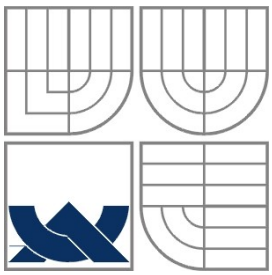
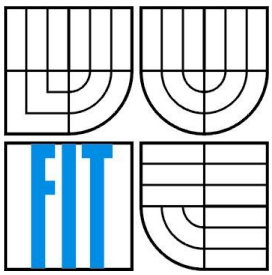


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GENERÁTOR HARDWARE A OPTIMALIZÁTOR V PROJEKTU LISSOM

PRÁCE DO PŘEDMĚTU TEORIE PROGRAMOVACÍCH JAZYKŮ

AUTOR PRÁCE Ing. Martin Mecera

BRNO 2011

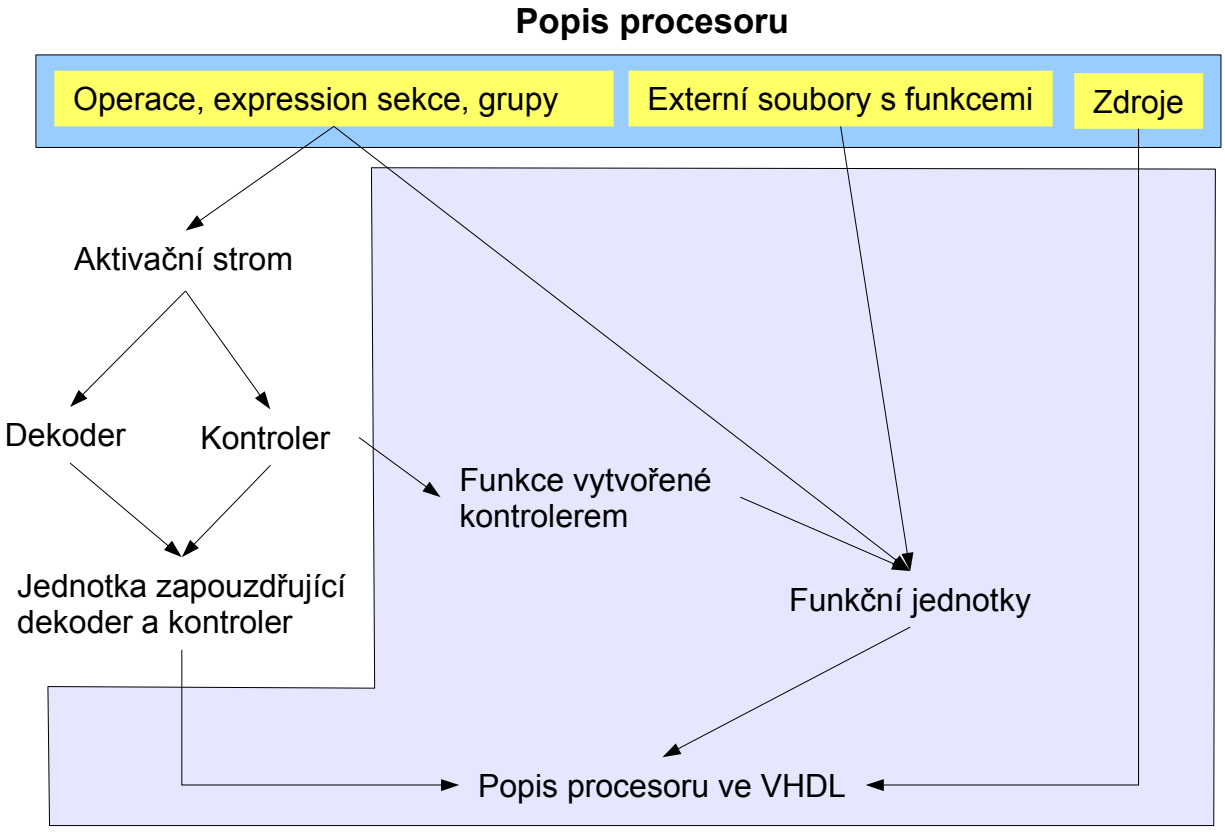
Obsah

1 Úvod

Tento dokument popisuje vnitřní a zadní část překladače jazyka ISAC. Z dokumentace je vynecháno generování dekoderu instrukcí a kontroleru procesoru.

Jazyk ISAC slouží k popisu procesoru. Dokumentace k jazyku ISAC je dostupná v [1]. Dále v textu se předpokládá, že čtenář je seznámen s metodikou návrhu procesoru v jazyku ISAC.

Na obrázku 1 je znázorněno generování jednotlivých částí výsledného popisu HW. Šipky označují tok dat od popisu částí procesoru (žluté rámečky) přes jednotlivé transformace po cílový popis procesoru v jazyku VHDL. Tento dokument se zabývá popisem transformací v modro-šedé oblasti.



