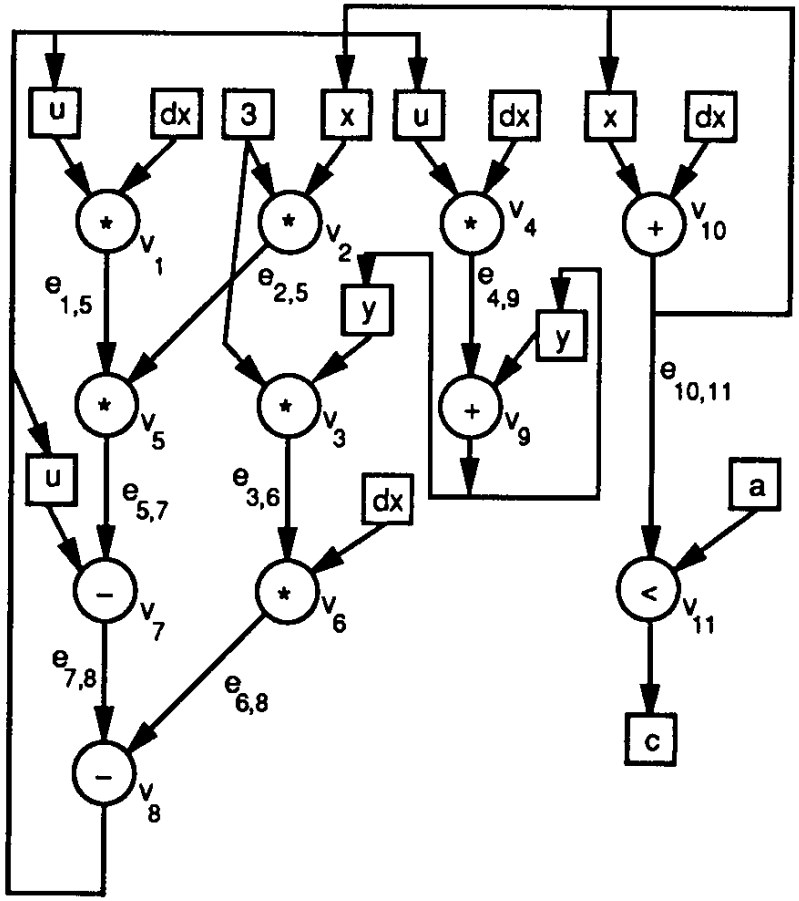
Příklad obvodu

- Řešení diferenciální rovnice:

  $y'' + 3xy' + 3x = 0$

- Často používaný příklad pro porovnání různých přístupů (benchmark-ový obvod)
- Zápis algoritmu:

```
while (x < a) do
   x1 := x + dx;
   u1 := u - (3*x*u*dx) - (3*y*dx)
   y1 := y + (u*dx)
   x := x1; u:=u1; y:=y1;
end while;
```



- V dalším výkladu bude pro jednoduchost používáno pouze tělo smyčky

Základní plánovací algoritmy

- ASAP (As Soon As Possible)
- Při plánování operátoru vybírá nejbližší možný kontrolní krok
- Princip algoritmu:
  - Uzly, které nemají žádného předka, jsou naplánovány do prvního kroku
  - Následně jsou prohledávány všechny ostatní uzly a pokud je nalezen uzel jehož všichni předci jsou již naplánováni, potom tento uzel je naplánován do kroku, který odpovídá maximu jeho předků inkrementovaného o jedničku
  - Algoritmus pokračuje, dokud nejsou naplánovány všechny uzly

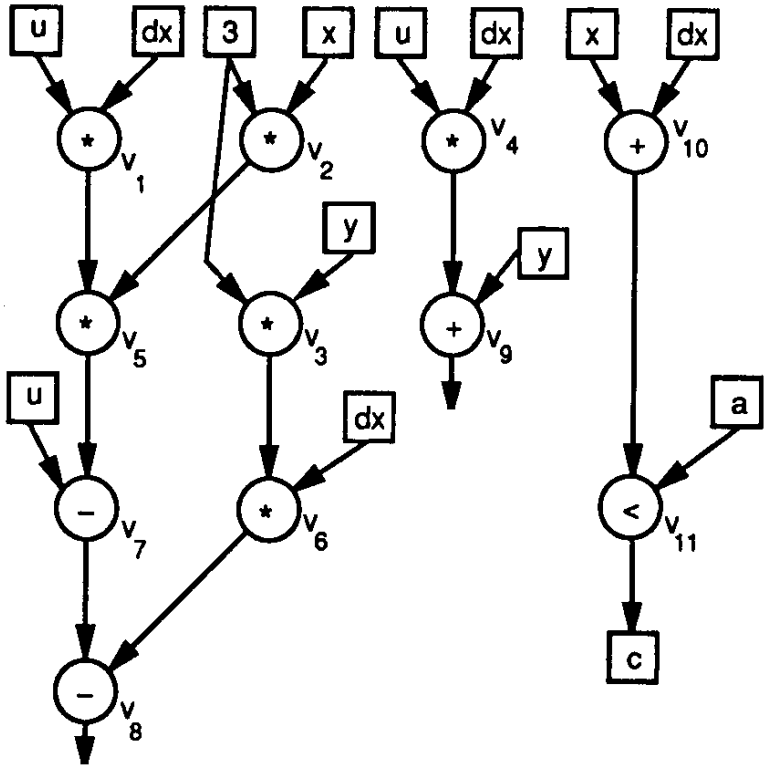
Algoritmus ASAP

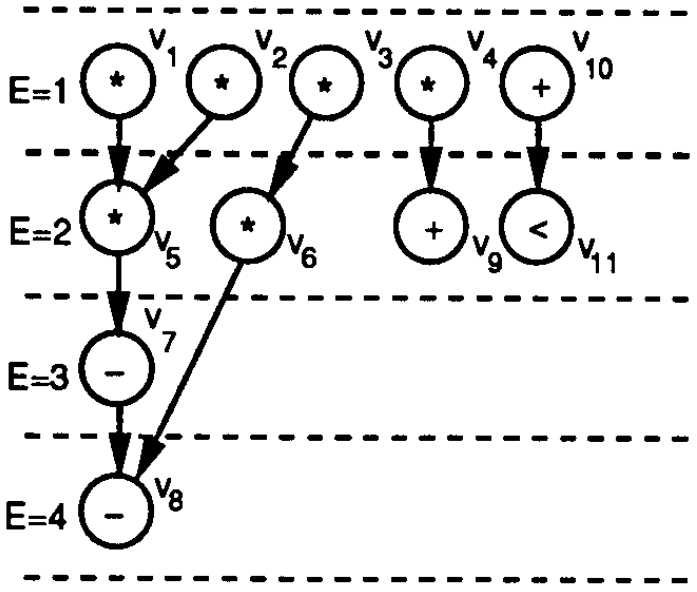
```
for each node v_i ∈ V do
    if Pred_{v_i} = φ then
        E_i = 1;
        V = V - { v_i };
    else
        E_i = 0;
    endif
endfor

while V ≠ φ do
    for each node v_i ∈ V do
        if ALL_NODES_SCHED(Pred_{v_i}, E) then
            E_i = MAX(Pred_{v_i}, E) + 1;
            V = V - { v_i };
        endif
    endfor
endwhile
```

Základní plánovací algoritmy

- Příklad:

```
x1 := x + dx;
u1 := u - (3*x*u*dx) - (3*y*dx)
y1 := y + (u*dx)
```





- Graf rozdělen do čtyř pod-grafů Realizace vyžaduje: 4xMULT, 1xADD, 1xSUB, 1xCMP

Základní plánovací algoritmy

- DFG je acyklický orientovaný graf definován jako uspořádaná dvojice G(V,E) kde:
  - V je konečná množina všech vrcholů
  - E je konečná množina orientovaných hran e_{ij} z vrcholu v_i do vrcholu v_j
- Nad tímto grafem jsou dále definovány operace:
  - $Pred_{vi}$ – množina přímých předchůdců vrcholu v_i
  - $Succ_{vi}$ – množina přímých následovníků vrcholu v_i
- Proces plánování přiřazuje každému vrcholu v_i kontrolní krok E_i, ve kterém bude proveden
- Pomocná funkce ALL_NODES_SCHED(*V*,*E*) – vrací
  - true, pokud vrcholy *V* jsou již naplánovány,
  - false, v ostatních případech

Základní plánovací algoritmy

- ALAP (As Late As Possible)
- Při plánování operátoru vybírá nejvzdálenější možný kontrolní krok
- Princip algoritmu:
  - Uzly, které nemají žádného potomka, jsou naplánovány do kroku T (očekávané maximu)
  - Následně jsou prohledávány všechny ostatní uzly a pokud je nalezen uzel jehož všichni potomci jsou již naplánováni, potom tento uzel je naplánován do kroku, který odpovídá minimu jeho potomků sníženého o jedničku
  - Algoritmus pokračuje, dokud nejsou naplánovány všechny uzly

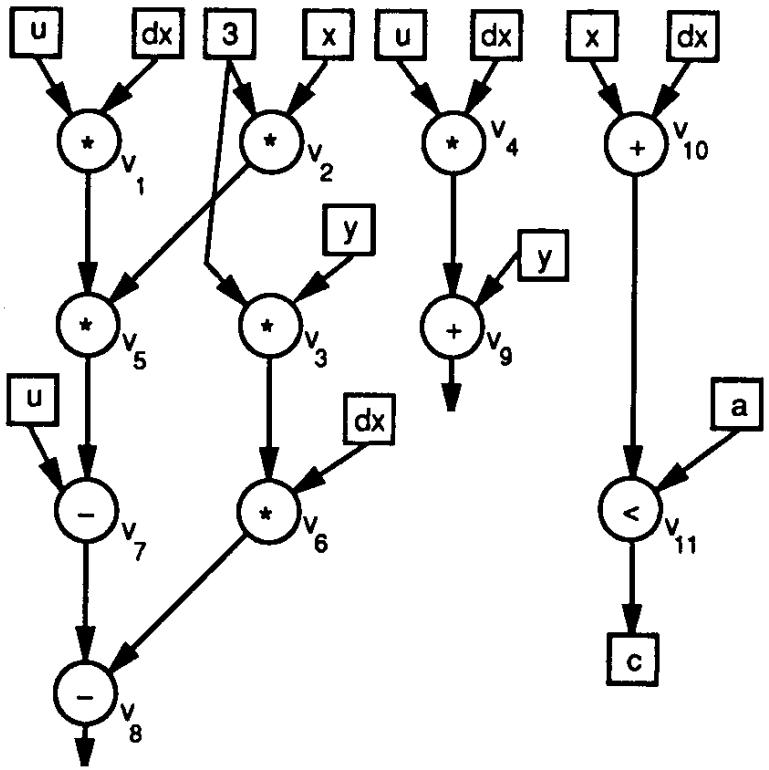
Algoritmus ALAP

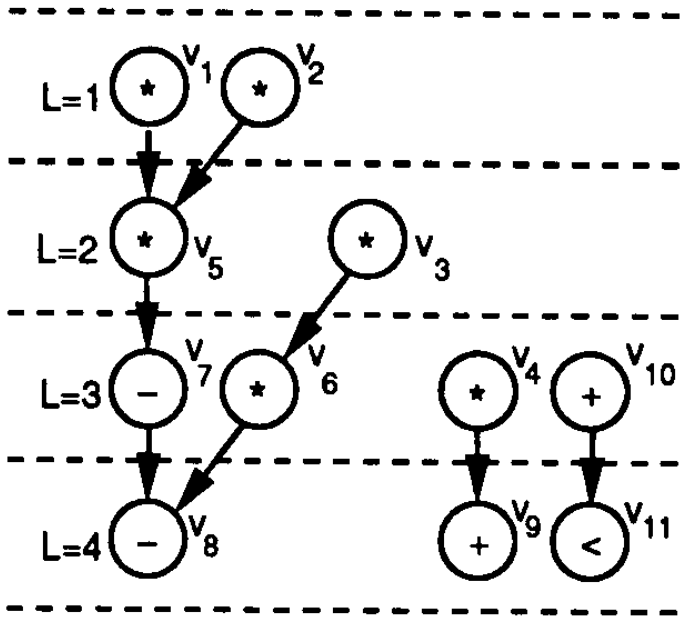
```
for each node v_i ∈ V do
    if Succ_{v_i} = φ then
        L_i = T;
        V = V - { v_i };
    else
        L_i = 0;
    endif
endfor

while V ≠ φ do
    for each node v_i ∈ V do
        if ALL_NODES_SCHED(Succ_{v_i}, L) then
            L_i = MIN(Succ_{v_i}, L) - 1;
            V = V - { v_i };
        endif
    endfor
endwhile
```

Základní plánovací algoritmy

- Příklad:

```
x1 := x + dx;
u1 := u - (3*x*u*dx) - (3*y*dx)
y1 := y + (u*dx)
```

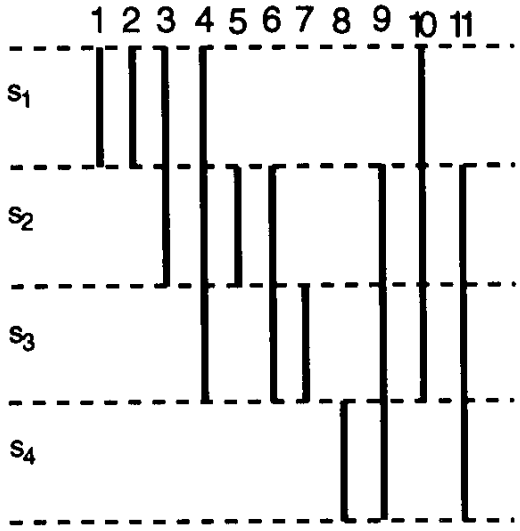


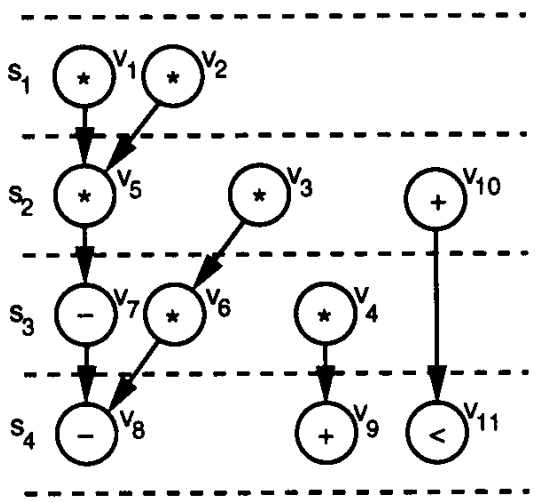


- Graf rozdělen do čtyř pod-grafů Realizace vyžaduje: 2xMULT, 1xADD, 1xSUB, 1xCMP

Základní plánovací algoritmy

- ASAP a ALAP se často používají jako pomocné výpočty pro složitější plánovací algoritmy
- Jejich hlavní předností je, že dokáží označit horní (E_i) a spodní (L_i) hranice kontrolních kroků, do kterých mohou být plánovány jednotlivé operace.
- Aritmetický rozdíl těchto hranic vyjadřuje tzv. mobilitu operátoru





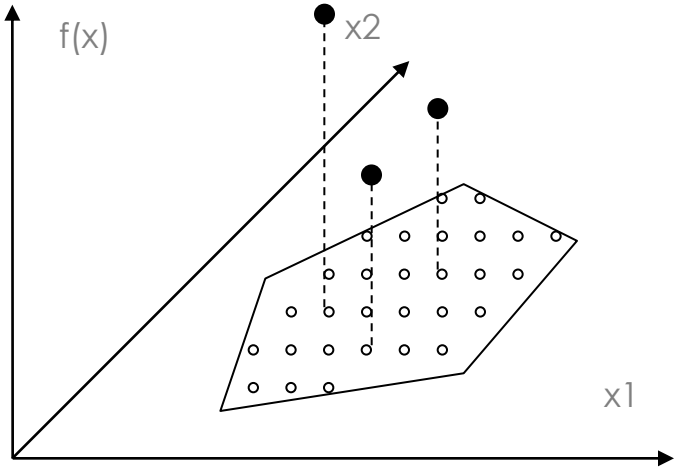
Časově omezené plánování

- Formulace problému:
  - Je zadán maximální počet kontrolních kroků, do kterých musí být DFG naplánován
  - Cílem je nalézt takový plán, který spotřebuje co nejméně výpočetních zdrojů
- Příklad použití: v DSP aplikacích, kde máme uvedenu fixní vzorkovací frekvenci vstupních dat a výpočet nad každým vzorkem dat, musí být ukončen v omezeném časovém intervalu
- Obecně se jedná o NP-těžký problém!
- Způsoby řešení:
  - Exaktní výpočet: Celočíselné lineární programování (ILP)
  - Přibližný výpočet: S využitím heuristik, např. Force-Directed heuristiky

Celočíselné lineární programování (ILP)

- Úloha plánování je nejprve převedena do standardního tvaru úlohy celočíselného programování:

$$\begin{aligned} &min[f(x)] \\ &Ax \leq b \\ &x \geq 0 \end{aligned}$$



- Hledáme minimu funkce $f(x)$, závislé na proměnných x_i
- Hodnoty proměnných x_i jsou celočíselné, kladné a jsou navíc ohraničeny na intervalu zadaném soustavou nerovnic $Ax \leq b$ (maticový tvar, tj, A je matice, x a b jsou vektory)

Celočíselné lineární programování (ILP)

- Bude použito následující značení:
  - Operace: o_i, $1 \leq i \leq n$
  - Kontrolní krok: s_j, $1 \leq j \leq r$
  - Typ operace: t_k, $1 \leq k \leq m$
  - Počet jednotek provádějících operaci t_k: N_{tk}
  - Cena za realizaci operace t_k: C_{tk}
  - Proměnná $x_{i,j}$ označuje, za je operace o_i naplánována do kroku s_j (0 – není naplánována, 1 – je naplánována)
  - E_i: spodní hranice pro naplánování operace o_i získaná algoritmem *ASAP*
  - L_i: horní hranice pro naplánování operace o_i získaná algoritmem *ALAP*

Celočíselné lineární programování (ILP)

- Minimalizační funkce:
  - Cílem je minimalizovat množství spotřebovaných zdrojů

$$\min \left[\sum_{k=1}^{m} c_{t_k} \times N_{t_k} \right]$$

- Omezení prostoru proměnných $x_{i,j}$:
  1. Každá operace o_i musí být naplánována do některého kroku v rozsahu její mobility

$$\sum_{E_i \le j \le L_i} x_{i,j} = 1 \qquad \forall i, 1 \le i \le n$$

2. V každém kontrolním kroku nesmí být naplánováno více operací typu t_k než je dostupné množství N_{tk}

$$\sum_{i \text{ typu } t_k} x_{i,j} \leq N_{t_k} \qquad \forall j, 1 \leq j \leq r, \forall k, 1 \leq k \leq m$$

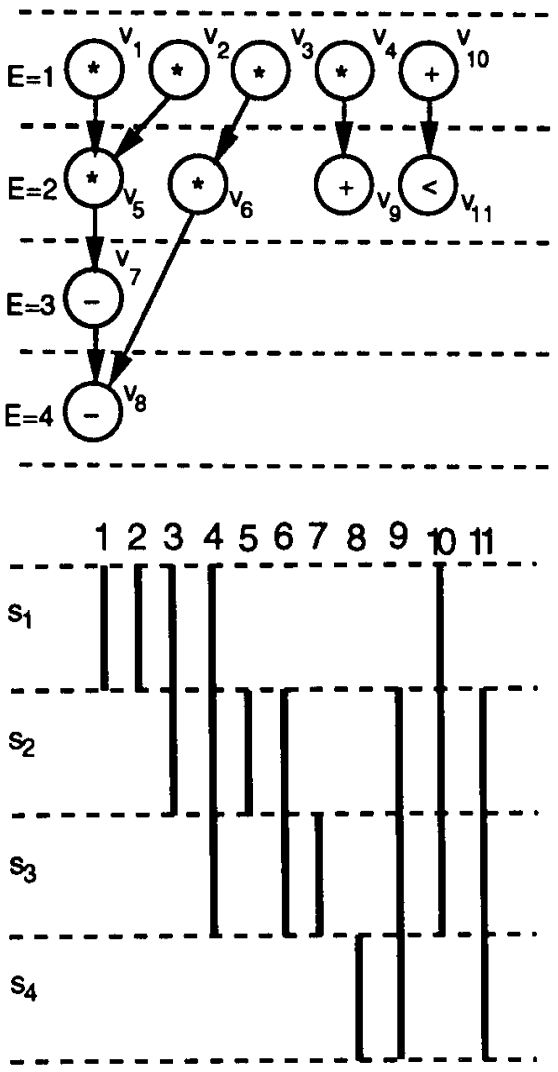
3. Operace musí být naplánovány s ohledem na datové závislosti

$$\left[\sum_{E_i \leq k \leq L_i} k \times x_{i,k} - \sum_{Ej \leq l \leq Lj} l \times x_{j,l} \right] \leq -1 \qquad \forall i, j, o_i \in \text{Pred}_{oj}$$

ILP - Příklad

- Příklad:
  - Technikou ILP nalezněte pro úlohu HAL plán s nejmenším počtem zdrojů
  - Maximální doba pro výpočet
    $T = 4$ takty hodinového signálu
  - Ceny za realizaci funkčních jednotek jsou:
    - $C_{*}=2$,
    - $C_{+}=C_{-}=C_{<}=1$
- Minimalizační funkce:

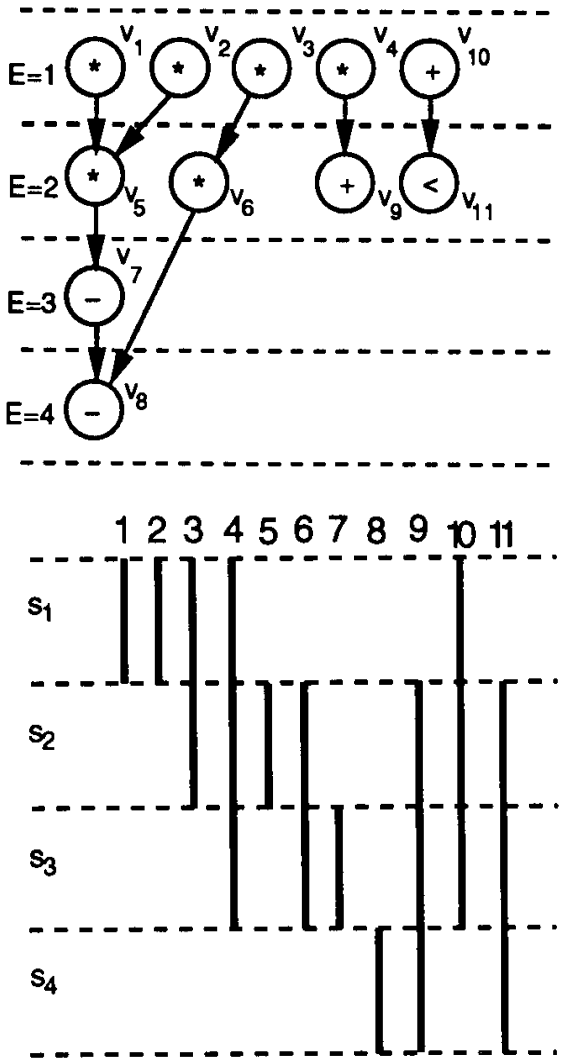
$$min[C_{*}xN_{*} + C_{+}xN_{+} + C_{-}xN_{-} + C_{<}xN_{<}]$$



ILP - Příklad

- Omezující podmínky
  1. Každá operace o_i musí být naplánována do některého kroku v rozsahu její mobility

$$
\begin{aligned}
x_{1,1} &= 1 \\
x_{2,1} &= 1 \\
x_{3,1} + x_{3,2} &= 1 \\
x_{4,1} + x_{4,2} + x_{4,3} &= 1 \\
x_{5,2} &= 1 \\
x_{6,2} + x_{6,3} &= 1 \\
x_{7,3} &= 1 \\
x_{8,4} &= 1 \\
x_{9,2} + x_{9,3} + x_{9,4} &= 1 \\
x_{10,1} + x_{10,2} + x_{10,3} &= 1 \\
x_{11,2} + x_{11,3} + x_{11,4} &= 1
\end{aligned}
$$



ILP - Příklad

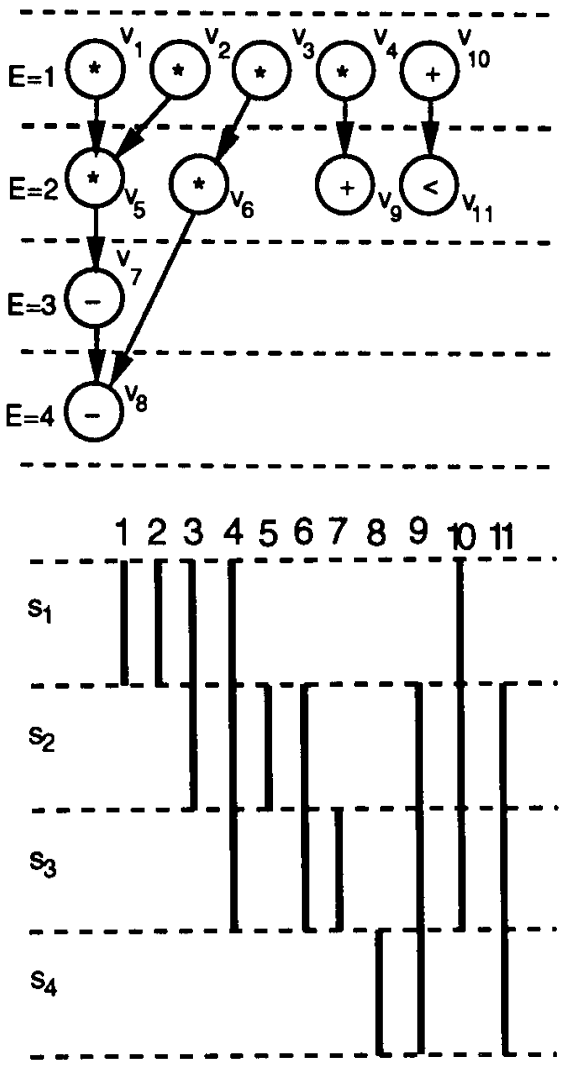
- Omezující podmínky
  2. V každém kontrolním kroku nesmí být naplánováno více operací typu t_k než je dostupné množství N_{tk}

$$x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} \leq N_*$$
$$x_{3,2} + x_{4,2} + x_{5,2} + x_{6,2} \leq N_*$$
$$x_{4,3} + x_{6,3} \leq N_*$$

$$x_{10,1} \leq N_+$$
$$x_{9,2} + x_{10,2} \leq N_+$$
$$x_{9,3} + x_{10,3} \leq N_+$$
$$x_{9,4} \leq N_+$$

$$x_{11,2} \leq N_<$$
$$x_{11,3} \leq N_<$$
$$x_{11,4} \leq N_<$$

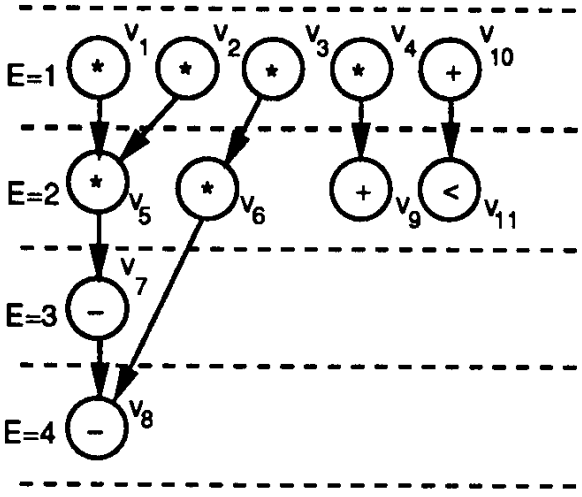
$$x_{7,3} \leq N_-$$
$$x_{8,4} \leq N_-$$

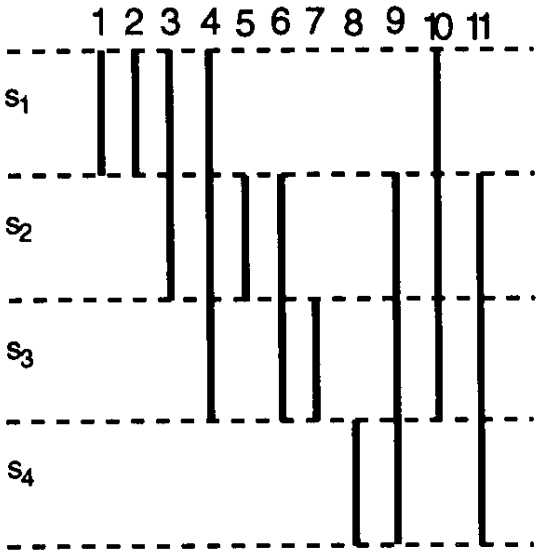


ILP - Příklad

- Omezující podmínky
  3. Operace musí být naplánovány s ohledem na datové závislosti

$$
\begin{aligned}
1x_{3,1} + 2x_{3,2} - 2x_{6,2} - 3x_{6,3} &\leq -1 \\
1x_{4,1} + 2x_{4,2} + 3x_{4,3} - 2x_{9,2} - 3x_{9,3} - 4x_{9,4} &\leq -1 \\
1x_{10,1} + 2x_{10,2} + 3x_{10,3} - 2x_{11,2} - 3x_{11,3} - 4x_{11,4} &\leq -1
\end{aligned}
$$

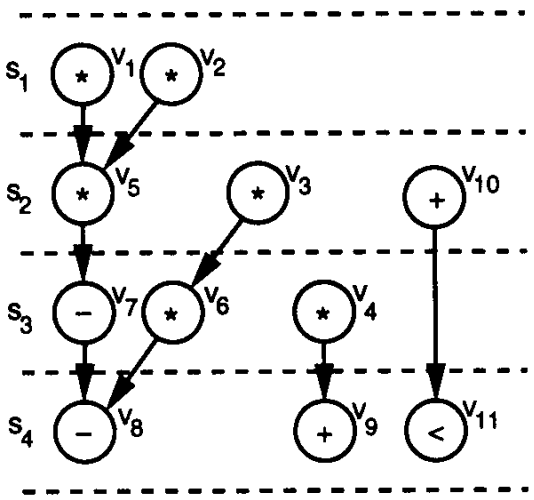




ILP - Příklad

- Minimalizační funkce a množina všech nerovnic sestavených na základě omezujících podmínek reprezentuje vstup pro řešení ILP
- Pro řešení ILP se použije některá ze standardních metod – např. metoda Větví a mezí (nad rámec tohoto kurzu)
- Výsledkem je optimální plán
- Nevýhody ILP
  - S rostoucím počtem kontrolních kroků velmi rychle narůstá množství nerovnic a tím i doba řešení – použitelní pouze pro velmi malé úlohy
  - Pro větší úlohy je nahrazována heuristikami

Výsledný plán



Zdroje:

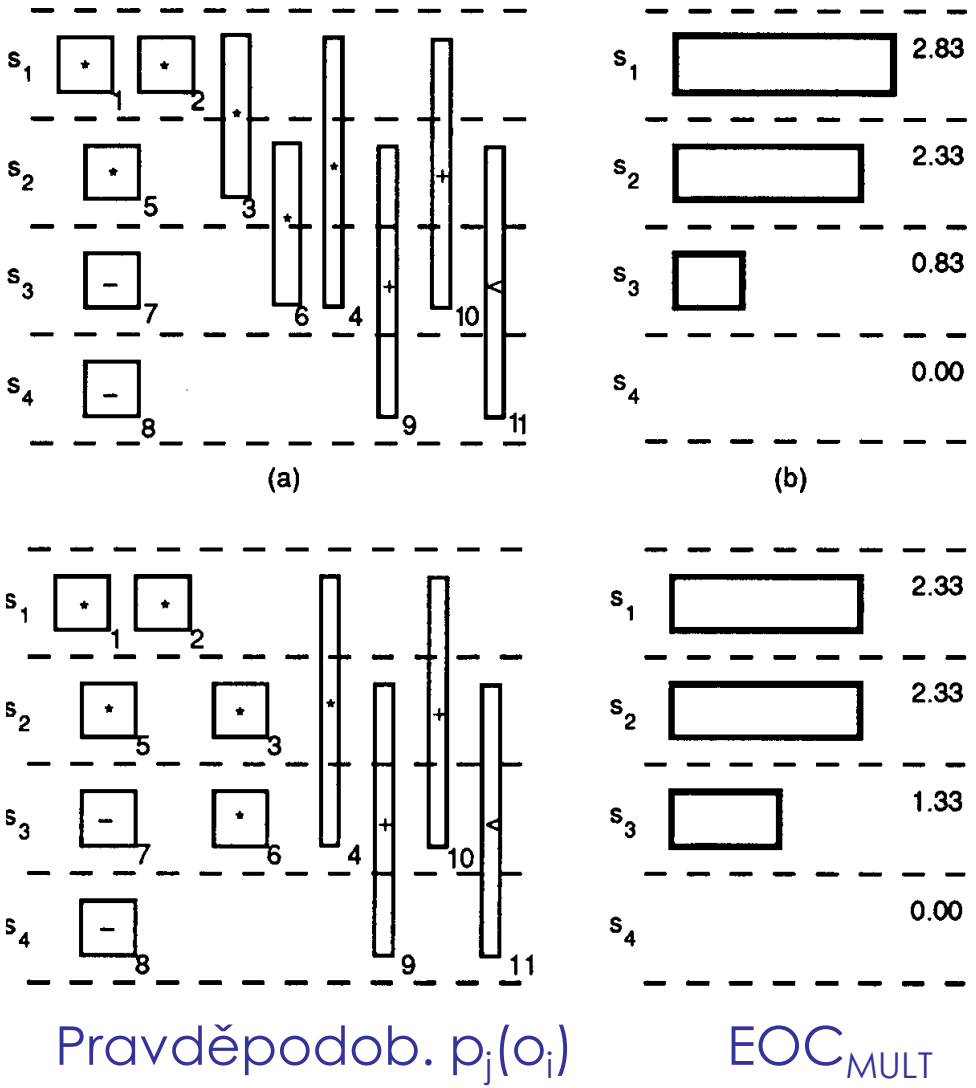
- 2xMULT, 1xADD,
- 1xSUB, 1xCMP

Force-Directed Heuristic

- Cílem je rovnoměrně rozprostřít operátory stejného typu do různých kontrolních kroků
- Princip algoritmu:
  - Pro každý vrchol se vypočte (na základě jeho mobility) pravděpodobnost mapování operace o_i do kontrolního kroku s_j:

$$p_j(o_i) = 1/(L_i - E_i + 1)$$

  - Čím větší mobilita, tím menší pravděpodobnost umístění vrcholu v_i do kroku s_j. Znázorněno na obrázku šířkou obdélníku