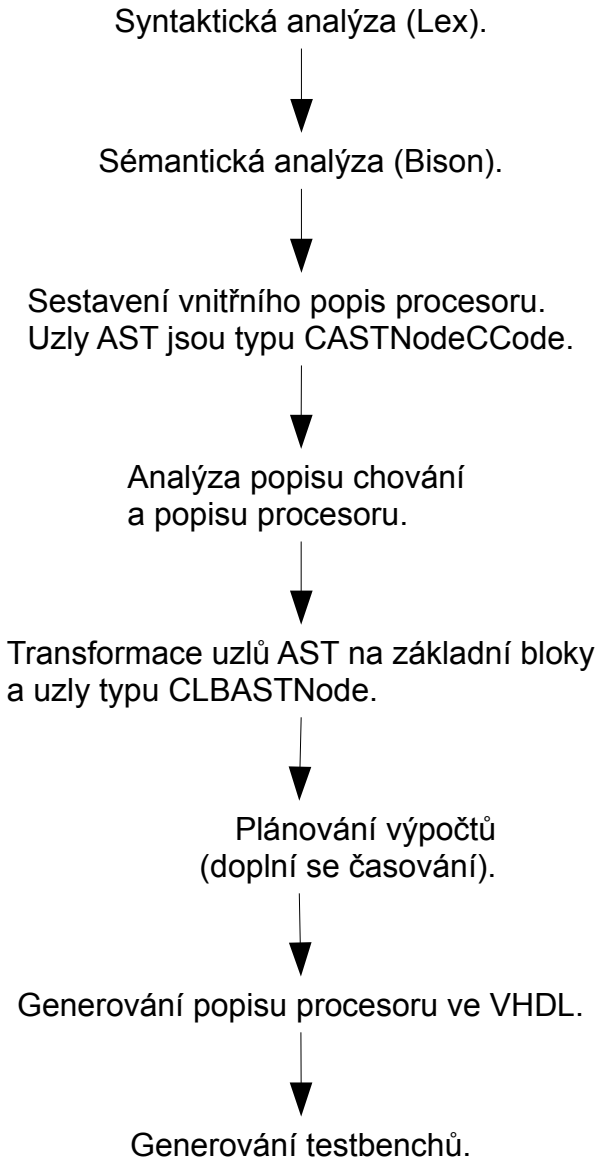
Obrázek 1: generování jednotlivých částí výsledného popisu HW

1.1 Slovník pojmů

Název	Popis
Operace	Operace v jazyku ISAC. Mezi její významné sekce patří BEHAVIOR (popis chování v jazyku C) a EXPRESSION (viz dále). Operace nemůže obsahovat jak sekci BEHAVIOR tak EXPRESSION. Sekce BEHAVIOR nevrací hodnotu.
Expression sekce	Obsahuje výraz, který vrací hodnotu (podobně jako funkce). Tato sekce je součástí operace.
Grupa	Seskupuje operace.
Externí soubor (*.c, *.h)	Soubor se zdrojovým kódem funkcí (resp. hlavičkový soubor). Bývá přilinkován k popisu procesoru (*.isac). Obsahuje funkce, které jsou volány z operací. Operace provádí aktivaci funkční jednotky odpovídající volané funkci.
Zdroj	Např. registr nebo paměť. Zdroje jsou definované v RESOURCE sekci.
Aktivační strom	Vnitřní reprezentace pro účel konstrukce kontroleru a dekoderu. Získá se z operací a grup.
Dekoder	Instrukční dekoder.
Kontroler	Řadič výpočtu. Aktivuje výpočty v operacích a funkcích vytvořených kontrolerem.
Funkce vytvořená kontrolerem	Funkce je automaticky vytvořena z některých jazykových konstrukcí v sekci ACTIVATION. Typicky se jedná o vyhodnocení podmínky nebo o čtení ze zdroje.
Funkční jednotka	Souhrnný pojem pro zapouzdření výpočtu v HW. Funkční jednotka se získá z behaviorálního popisu. V HW typicky obsahuje stavový automat a datovou cestu (angl. datapath).
Jednotka zapouzdřující kontroler a dekoder	Jednotka v HW. Bývá propojena s funkčními jednotkami, které aktivuje (zahajuje jejich výpočet).
Popis procesoru ve VHDL	Finální podoba procesoru po provedení všech transformací.

2 Posloupnost transformací

Na obrázku 2 je vyznačna posloupnost transformací z popisu procesoru v jazyku ISAC do popisu procesoru v jazyku VHDL. Na proces generování popisu ve VHDL navazuje automatizovaný proces generování testbenchů ve VHDL. V následujících kapitolách bude pojednáno o procesech „Sestavení vnitřního popisu procesoru“, „Analýza popisu chování a popisu procesoru“, „Transformace uzlů AST na základní bloky a uzly typu CLBASTNode“, „Plánování vykonání výpočtů“ a „Generování popisu procesoru ve VHDL“.