Force-Directed Heuristic

- Dále se vypočte očekávaná cena realizace operace *EOC* (Expected Operation Cost) o_i ve stavu s_j:

$$EOC_{j,k} = c_k * \sum_{i, s_j \in mrange(o_i)} p_j(o_i)$$

  - kde c_k je množství zdrojů pro realizaci operace *k* (pro jednoduchost uvažujme c_k=1)
  - vypočtenou cenu zobrazují obdélníkové grafy v pravé části obrázku
- Snahou algoritmu je vyvážit *EOC* přes všechny kontrolní kroky
- Například přesunem vrcholu v_3 do stavu s_2) se upevní pozice vrcholu v_6 do stavu s_3 a zlepší se využití násobičky

Algoritmus: Force-Directed Scheduling

```
Call ASAP (V);
Call ALAP (V);
while there exists o_i such that E_i ≠ L_i do
   MaxGain = −∞;
   /* Try scheduling all unscheduled operations to every */
   /* state in its range */
   for each o_i, E_i ≠ L_i do
      for each j, E_i ≤ j ≤ L_i do
         S_work = SCHEDULE_OP(S_current, o_i, s_j);
         ADJUST_DISTRIBUTION(S_work, o_i, s_j);
         if COST(S_current) - COST(S_work) > MaxGain ther
            MaxGain = COST(S_current) - COST(S_work);
            BestOp = o_i; BestStep = s_j;
         endif
      endfor
   endfor
   S_current = SCHEDULE_OP(S_current, BestOp, BestStep);
   ADJUST_DISTRIBUTION(S_current, BestOp, BestStep);
endwhile
```

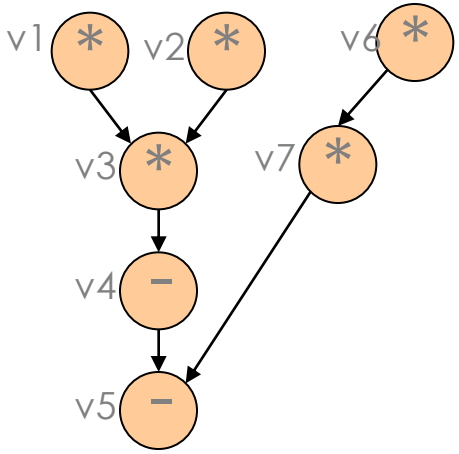
Force-Directed Heuristic

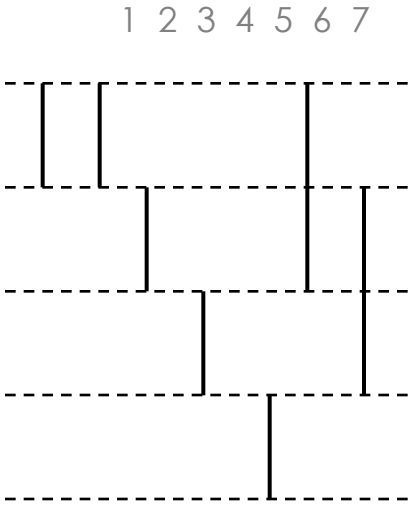
- Postup algoritmu:
  - Algoritmus se v pokouší postupně posouvat doposud nenaplánovaný vrchol do všech možných pozic a vypočítat cenu takto modifikovaného plánu
  - Jakmile toto provede pro všechny nenaplánované uzly, vybere takový posun, který vedl na nejlepší cenu (*MaxGain, BestOp, BestStep*) – vyváženost grafu
  - Vybraný vrchol zafixuje a pokračuje obdobně další iterací, dokud nejsou naplánovány všechny vrcholy
- Použité funkce:
  - SCHEDULE_OP(S,o_i,s_j) - naplánuje operaci o_i do stavu s_j v částečném plánu S
  - ADJUST_DISTRIBUTION(S,o_i,s_j) - přepočítá pravděpodobnosti předchozích a následujících uzlů po provedení fixace operace o_i do stavu s_j
  - MaxGain – hodnota vyjadřující nejlepší rozdíl v ceně oproti současnému plánu
  - COST(S) - vypočte cenu plánu S podle vzorce:

$$COST\ (S) = \sum_{1 \le k \le m} \max_{1 \le j \le s} EOC_{j,k}$$

Force-Directed Heuristic: Příklad

- Příklad:
  - Technikou FDS nalezněte plán s minimálním počtem zdrojů pro zadaný DFG (zjednodušený HAL)
  - Max. doba pro výpočet $T = 4$
  - Cena za operaci násobení $c_* = 10$
  - Cena za operaci odečítání $c_- = 2$





$S_{Current}$	P_*	P_-	EOC_*	EOC_-
	2,5	0	25	0
	2	0	20	0
	0,5	1	5	2
	0	1	0	2

$$COST(S_{Current}) = max[EOC_*] + max[EOC_-]$$
$$= 25 + 2 = 27$$

Force-Directed Heuristic: Příklad

- Celkem 4 možnosti, jak sestavit plán
  1. S_{Work1}: $V_6 \Rightarrow S_1$, $V_7 \Rightarrow \{S_2, S_3\}$
  2. S_{Work2}: $V_6 \Rightarrow S_2$, $V_7 \Rightarrow S_3$
  3. S_{Work3}: $V_7 \Rightarrow S_2$, $V_6 \Rightarrow S_1$
  4. S_{Work4}: $V_7 \Rightarrow S_3$, $V_6 \Rightarrow \{S_1, S_2\}$

S_{Work1}	P_*	P_-	EOC_*	EOC_-
* * *	3	0	30	0
*	1,5	0	15	0
- *	0,5	1	5	2
-	0	1	0	2

$COST(S_{Work1}) = max[EOC_*] + max[EOC_-]$

$= 30 + 2 = 32$

$MaxGain = COST(S_{Current}) - COST(S_{Work3}) =$

$= 27 - 32 = -5$

Force-Directed Heuristic: Příklad

S_{Work2}	P_*	P_-	EOC_*	EOC_-
* *	2	0	20	0
* *	2	0	20	0
- *	1	1	10	2
-	0	1	0	2

$$COST(S_{Work2}) = max[EOC_*] + max[EOC_-]$$
$$= 20 + 2 = 22$$
$$MaxGain = COST(S_{Current}) - COST(S_{Work2}) =$$
$$= 27 - 22 = 5$$

S_{Work3}	P_*	P_-	EOC_*	EOC_-
* * *	3	0	30	0
* *	2	0	20	0
-	0	1	0	2
-	0	1	0	2

$$COST(S_{Work4}) = max[EOC_*] + max[EOC_-]$$
$$= 30 + 2 = 32$$
$$MaxGain = COST(S_{Current}) - COST(S_{Work3}) =$$
$$= 27 - 32 = -5$$

Force-Directed Heuristic: Příklad

S_{Work4}	P_*	P_-	EOC_*	EOC_-
* * *	2,5	0	25	0
*	1,5	0	15	0
- *	1	1	10	2
-	0	1	0	2

$$COST(S_{Work4}) = max[EOC_*] + max[EOC_-]$$
$$= 25 + 2 = 27$$
$$MaxGain = COST(S_{Current}) - COST(S_{Work3}) =$$
$$= 27 - 27 = 0$$

- Výsledek první iterace:
  - Nejlepšího výsledku dosahuje plán: S_{Work2}: $V_6 \Rightarrow S_2$, $V_7 \Rightarrow S_3$
  - Po aplikaci tohoto kroku získáme navíc plán, jehož všechny operace jsou již přiřazeny do konkrétních kontrolních kroků $\Rightarrow$ konec algoritmu
- Nevýhody FDH přístupu:
  - Jakmile je operace naplánována do konkrétního kontrolního kroku, nemůžeme toto rozhodnutí vrátit zpět
  - Existuje řada variant FDH, která je schopna sledovat několik kroků do budoucnosti

Další algoritmy

- Iterative refinement
  - Nejprve se vygeneruje počáteční plán
  - V následujícím kroku se každá operace zkusí postupně naplánovat do všech přípustných stavů a vybere se takový posun, který je pro ni nejvýhodnější
  - Takto modifikovaný plán se vezme jako počáteční plán pro další iteraci a výpočet se opakuje tak dlouho, dokud jsou získávány lepší výsledky

Prostorově omezené plánování

- Formulace problému:
  - Je zadán maximální počet dostupných zdrojů pro realizaci algoritmu zadaného pomocí DFG
  - Cílem je nalézt takový plán, který bude obsahovat co nejméně kontrolních kroků
- Množství dostupných zdrojů zadáno ve formě:
  1. Konkrétního počtu funkčních jednotek
     - Příklad použití: procesory VLIW
     - Lze efektivně použít např. List Based Scheduling
  2. Plochy čipy resp. počtu základních logických hradel
     - Příklad použití: technologie FPGA
     - Algoritmus musí navíc prozkoumat, které kombinace funkčních jednotek vedou na nejrychlejší výpočet
     - NP těžký problém – ILP popř. heuristika

List-Based Scheduling

- Jedná se o zobecněnou verzi algoritmu ASAP
- Princip algoritmu:
  - Pro jednotlivé operátory se v každém kroku udržuje seznam vrcholů, které je možné naplánovat – tj. uzly, jejichž všichni předci jsou již naplánováni
  - Vrcholy v tomto seznamu se uspořádají podle priority
  - Ze seznamu se vyberou prvky s nejvyšší prioritou a ty se naplánují na dostupné zdroje
  - Plánováním vznikají nové uzly, která se vloží do seznamů připravených vrcholů a výpočet se opakuje tak dlouho, dokud není celý graf naplánován

```
INSERT_READY_OPS(V, PList_t1, PList_t2, ... PList_tm);
Cstep = 0;
while((PList_t1 ≠ φ) or ... or (PList_tm ≠ φ)) do
   Cstep = Cstep + 1;
   for k = 1 to m do
      for funit = 1 to N_k do
         if PList_tk ≠ φ then
            SCHEDULE_OP(S_current, FIRST(PList_tk),Cstep);
            PList_tk = DELETE(PList_tk, FIRST(PList_tk));
         endif
      endfor
   endfor
   INSERT_READY_OPS(V, PList_t1, ..., PList_tm );
endwhile
```

- Příklad:
  - řešení diferenciální rovnice
  - dostupné zdroje: 2xMULT, ADD, SUB, CMP
  - prioritní funkce: mobilita

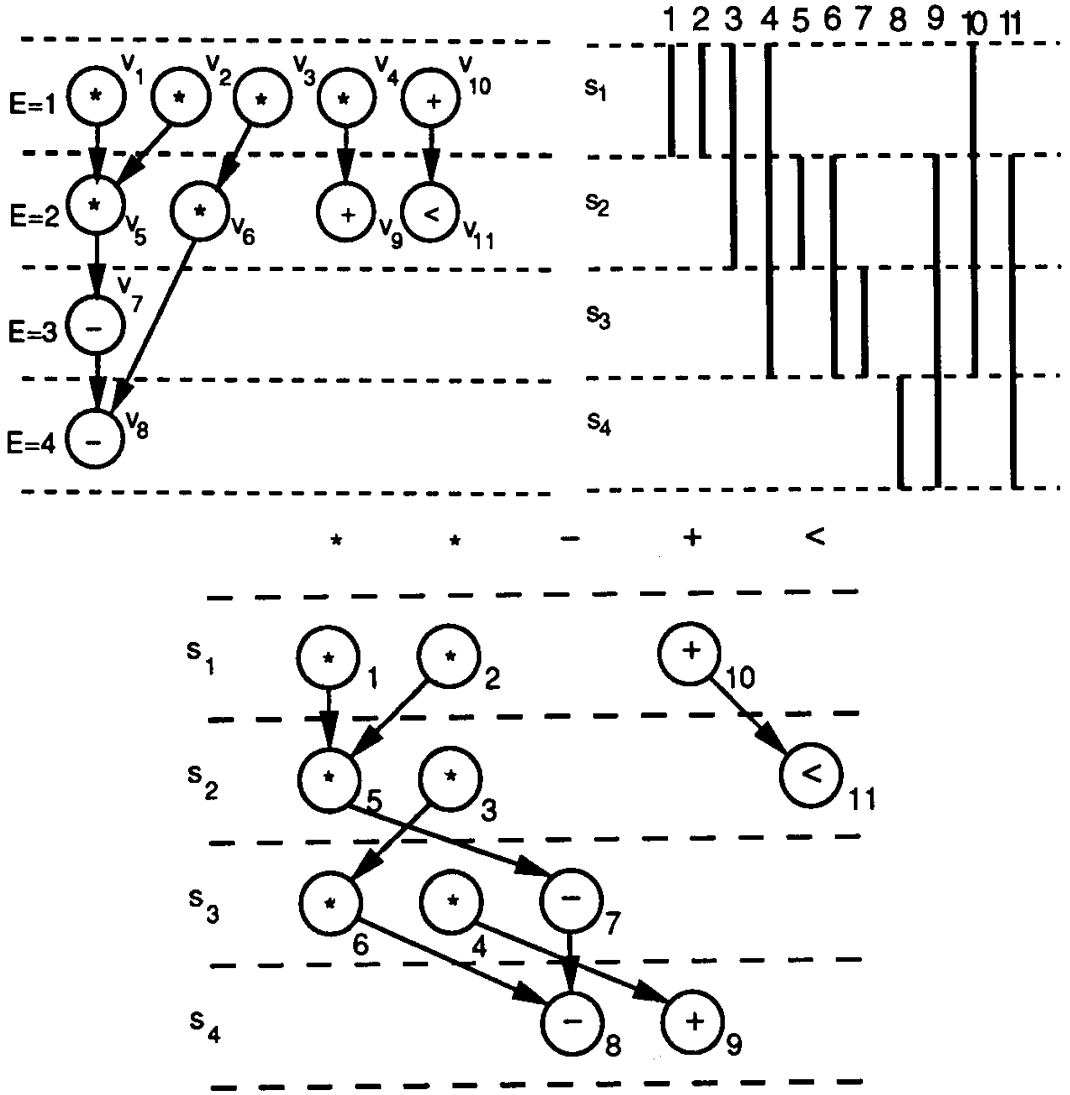
List-Based Scheduling - příklad

```
Prioritní seznamy:
PList* : 1<1>, 2<1>, 3<2>, 4<3>
PList+ : 10<3>
PList- : NIL
PList> : NIL
```

```
Prioritní seznamy:
PList* : 5<1>, 3<1>, 4<2>,
PList+ : NIL
PList- : NIL
PList> : 11<3>
```

```
Prioritní seznamy:
PList* : 6<1>, 4<1>
PList+ : NIL
PList- : 7<1>
PList> : NIL
```

```
Prioritní seznamy:
PList* : NIL
PList+ : 9<1>
PList- : 8<1>
PList> : NIL
```

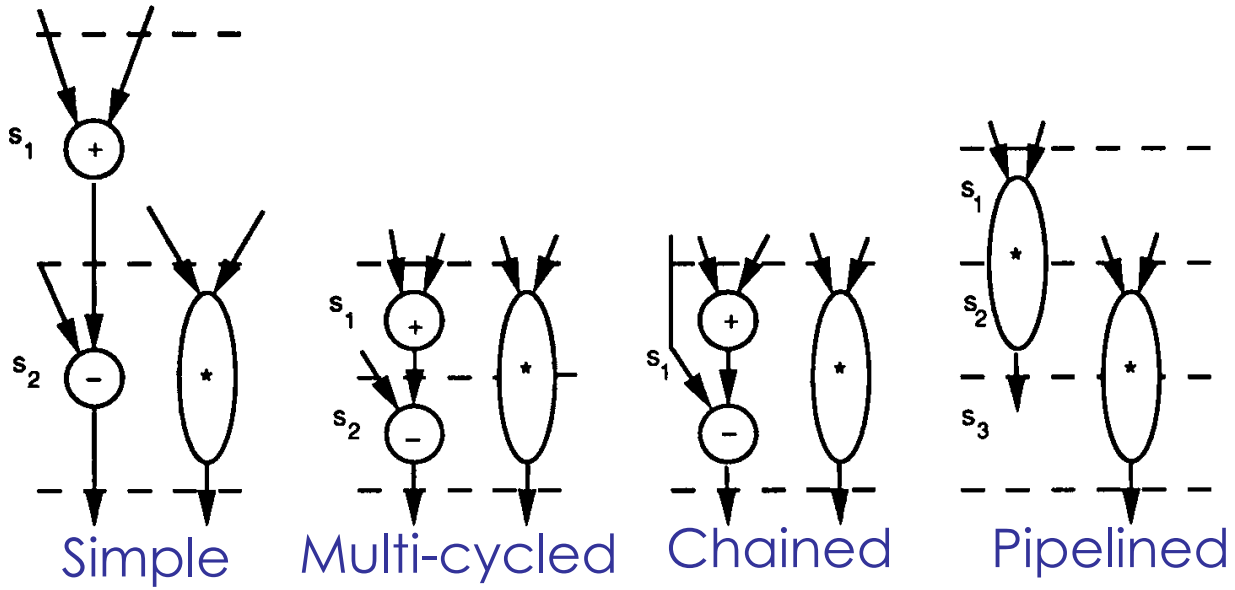


List-Based Scheduling

- Kvalita plánovací funkce závisí především na zvolené prioritní funkci
- V předchozím příkladě byla použita prioritní funkce, která upřednostňovala vrcholy s nižší mobilitou – mají menší možnosti s ohledem na další plánování
- Alternativní prioritní funkce:
  - Podle nejdelší cesty, která vede od vrcholu až do konce grafu. Upřednostněny jsou vrcholy, které mají delší cestu ke konci grafu – je potřeba je naplánovat co nejdříve
  - Podle toho kolik má daný vrchol následovníků. Upřednostněny jsou vrcholy, které mají více následovníků

Plánování reálných obvodů

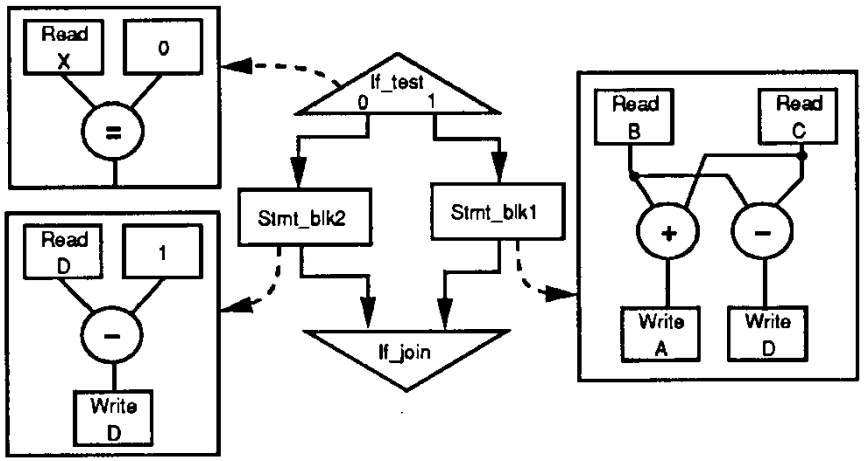
- Při plánování reálných obvodů je potřeba uvažovat mnohem složitější případy než mapování každého vrcholu do jednoho kontrolního kroku
- Vyhodnocení některých operací trvá déle (např. násobička). Výpočet násobičky je mapován do dvou kontrolních kroků (multi-cycled)
- V opačném případě mohou existovat operátory, které jsou vyhodnoceny velmi rychle a lze je sdružovat do jednoho kontrolního kroku (chained)
- Některé jednotky mohou být vnitřně zřetězeny a vstupní data tak mohou být do těchto jednotek zasílána v každém kontrolním kroku (pipelined)



Plánování podmíněných sekcí

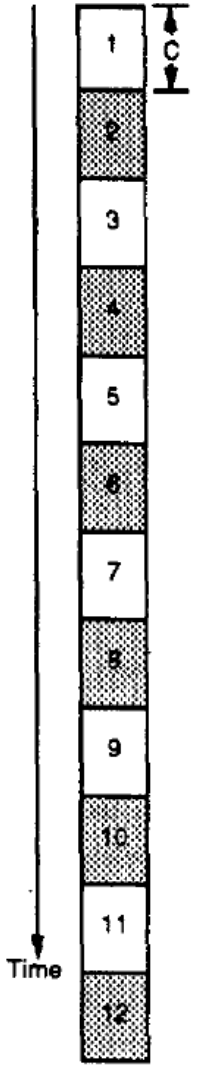
- Plánovací algoritmy aplikované na reálné úlohy musí uvažovat i s větvením programu
- Výpočet v jedné podmíněné sekci je vzájemně vyloučen s výpočtem v druhé sekci
- ⇒ Zdroje lze mezi sekcemi sdílet

```
if (X = 0) then
   A := B + C;
   D := B - C;
else
   D := D - 1;
end if;
```



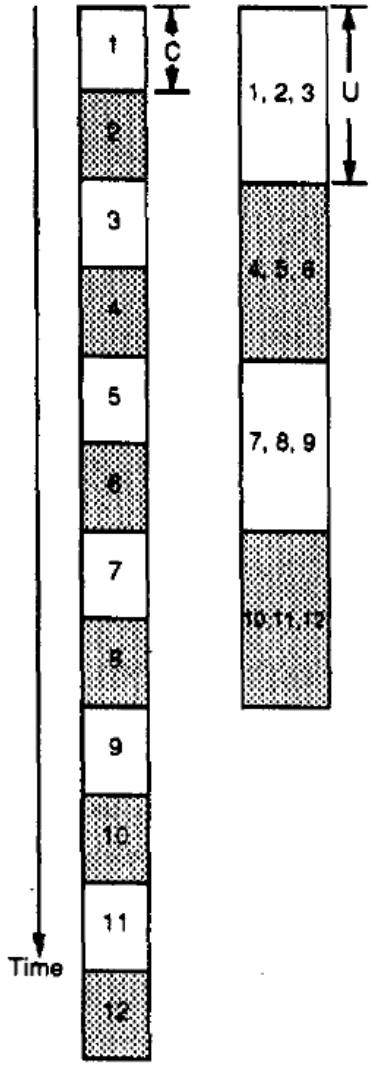
Plánování smyček

- Sekvenční zpracování
  - Jednotlivé iterace jsou zpracovávány postupně
  - Příklad:
    - Počet iterací: *12*
    - Délka jedné iterace: *C* kontrolních kroků
    - Celkový čas *12C*



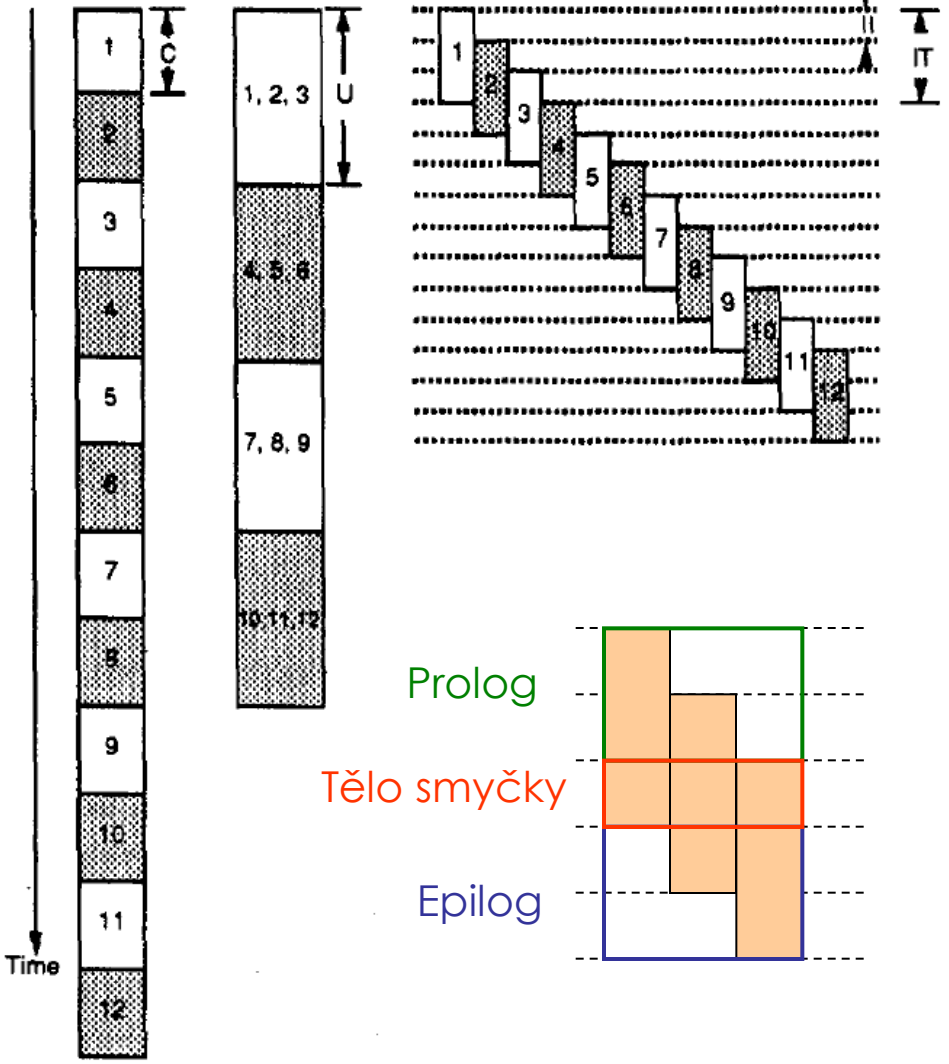
Plánování smyček

- Rozbalování smyček (Loop unrolling)
  - Určitý počet iterací je rozbalen do tzv. superiterace, které se provedou sekvenčně
  - Příklad:
    - 3 iterace jsou rozbaleny do superiterace
    - Celkem provedeny 4 superiterace
    - Každá superiterace naplánována do *U* kontrolních kroků
    - Pokud *U* < *3C*, potom je celková doba výpočtu lepší než *12C*
  - Aplikovatelné, pokud je dopředu znám počet iterací



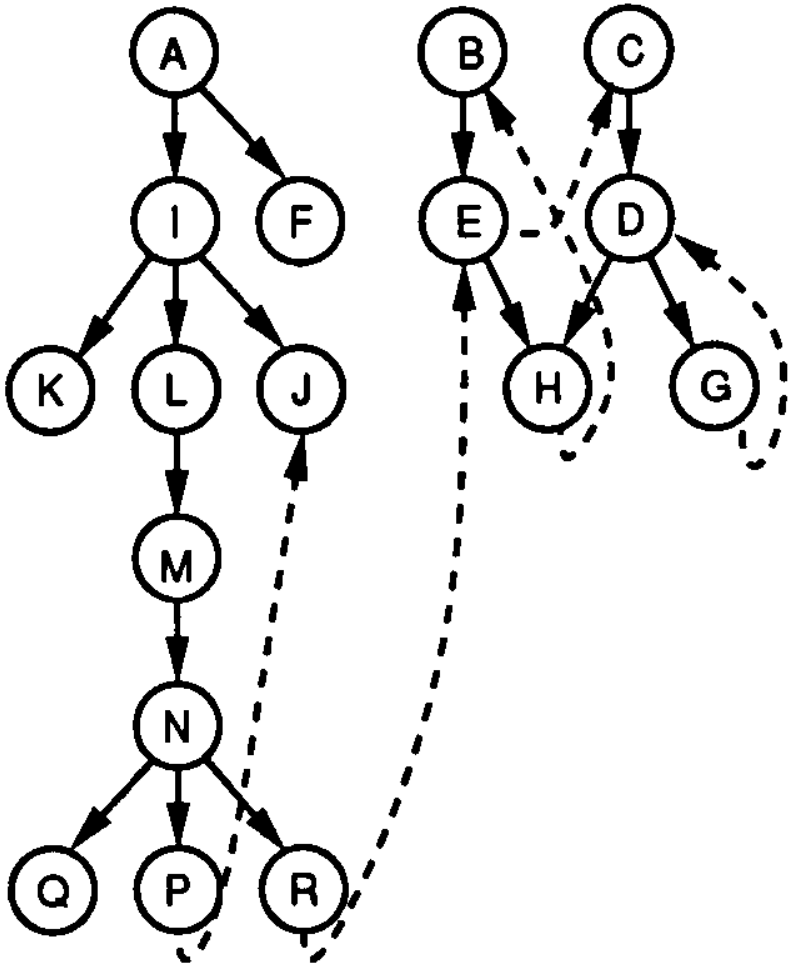
Plánování smyček

- Skládání smyček (Loop folding)
  - Těla iterací jsou překryty ve smyslu zřetězeného zpracování
  - Plán se dělí na tělo smyčky (Fold), inicializační (Prolog) a závěrečnou fázi (Epilog)
  - Tělo smyčky je dlouhé *II* kontrolních a může být aplikováno vícekrát
  - Příklad:
    - Délka iterace: *IT* kontrolních kroků (*IT* – Iteration Time)
    - Výpočet nové iterace začíná každých II kroků, kde *II<IT* (*II* - Inicialization Interval)
    - Celkový čas výpočtu je *IT+(n-1)×II* kontrolních kroků
  - Aplikovatelné i v případě, že není dopředu znám počet iterací



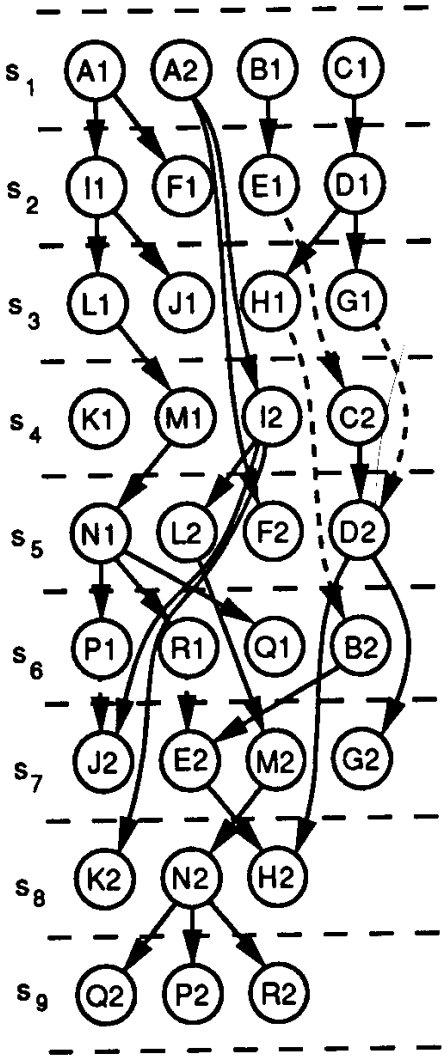
Rozbalování vs. Skládání smyček

- Příklad:
  - jednoduchý graf zahrnující 17 stejných operací
  - Plné šipky vyjadřují datové závislosti v rámci iterace
  - Přerušované šipky vyjadřují datové závislosti mezi iteracemi
  - Doba jedné iterace je 6 kroků
- Výsledek se bude posuzovat dle:
  - Využitelnost dostupných zdrojů
  - Počtu kontrolních kroků v přepočtu na jednu iteraci



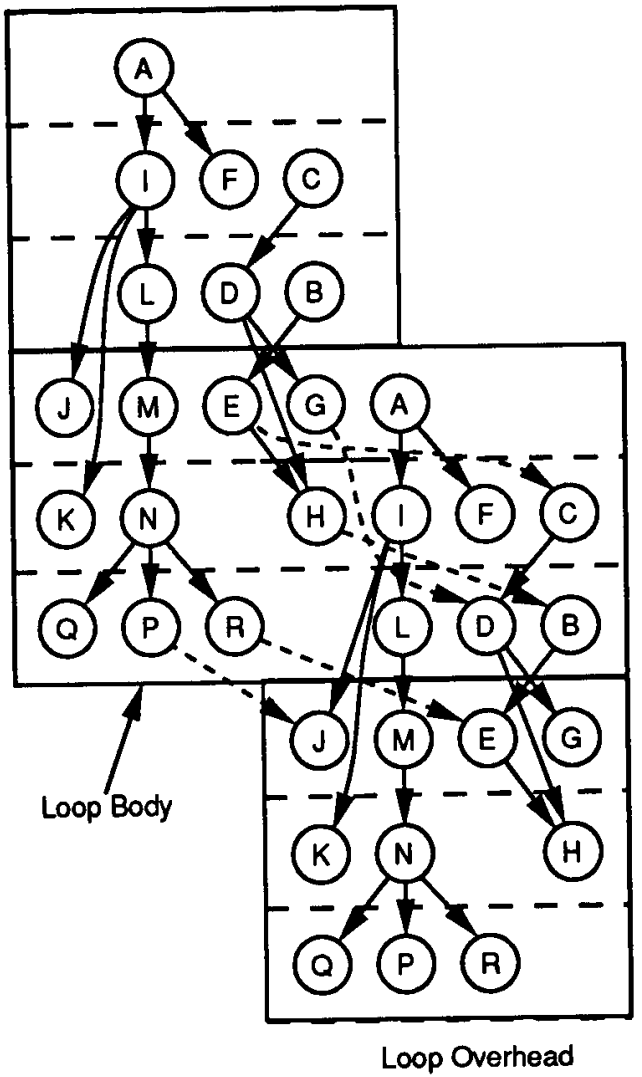
Rozbalování vs. Skládání smyček

- Rozbalování smyček
  - DFG jedné iterace je rozbalen ještě jednou
  - Výsledný DFG je dále plánován běžnými technikami
  - Graf je větší/složitější – plánování trvá delší dobu
  - Lze aplikovat pouze v případě známého počtu operací
- Příklad:
  - Využití operátorů:
    - (17x2)/(4x9) = 17/18
  - Počet kroků na iteraci: 9/2 = 4.5



Rozbalování vs. Skládání smyček

- Skládání smyček
  - Jednotlivé části iterace jsou překryty podobně jako u pipeliningu
  - Plán se dělí na tělo smyčky, inicializační a závěrečnou fázi
  - Překrytí závisí na datových závislostech
  - Lze aplikovat i v případě, že není dopředu znám počet iterací
- Příklad:
  - Využití operátorů:
    - (5+6+6)/(6x3) = 17/18
  - Počet kroků na iteraci: 3

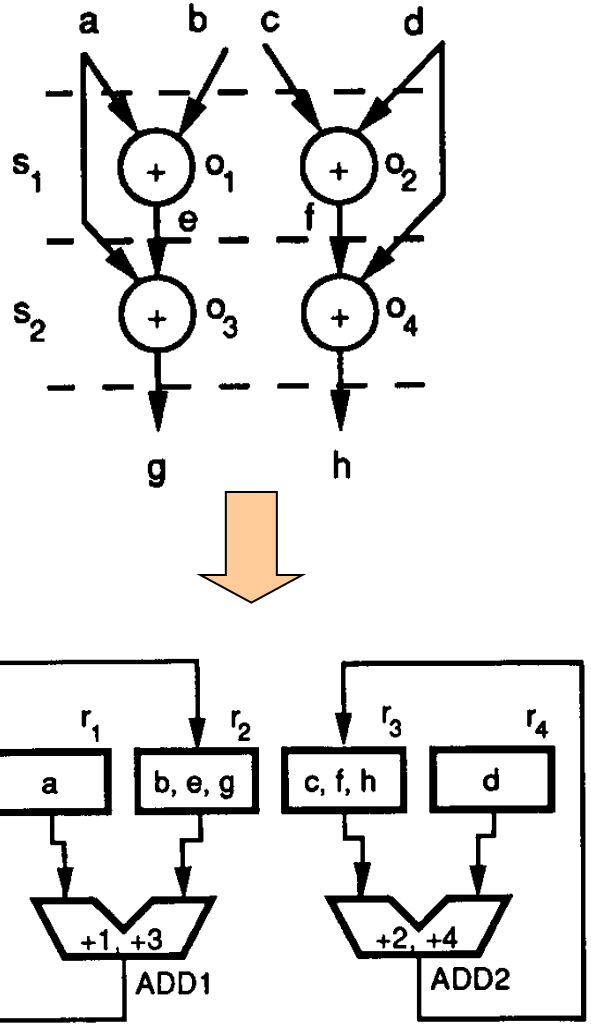


Osnova

- Úvod
- Reprezentace obvodu
- Základní transformace
- Plánování
- Přiřazení
- Alokace
- Generování RTL
- Shrnutí

Přiřazení zdrojů

- Cílem alokace zdrojů je mapovat proměnné a operace naplánovaného CDFG grafu do funkčních jednotek, registrů a propojovacích sítí
- Příklad:
  - jsou využity 2 sčítačky pracující paralelně
  - pro uložení proměnných jsou použity 4 registry
  - operace o_1, o_3 resp. o_2, o_4 mapovány do ADD1 resp. ADD2
  - alternativně lze mapovat operace o_1, o_4 resp. o_2, o_3 do ADD1 resp. ADD2



Přiřazení zdrojů

- Alokace zdrojů se skládá ze tří hlavních částí:
  1. Přiřazení funkčních jednotek
  2. Přiřazení paměťových elementů
  3. Přiřazení propojovacích sítí

- Přiřazení funkčních jednotek:
  - pokud je k dispozici pouze jedna funkční jednotka daného typu, potom je operace z DFG mapována na tuto jednotku
  - pokud je k dispozici více jednotek stejného typu, musí algoritmus přiřazení vybrat nejvhodnější kombinaci (viz. předchozí příklad operace se dvěmi sčítačkami)

Přiřazení zdrojů

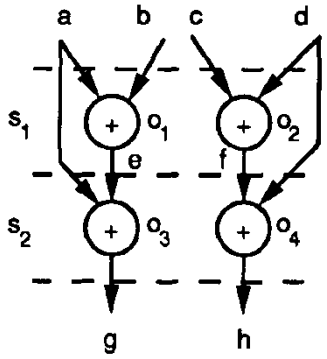
- Přiřazení paměťových elementů
  - mapuje konstanty, proměnné, datové struktury a pole do pamětových elementů (RAM, ROM, register)
  - konstanty jsou uloženy do paměti typu ROM
  - proměnné jsou uloženy v registrech nebo pamětech RAM
  - na základě doby života proměnné může být sdílen registr (nebo paměťová pozice) mezi více proměnnými
  - doba života - je interval (v počtu kontrolních kroků) mezi prvním a posledním použitím proměnné
  - registry mohou být sdružovány do registrových polí, v případě, že se k nim přistupuje v oddělených kontrolních krocích
  - alternativně lze používat registrová pole s více porty

Přiřazení zdrojů

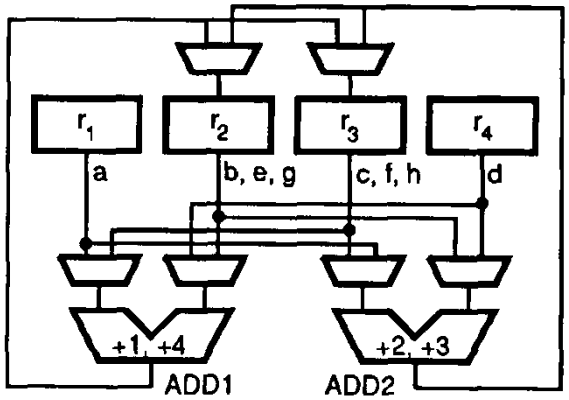
- Přiřazení propojovacích sítí
  - každý datový přenos je potřeba realizovat pomocí propojení zdrojového a cílového uzlu
  - tyto propojovací cesty mohou být sdíleny několika nezávislými přenosy (neprobíhají současně)
  - cílem algoritmu je co nejvíce sdílet propojovací cesty a snížit tak cenu propojovací sítě

Přiřazení zdrojů

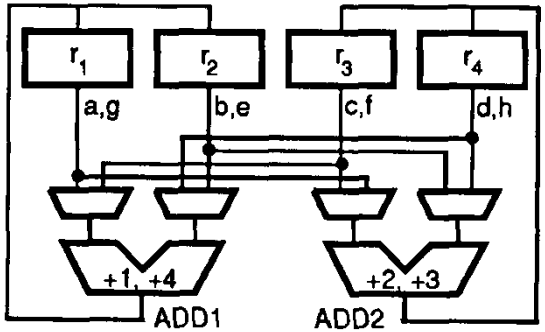
- Závislost jednotlivých částí alokace
  - Přiřazení jednotek, paměti i propojovacích sítí se navzájem ovlivňuje
  - Složitost propojovací sítě závisí na přiřazení jednotek a proměnných (a naopak)



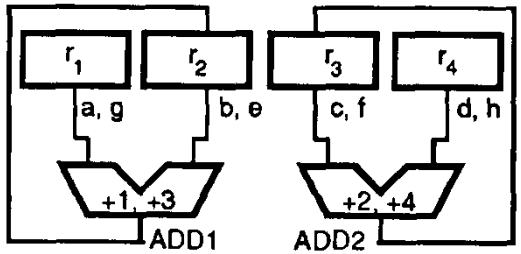
(a)



(b)



(c)

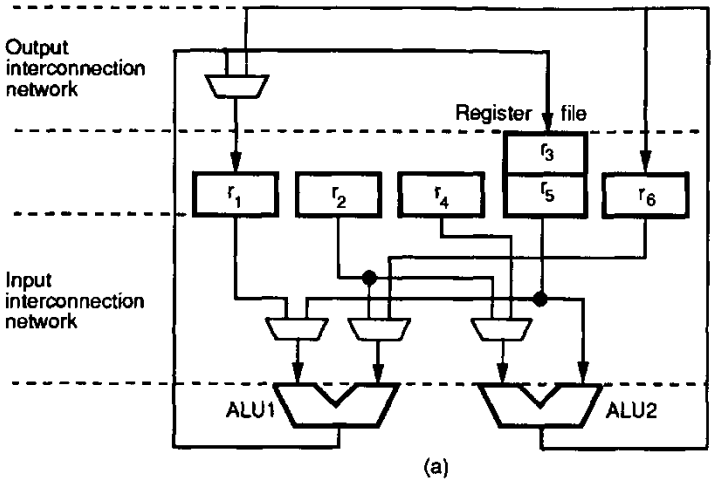


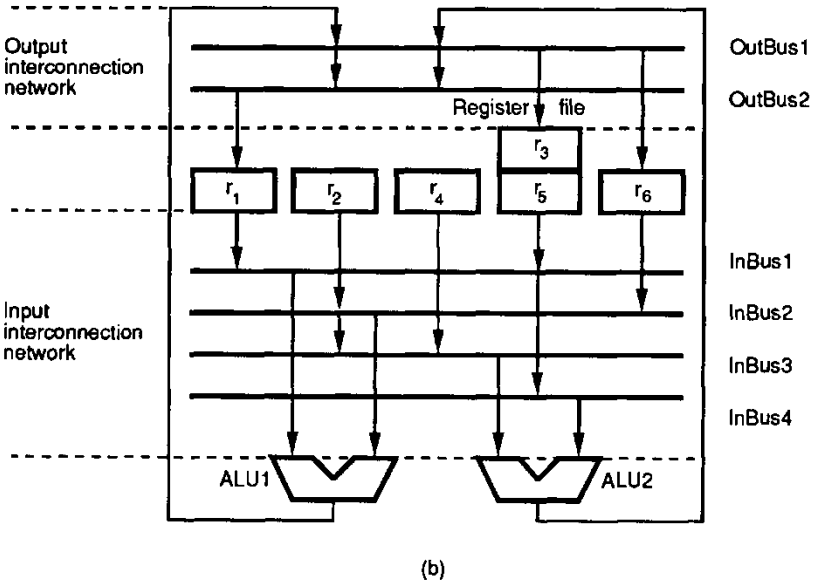
(d)

Přiřazení zdrojů

- Základní schéma :
  - Paměťové elementy
  - Funkční jednotky
  - Vstupní propojovací síť
  - Výstupní propojovací síť
- Typy propojovacích sítí:
  - Závisí na použité technologii
  1. Multiplexované
  2. Sběrnicové (třístavové)
- Příklad:

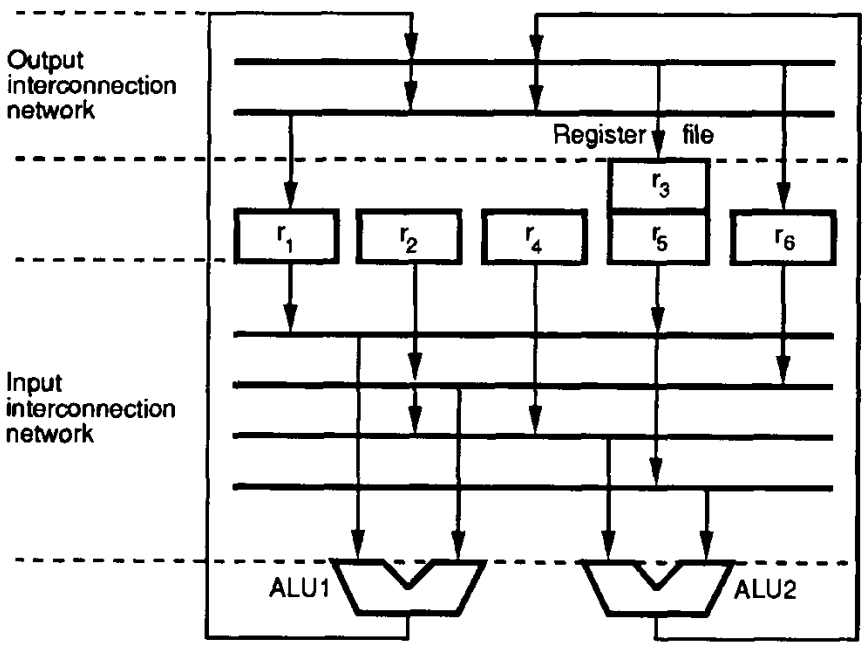
```
s1: r3=ALU1(r1,r2); r1=ALU2(r3,r4);
s2: r1=ALU1(r5,r6); r6=ALU2(r2,r5);
s3: r3=ALU1(r1,r6);
```

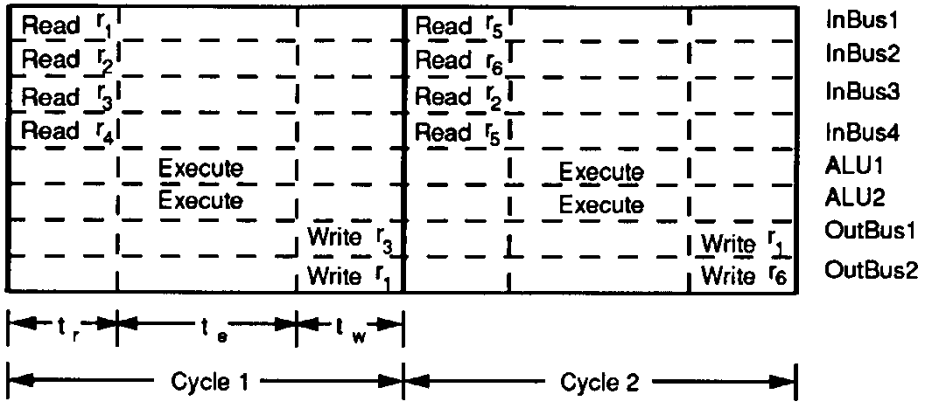




Přiřazení zdrojů

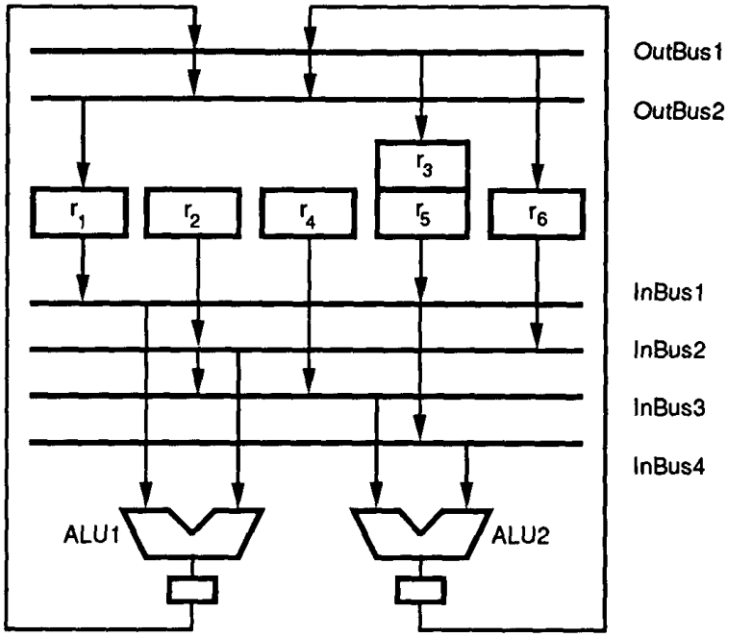
- Každý krok výpočtu se skládá ze tří částí:
  1. Načtení operandu z registru
  2. Výpočet
  3. Zápis výsledku do registru
- Vstupní i výstupní propojovací síť mají určité zpoždění podobně jako zápis a čtení do/z registru
- Doba výpočtu jednoho kroku je: $t = t_r + t_e + t_w$, kde
  - t_r – doba pro čtení vstupu
  - t_e – doba výpočtu
  - t_w – doba pro zápis výsledku

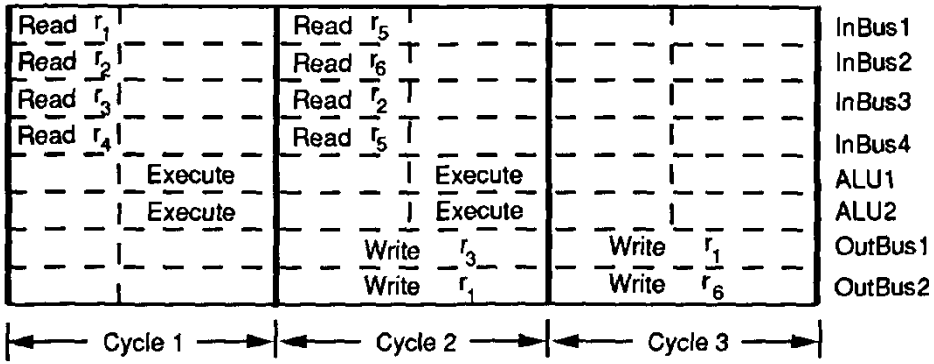




Přiřazení zdrojů

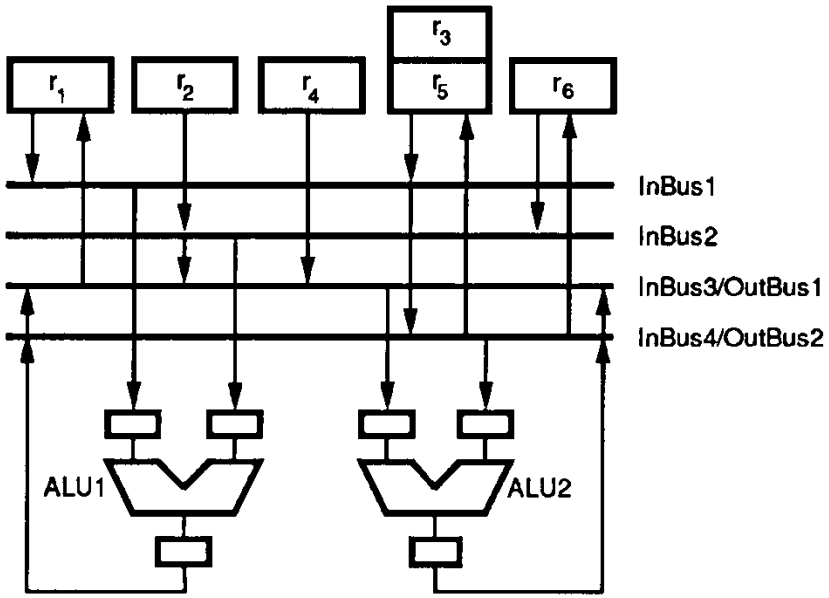
- Zrychlení výpočtu lze dosáhnout vložením registru za ALU (pipelining)
- Výpočet se rozdělí na dvě oddělené fáze:
  1. Načtení operandů, Výpočet
  2. Zápis výsledku
- Doba výpočtu jednoho kroku je: $t = \max(t_r+t_e, t_w)$
- Je potřeba dát pozor na datové závislosti, tj. čtení operandu, který byl vypočten v předchozím kroku
- Alternativně registr před ALU

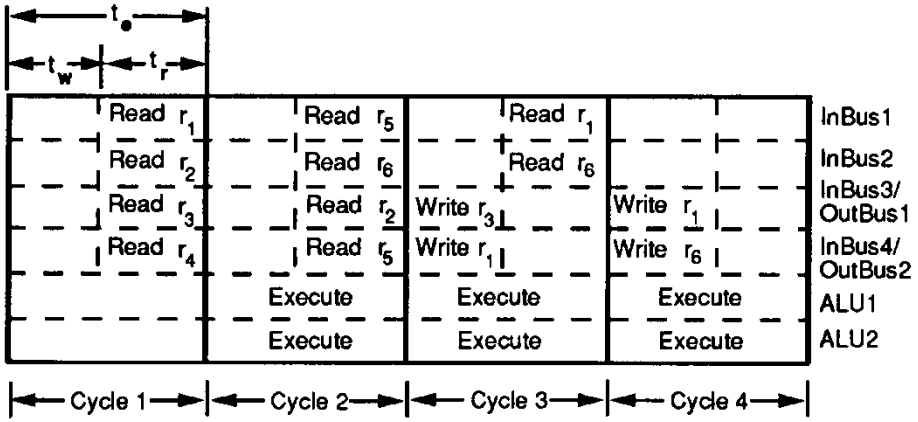




Přiřazení zdrojů

- Další zrychlení lze dosáhnout vložením registru před i za ALU
- Výpočet se rozdělí na tři oddělené fáze:
  1. Načtení operandů
  2. Výpočet
  3. Zápis výsledku
- Čtení a zápis operandu prováděny v rámci jednoho taktu hodin (zápis v první polovině, čtení v druhé)
- Lze sdílet vstupní a výstupní propojovací síť – menší cena
- Doba výpočtu jednoho kroku je: $t = \max(t_r+t_w, t_e)$
- Rovněž je potřeba dát pozor na datové závislosti

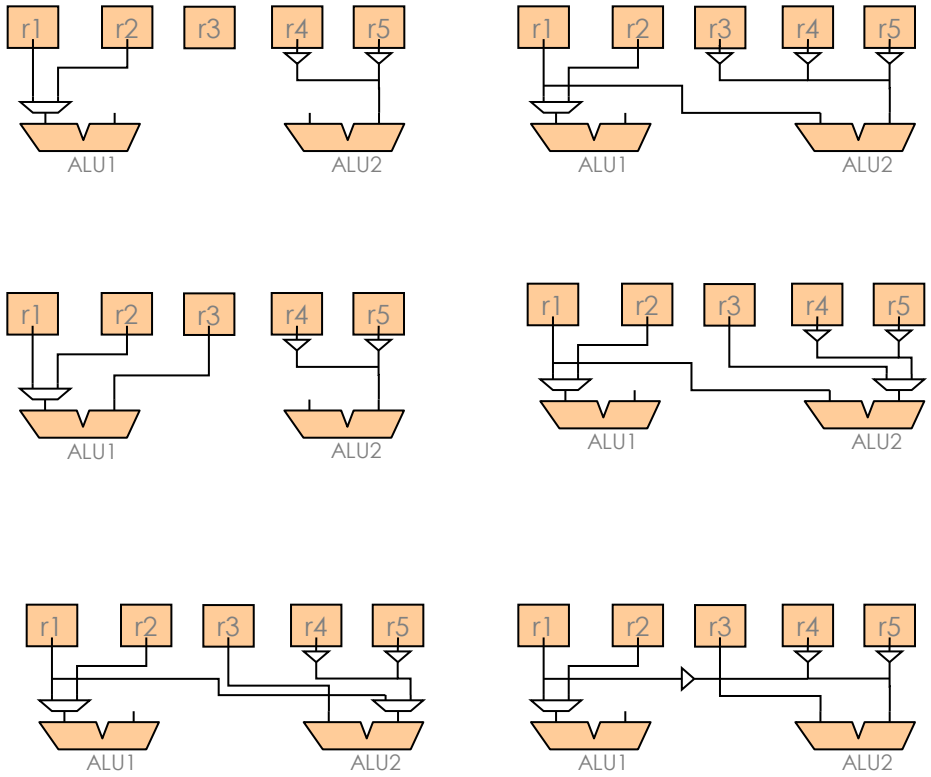




Konstruktivní přístup

- Základní princip:
  - Vychází z prázdného schématu a postupně přidává funkční jednotky, paměti a vodiče
  - Pro každou operaci hledá vhodnou funkční jednotku
  - Pokud je více možností, vybere tu, která vede nižší cenu vodičů
  - Pokud není k dispozici FU, alokuje novou
  - Podobně jsou přiřazovány proměnné do registru na základě jejich doby života

- Příklad:
  - přidání operace ADD r1, r3

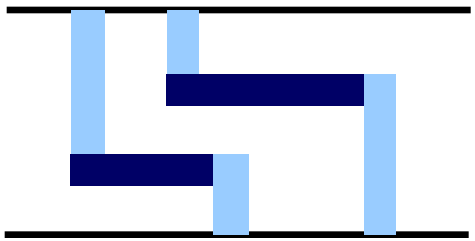


Konstruktivní přístup

- Funkce ceny zavisí na cílové technologii (např. rozhoduje, zda je výhodnější multiplexovaná nebo třístavová sběrnice)
- Rozlišujeme mezi přístupy:
  - Statický - pořadí mapování entit je dáno předem a v průběhu výpočtu se nemění
  - Dynamický - pořadí mapování se mění a v každém kroku se vybere entita, která je nejvýhodnější
- Nevýhody:
  - Nedává moc dobré výsledky
  - Sdílení jednoho registru/vodiče se může zdát v jednom kroku výhodné, zatímco v některém z dalších kroků už nikoliv
  - Do ohodnocující funkce je dobré přidat faktor predikce

Left-Edge algoritmus

- Problém channel-routing:
  - uvažujme kanál a body, které jsou umístěny na protějších březích kanálu
  - aby bylo možné propojit libovolné dva body, jsou potřeba alespoň dva vertikální a jeden horizontální spoj
  - cílem je propojit zadané dvojice bodů s minimem horizontálních spojů (minimální šířka kanálu)



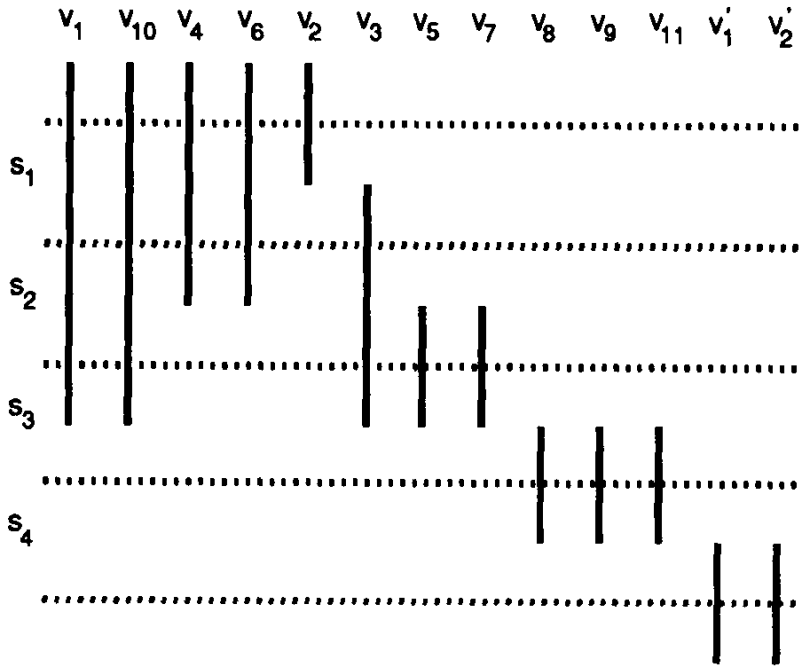
- Aplikace algoritmu na úlohu přiřazení registrů
  - každá proměnná v DFG má svoji dobu života a je mapována do některého z registrů
  - registr reprezentuje horizontální spoj a doba života zabírá část tohoto horizontálního spoje
  - cílem je namapovat všechny proměnné do co nejmenšího počtu registrů (minimalizace počtu horizontálních spojů)

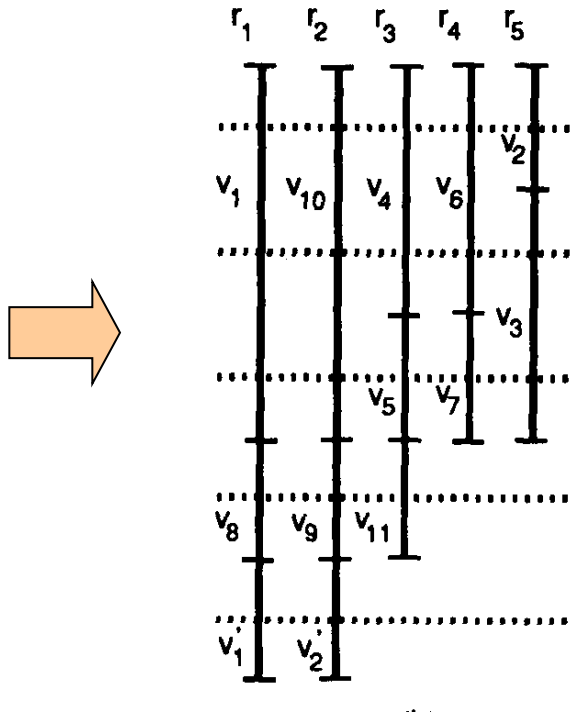
- Princip algoritmu:
  - proměnné se seřadí do seznamu podle:
    - primární klíč: začátek jejich doby života (vzestupně)
    - sekundární klíč: konec jejich doby života (sestupně)
  - v každém kroku iterace:
    - alokuje se jeden nový registr
    - prochází se seznamem proměnných a první nepřekrývající-se výskyty proměnných se do tohoto registru namapují
    - namapované proměnné se odstraní ze seznamu

```
for all v ∈ L do  MAP[v] = 0;  endfor
SORT(L);

reg_index = 0;
while L ≠ φ do
    reg_index = reg_index + 1;
    curr_var = FIRST(L);
    last = 0;

    while curr_var ≠ null do
        if Start(curr_var) ≥ last then
            MAP[curr_var] = r_reg_index;
            last = End(curr_var);

            temp_var = curr_var;
            curr_var = NEXT(L, curr_var);
            DELETE(L, temp_var);
        else
            curr_var = NEXT(L, curr_var);
        endif
    endwhile

endwhile
```

Left-Edge algoritmus

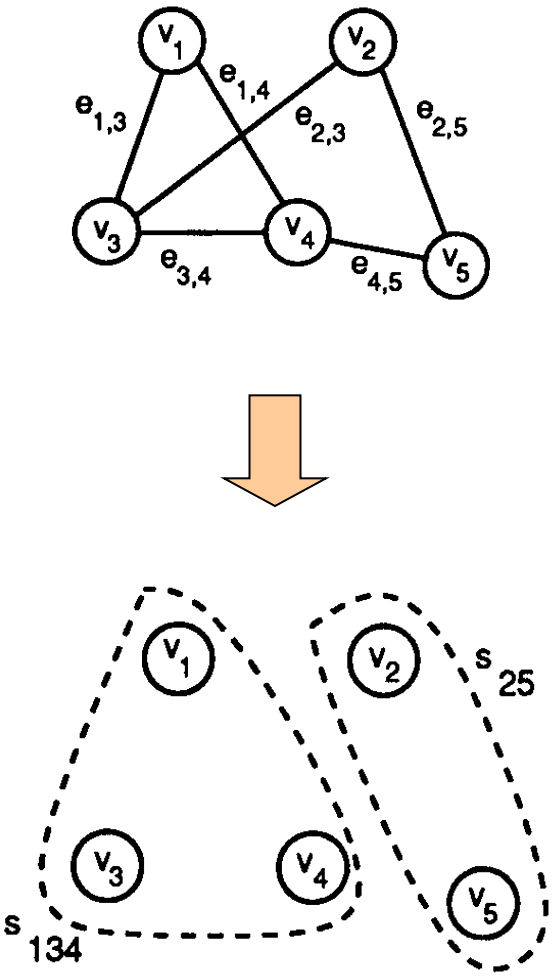
- Poznámky:
  - reg_index – číslo alokovaného registru
  - MAP[n] – pole, které přiřazuje každé proměnné registr, do kterého bude mapována
- Výhody:
  - velmi rychlý algoritmus
  - zaručuje optimální využití registrů
- Nevýhody:
  - nebere v úvahu ostatní části úlohy alokace





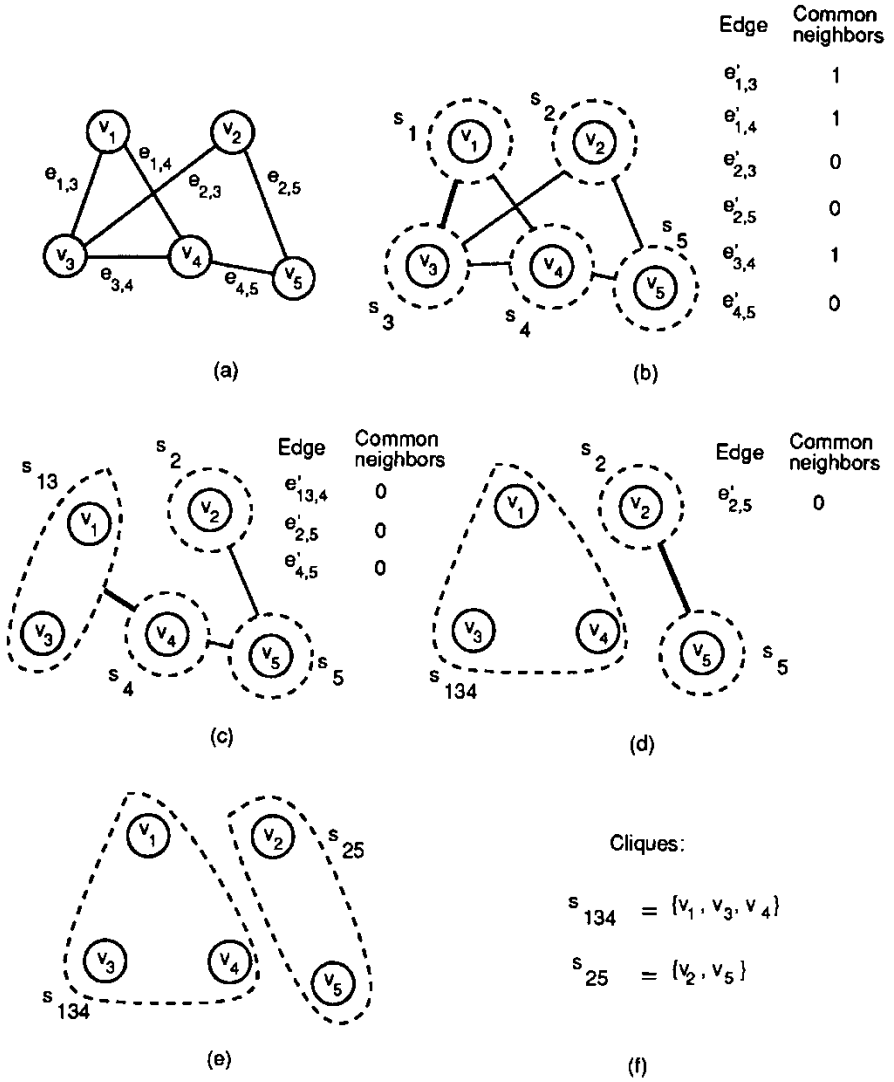
Dekompozice grafu na kliky

- Rozdělení grafu na kliky
  - jedná se o obecnou úlohu z teorie grafů
  - Úplný graf - je takový graf jehož všechny vrcholy jsou propojeny navzájem (každý s každým)
  - Klika - je úplný pod-graf grafu G(V,E)
  - Rozdělení grafu na kliky – je problém, jehož cílem je rozdělit graf G(V,E) na minimální počet klik
  - Obecně se jedná o NP-úplný problém – nutno řešit heuristikou



Dekompozice grafu na kliky

- Princip heuristiky
  - Cílem je změnit graf G(V,E) na super-graf G'(S,E'), kde S jsou vrcholy odpovídající klikám grafu G
  - Vychází se ze stavu, kdy G' odpovídá grafu G
  - V každém kroku se pro všechny dvojice vrcholu vypočte množina společných sousedů
  - Dva vrcholy, které mají nejvíce společných sousedů, se spojí do jednoho vrcholu
  - Všechny hrany vycházející s těchto vrcholů se odstraní a vytvoří se nové hrany propojující vzniklý vrchol se všemi společnými sousedy

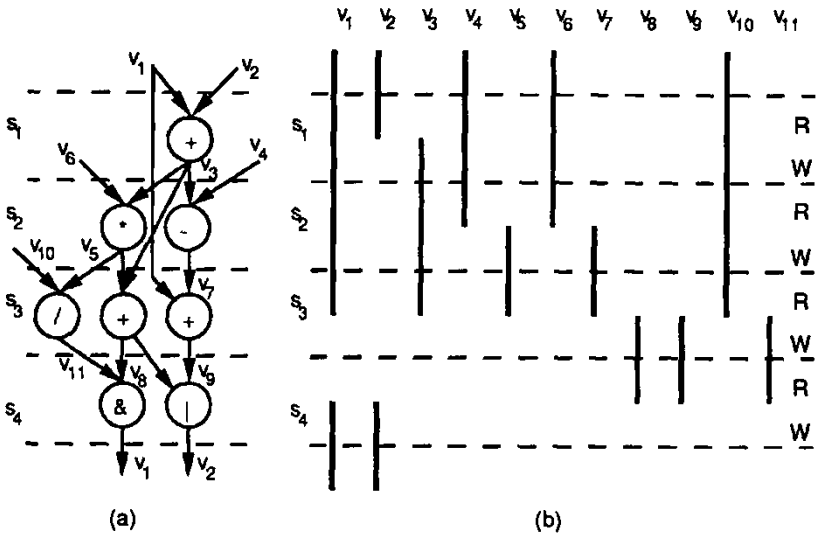


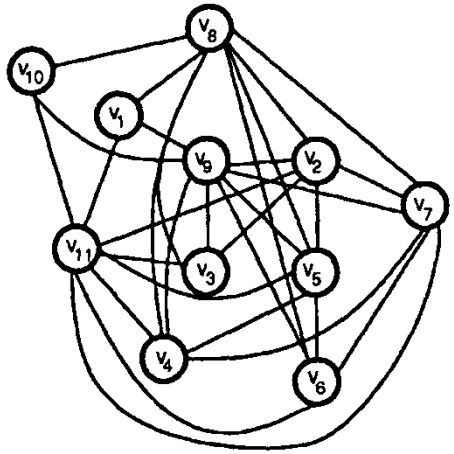
Dekompozice grafu na kliky

- Aplikace na úlohu alokace
  - Přiřazení funkčních jednotek
    - vrcholy grafu jsou operace vstupního DFG
    - hrany propojují takové vrcholy, pro které platí:
      - nejsou vykonávány ve stejném kontrolním kroku
      - mohou být zpracovány stejnou funkční jednotkou (stejný typ)
  - Přiřazení paměťových elementů
    - vrcholy jsou proměnné
    - hrany propojují ty proměnné, které mohou být uloženy ve stejném registru (určí se na základě doby života)
  - Přiřazení propojovací sítě
    - vrcholy jsou vodiče mezi entitami (registry a funkční jednotky)
    - hrany propojují pouze ty vrcholy, kde neprobíhá souběžně přenos dat

Dekompozice grafu na kliky

- Příklad:
  - alokace paměťových elementů na základě doby života
- Nevýhody:
  - Jednotlivé kroky se provádějí nezávisle i přes to, že mezi nimi existuje závislost - nemusíme dostat optimální řešení
- Vylepšení:
  - Alternativně existují metody, kde jednotlivé hrany grafu jsou navíc ohodnoceny podle toho, jak jsou výhodné pro další kroky v rámci alokace





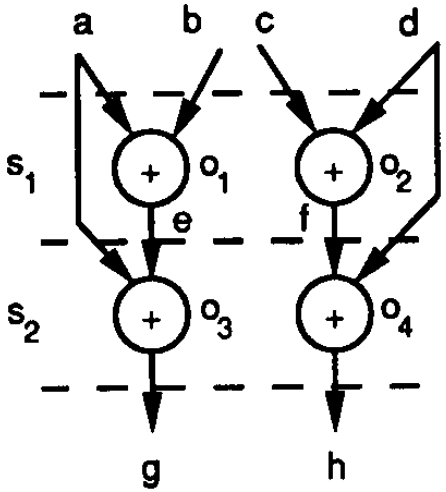
Dekompozice grafu na kliky

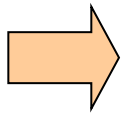
1. Přiřazení operací k funkčním jednotkám

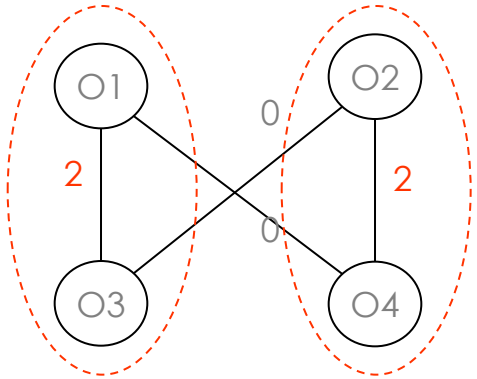
- Sestavení grafu, kde vrcholy reprezentují operace a hrany spojují takové operace, kterou jsou mapovány do oddělených kontrolních kroků
- Ohodnocení hran grafu
  - Počáteční ohodnocení hran nastaveno na nulu
  - K ohodnocení hrany je přičtena jednička, pokud dvojice operací sdílí některou se svých vstupních nebo výstupních proměnných
- Rozdělení grafu na kliky s ohledem na maximální ohodnocení hran

Dekompozice grafu na kliky

```
O1: e = a + b;
O2: f = c + d;
O3: g = a + e;
O4: h = f + d;
```







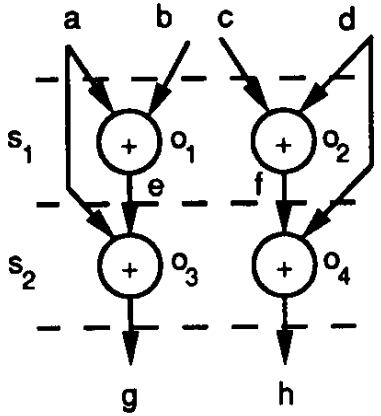
F1	F2
O1: e = a + b;	O2: f = c + d;
O3: g = a + e;	O4: h = f + d;

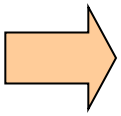
2. Přiřazení proměnných do registrů

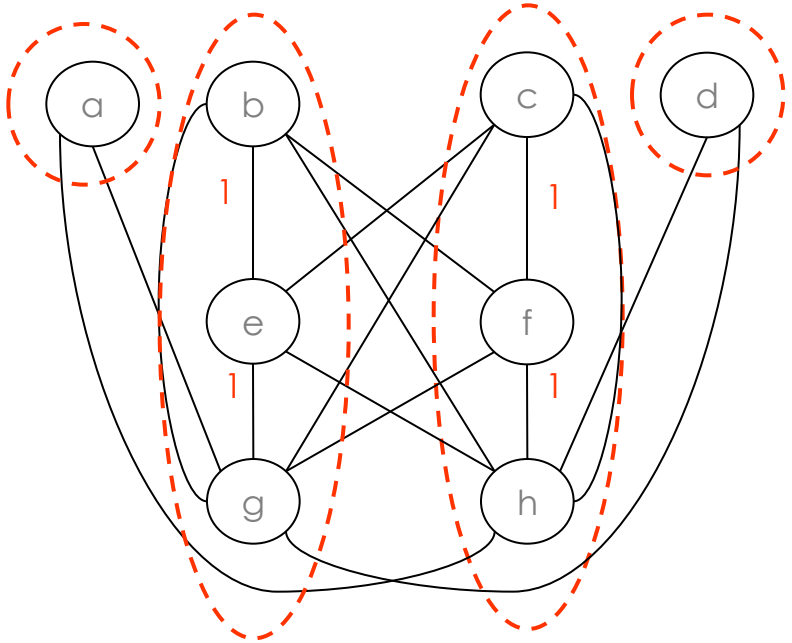
- Sestavení grafu, kde vrcholy reprezentují proměnné a hrany spojují takové proměnné, jejichž doba života se nepřekrývá
- Ohodnocení hran grafu
  - Počáteční ohodnocení hran nastaveno na nulu
  - K ohodnocení hrany je přičtena jednička, pokud dvojice proměnných sdílí při výpočtu některý ze vstupních nebo výstupních portů. Toto sdílení se posuzuje s ohledem na přiřazení operací k funkčním jednotkám z předchozího kroku
- Rozdělení grafu na kliky s ohledem na maximální ohodnocení hran

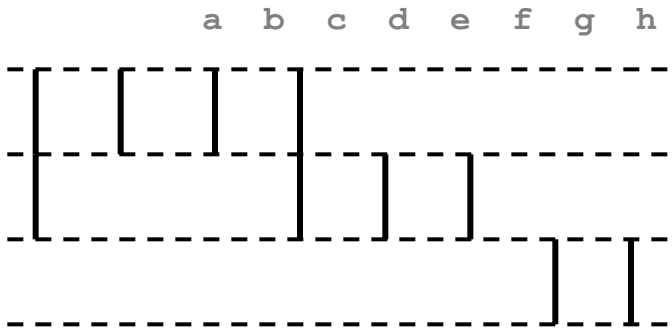
Dekompozice grafu na kliky

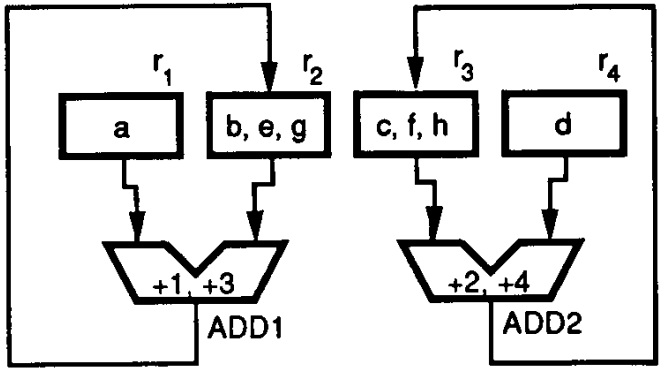
F1	F2
O1: e = a + b;	O2: f = c + d;
O3: g = a + e;	O4: h = f + d;









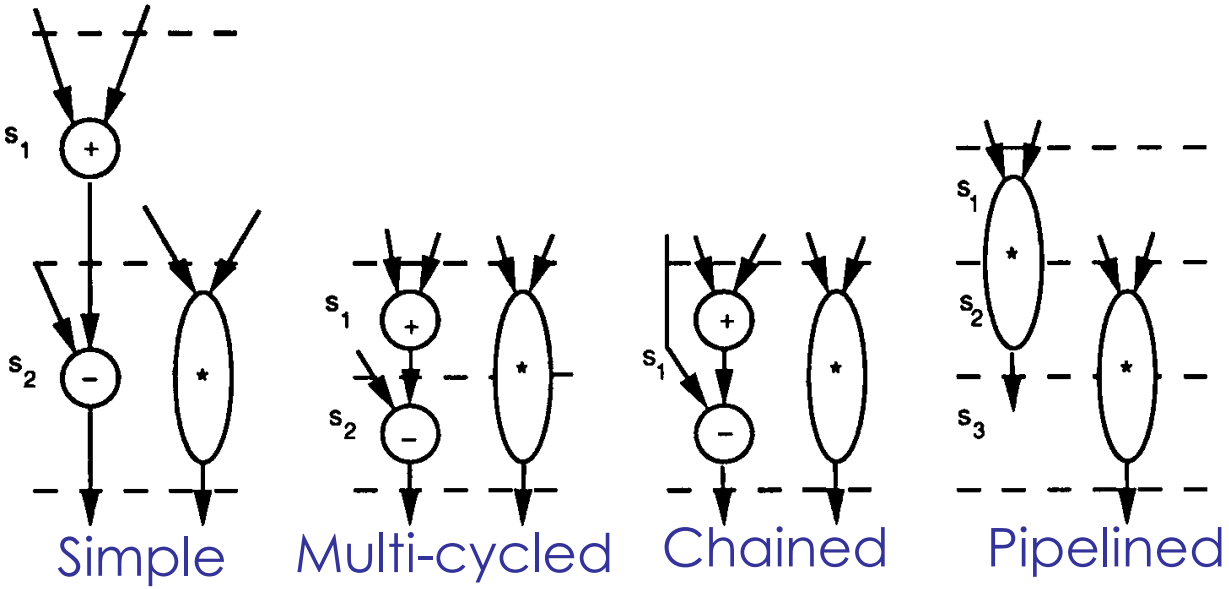


Osnova

- Úvod
- Reprezentace obvodu
- Základní transformace
- Plánování
- Přiřazení
- Alokace
- Generování RTL
- Shrnutí

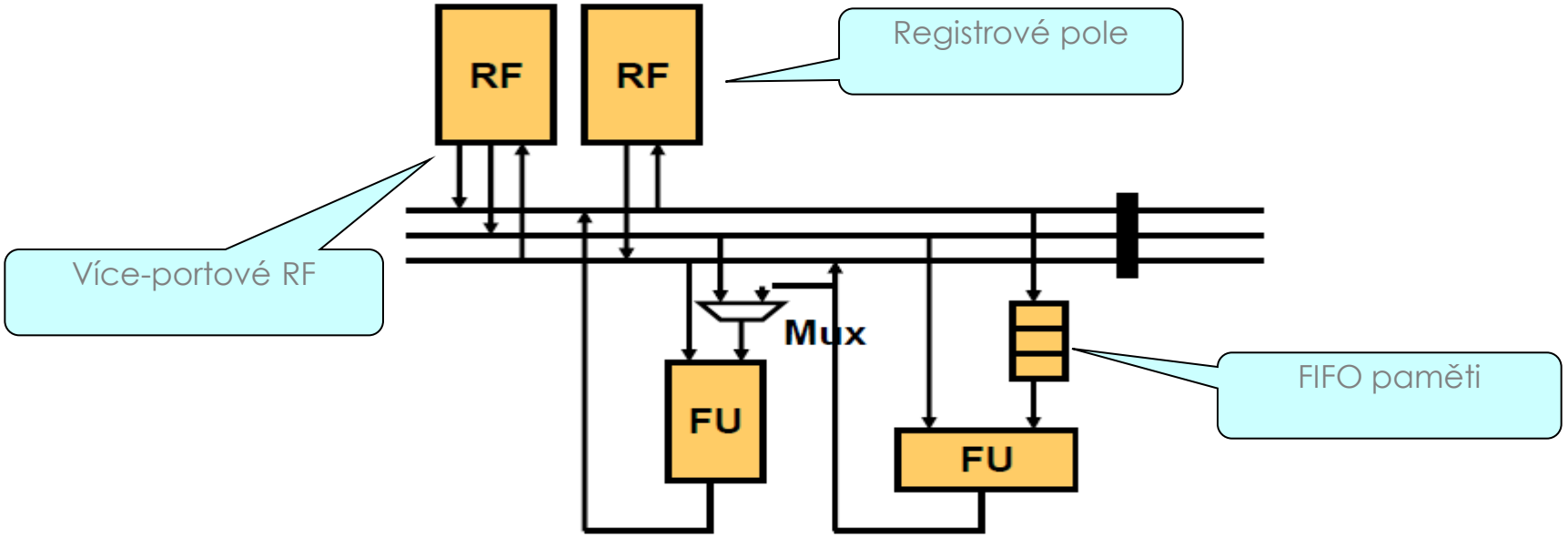
Alokace zdrojů

- Výběr funkčních jednotek
  - Zřetězené, více-cyklové, multifunkční, z různým poměrem latence vs. Množství zdrojů
  - Často se provádí jako součást plánování
  - Jednotlivé jednotky se vybírají z dostupné knihovny pro konkrétní technologii



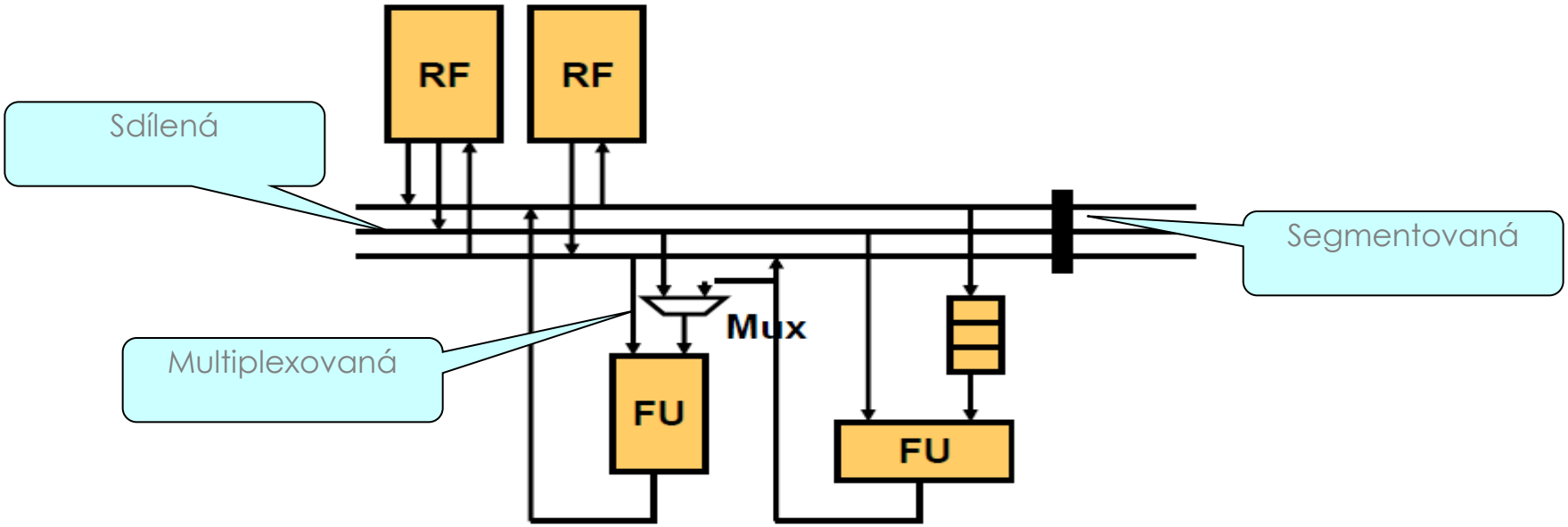
Alokace zdrojů

- Výběr paměťových elementů pro uložení proměnných
  - Registr, Registrové pole, Více-portové RF, RAM, ROM, FIFO
  - Často se provádí v průběhu přiřazení proměnných do paměťových elementů
  - Jednotlivé bloky se vybírají z dostupné knihovny pro konkrétní technologii



Alokace zdrojů

- Výběr propojení sloužících pro datové přenosy
  - Sdílená sběrnice, multiplexovaná sběrnice, segmentovaná sběrnice, specifický komunikační protokol
  - Často se provádí v průběhu přiřazení propojovací sítě
  - Výběr sběrnice je často ovlivněn konkrétní použitou technologií



Alokace, plánování, přiřazení

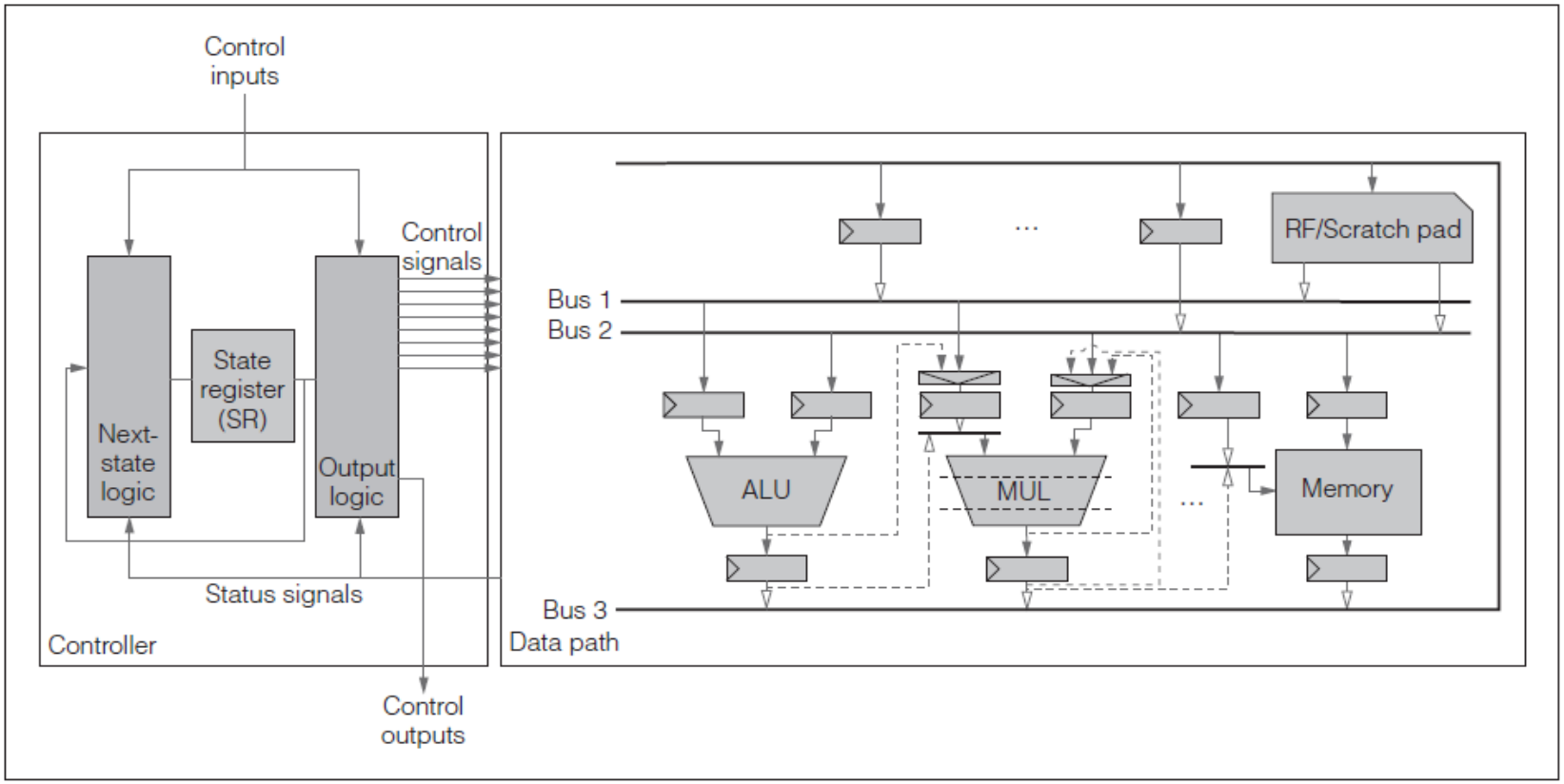
- Procesy alokace, plánování a přiřazení jsou na sobě závislé
  - Alokace/výběr funkčních jednotek (zřetězené, více-cyklové) výrazně ovlivňuje proces plánování
  - V průběhu časově omezeného plánování se mění počet funkčních jednotek s ohledem na splnění časového kritéria
  - V průběhu přiřazování se mohou alokovat dodatečné zdroje jako jsou multiplexory nebo paměťové elementy pro uložení proměnných – zbývá méně zdrojů pro funkční jednotky
- Nejlépe pokud by byly optimalizovány všechny tři fáze současně - s ohledem na složitost jednotlivých částí je to však obtížně realizovatelné
- Přesný postup závisí na požadovaných cílech obvodu:
  - Alokace bude prováděna jako první, pokud je cílem získat obvod s minimální latencí resp. maximální propustností s omezeným počtem zdrojů (prostorově omezené plánování) - FPGA
  - Alokace bude prováděna v průběhu plánování, pokud je cílem získat obvod s minimálním počtem zdrojů splňující zadaný čas výpočtu (časové omezené plánování) - telekomunikační nebo multimediální aplikace
- Čas a prostor nejsou jedinými kritérii - v poslední době se preferuje plánování s ohledem na hodinovou frekvenci, spotřebu, propustnost paměti apod.

Osnova

- Úvod
- Reprezentace obvodu
- Základní transformace
- Plánování
- Přiřazení
- Alokace
- Generování RTL
- Shrnutí

Generování RTL

- Na základě výsledků procesu alokace, plánování a přiřazení je generováno výsledné schéma obvodu

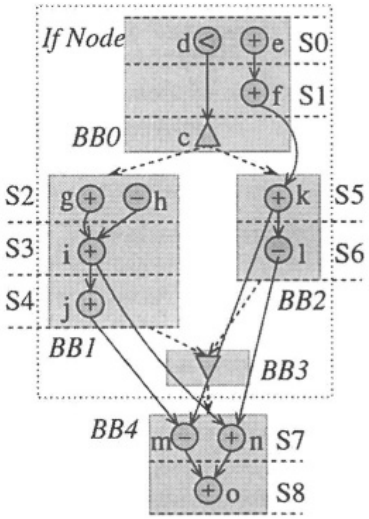


Generování RTL

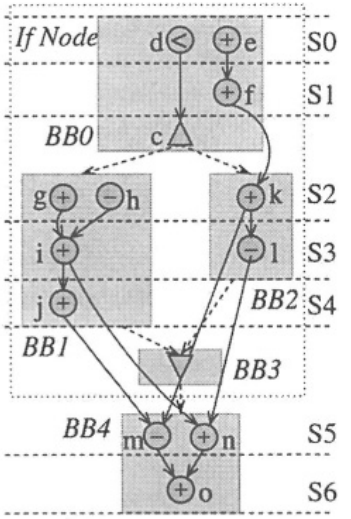
- Výsledný obvod je složen z:
  - Datové cesty: Obsahuje propojení funkčních jednotek, paměťových elementů a komunikačních sběrnic –
    je výsledkem procesu přiřazení
  - Řadiče: Reprezentována konečným automatem popř. programovatelným konečným automatem, který řídí běh výpočtu skrze kontrolní signály (aktivace funkčních jednotek, ovládání multiplexorů, aktivace signálů pro zápis do registrů a paměťových bloků, apod.)
  - Primární vstupy/výstupy jsou zapojeny jak do datové cesty (datové v/v), tak do řadiče (v/v kontrolní signály).

Generování FSM

- Mapování kontrolních kroků na stavy automatu:
  - Lokální rozdělení
    - u větvení grafu na vzájemně vyloučené části, se stavy generují ke každé části zvlášť
    - velký počet stavů
    - není potřeba stavový registr
  - Globální rozdělení
    - u větvení grafu na vzájemně vyloučené části, se stavy generují pro obě části společně
    - malý počet stavů
    - je potřeba stavový registr, aby rozlišil, jaké větev se vykonává
    - vede na menší zabranou plochu na čipu oproti lokálnímu rozdělení, syntezátor má větší prostor pro optimalizace

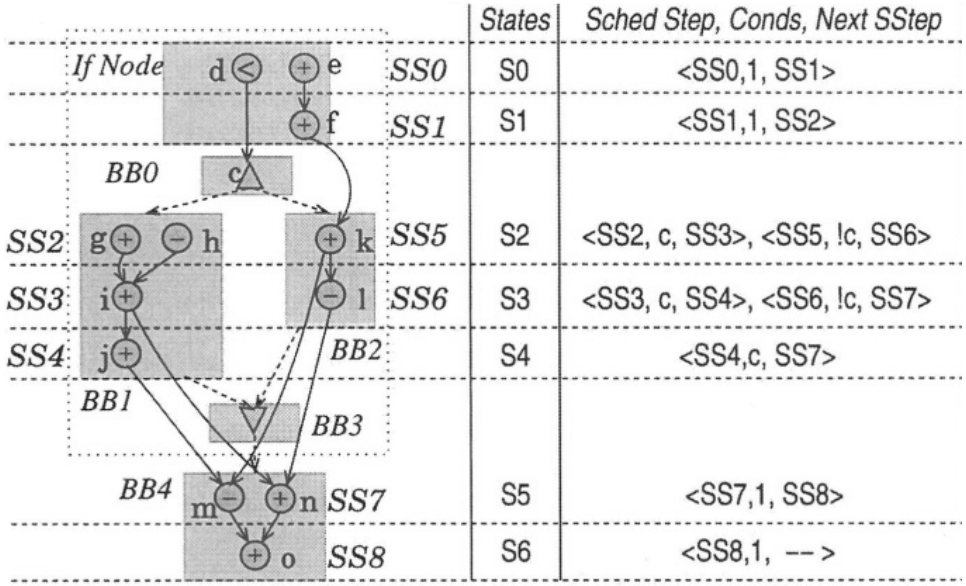


a) Lokální

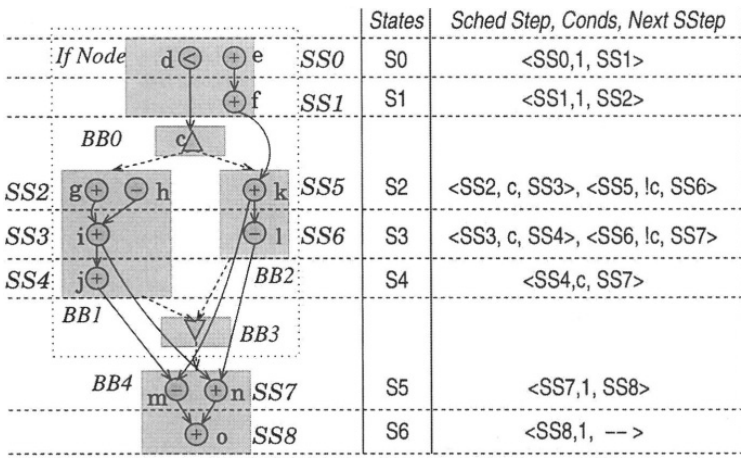


b) Globální

- Vytvoření tabulky stavů
  - Naplánovaný CDFG je podrobněji rozepsán do seznamu kroků ve tvaru *<Step, Conds, Next Step>*
  - Pro větvení resp. spojení rozvětvených bloků není potřeba extra krok – provádí se v navazujících blocích
  - Na základě vytvořené tabulky lze přímo generovat FSM (Lokální rozdělení)
  - Pro globální rozdělení je potřeba sledovat navíc větvení uvnitř jednotlivých stavů



- Generování VHDL kódu




```
Sync_logic: process(CLK, RESET)
Begin
   if (RESET = '1') then
      state <= S0;
   elsif(CLK'event and CLK = '1') then
      state <= next_state;
   end if;
End process;
```

```
Next_state_logic:
process(state, conds)
Begin
 case state is
 when S0 =>
   next_state <= S1
 when S1 =>
   next_state <= S2
 when S2 =>
   next_state <= S3
 when S3 =>
   if (cond) then
      next_state <= S4
   else
      next_state <= S5
   end if;
   ...
 end case;
End process;
```

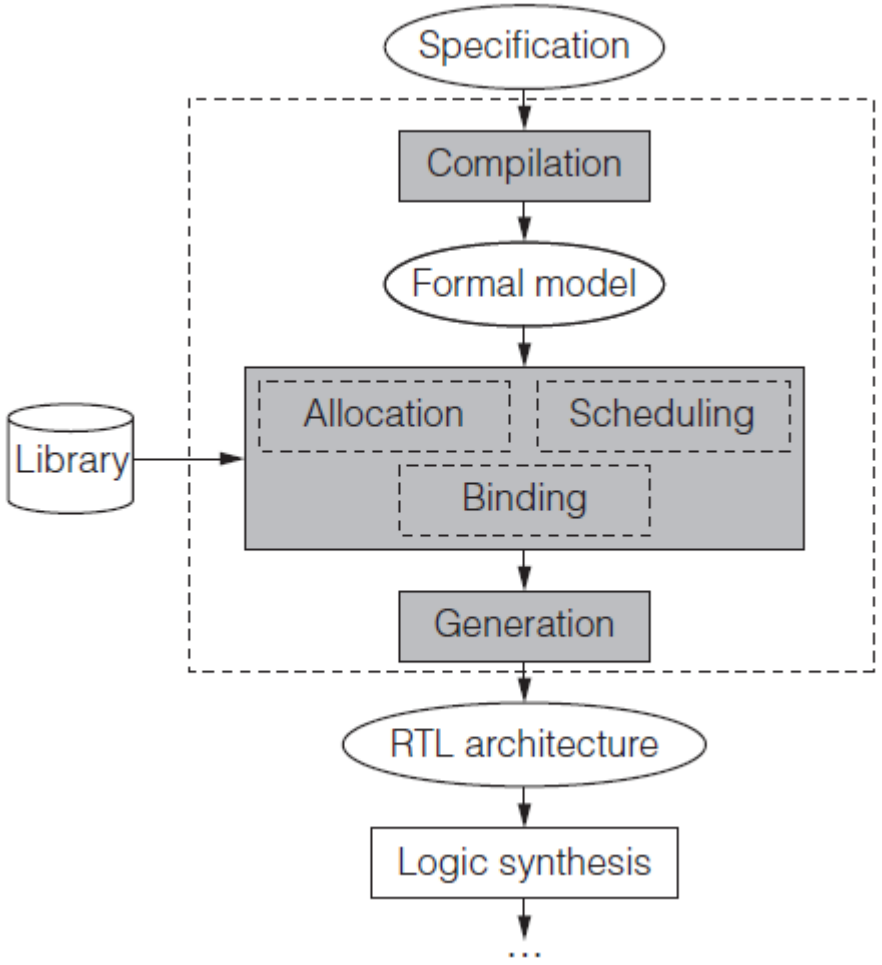
```
Output_logic:
process(state, conds)
Begin
 case state is
 when S0 =>
   set outputs for SSO
 when S1 =>
   set outputs for SS1
 when S2 =>
   if (cond) then
      set outputs for SS2
   else
      set outputs for SS5
   end if;
 when S3 =>
   if (cond) then
      set outputs for SS3
   else
      set outputs for SS6
   end if;
   ...
 end case;
End process;
```

Osnova

- Úvod
- Reprezentace obvodu
- Základní transformace
- Plánování
- Přiřazení
- Alokace
- Generování RTL
- Shrnutí

Syntéza obvodů na vyšší úrovni

- Vstup:
  - Popis algoritmu ve vyšším programovacím jazyce (C/C++, popř. varianty)
- Výstup:
  - Maska obvodu ASIC nebo konfigurace obvodu FPGA
- Základní schéma návrhu:
  1. Kompilace vstupního kódu a převod do vhodné formální reprezentace (Control-Data Flow Graph)
  2. Plánování, alokace a přiřazení prostředků
  3. Generování RTL schéma
  4. Logická syntéza pomocí konvenčních nástrojů



Shrnutí

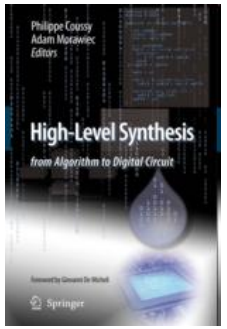
- Syntéza obecných algoritmů na úroveň digitálních obvodů často vede na velké množství realizací
- Výběr vhodné realizace je úlohou algoritmů v oblasti vysokoúrovňové syntézy (High-Level Synthesis):
  - Obvod se nejprve převede na vhodnou reprezentaci např. Control Data Flow Graph (CDFG)
  - Následuje fáze plánovaní, která vstupní CDFG mapuje do kontrolních kroků s ohledem na požadovaná kritéria – čas, prostor na čipu apod.
  - Na závěr nastává fáze alokace/přiřazení zdrojů, jejíž cílem je alokovat a mapovat proměnné a operace CDFG grafu do funkčních jednotek, registrů a propojovacích sítí s ohledem na cenu výsledné realizace

Shrnutí

- Dílčí úlohy syntézy reprezentují často NP-těžké problémy, které je potřeba řešit pomocí heuristik
- Pro syntézu reálných obvodů je navíc potřeba algoritmy doplnit o reálné vlastnosti funkčních jednotek (multi-cycled, chained, pipelined)
- Kvalitní výsledky mohou být dosaženy propojením jednotlivých fází syntézy (plánování, alokace, přiřazení)
- V současné době jsou již k dispozici komerční nástroje podporující syntézu na vysoké úrovni (SystemC, Catapult Synthesis, Spark, apod.)
- Prozatím při tvorbě obvodu vyžadují častou interakci s uživatelem. Do budoucna lze však očekávat, že budou schopny prohledávat stavový prostor obvodů efektivněji a interakce s uživatelem bude ubývat.

Literatura

- Gajsky D., Dutt N., Wu A., Lin S.:***High-Level Synthesis: Introduction to Chip and System Design***, ISBN 079239194-2, 1992
- ***The Design Warrior's Guide to FPGAs***, ISBN 0750676043, 2004 Mentor Graphics Corp.
- Micheli G., ***High-Level Synthesis from Algorithm to Digital Circuit***, A.Morawiec (Eds.), 2008
- ***Special issue on High Level Synthesis***, IEEE Design & Test of Computers, Volume 26, Issue 4, 2009
- Gupta S., Gupta R., Dutt N., ***SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits***, Springer, 2004



Děkuji za pozornost