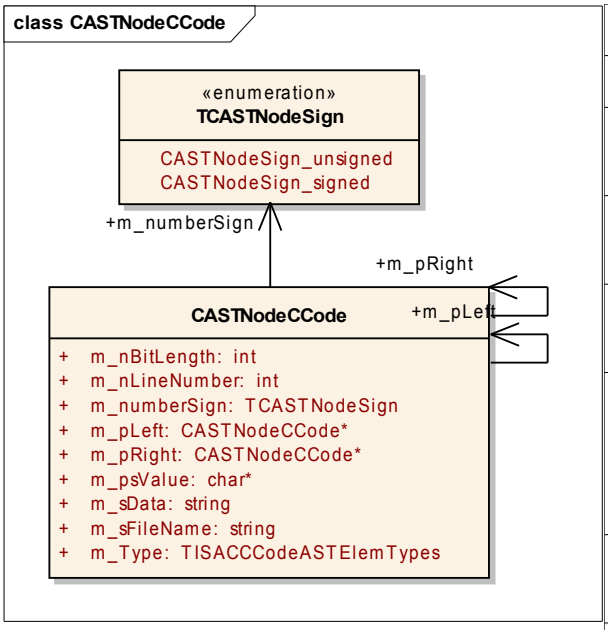
Obrázek 2: Posloupnost transformací

3 Sestavení vnitřního popisu procesoru

Proces „Sestavení vnitřního popisu procesoru“ zde nebude popisován, pouze jeho výsledek.

Celková struktura procesoru je uložena v jedinečném objektu (angl. singleton) typu CIsac (definován v knihovně parserlib2).

Behaviorální popis (chování) operací, expression sekcí a funkcí je popsán pomocí abstraktního syntaktického stromu (AST). Uzly AST jsou typu CASTNodeCCode. Uzel typu CASTNodeCCode je definován v knihovně parserlib2. Později v tomto dokumentu bude popsán uzel typu CLBASTNode, který obsahuje více informací než uzel typu CASTNodeCCode. Uzel typu CLBASTNode se dostane z uzlu typu CASTNodeCCode příslušnou transformací spojenou s důkladnější analýzou celkového popisu procesoru. Na obrázku 3 je zobrazen diagram tříd pro uzel typu CASTNodeCCode. Tabulka vedle obrázku popisuje význam jednotlivých atributů. Atributy m_nBitLength a m_numberSign nejsou nastaveny – k tomu je potřeba analyzovat syntaktický strom zdola nahoru. Tato analýza bude provedena v procesu „Analýza popisu chování a popisu procesoru“.



Obrázek 3: Třída uzlu AST typu CASTNodeCCode

Název atributu	Význam
m_Type	Typ uzlu
m_psValue	Název uzlu (např. „if“)
m_pLeft	Odkaz na levého potomka uzlu
m_pRight	Odkaz na pravého potomka uzlu
m_sData	Dodatečná textová informace (např. číselná hodnota konstanty)
m_sFileName	Název zdrojového souboru
m_nLineNumber	Řádek ve zdrojovém souboru
m_nBitLength	Bitová šířka hodnoty uzlu
m_numberSign	Znaménko hodnoty

Výčtový typ TCASTNodeSign označuje znaménko hodnoty v uzlu.

4 Analýza popisu chování a popisu procesoru

Tento proces je definován v knihovně astanalyzel. Knihovna obsahuje jedinou třídu ASTAnalyze. V systému se nachází jen jediná instance této třídy (singleton), podobně jako existuje jediná instance třídy CIsac pro vnitřní model procesoru. Mezi instancemi tříd ASTAnalyze a CIsac existuje tento vztah: v instanci ASTAnalyze jsou uloženy doplňkové informace o vnitřním modelu procesoru. Například obsahuje graf volání funkcí z operací. Doplňkové informace jsou sesbírány hlubší analýzou modelu procesoru než jaká je prováděna sémantickým analyzátorem během překladu.

Třída ASTAnalyze poskytuje tyto funkcionality:

1. Analýza volání funkcí.
2. Výpočet bitových šířek a znamének hodnot v uzlech typu CASTNodeCCode.

3. Výpočet grafu volání funkcí a aktivace operací. Výpočet pořadí zpracování funkcí a operací.

Funcionality jsou volány v uvedeném pořadí.
Následující podkapitoly popisují jednotlivé funkcionality.

4.1 Analýza volání funkcí

Proces analýzy volání funkcí probíhá takto:

Akce	**Popis**
Načtení deklarací funkcí z hlavičkových a zdrojových souborů. (funkce ASTAnalyze::LoadFunctionDeclarations)	Zapamatuje si název, parametry a návratovou hodnotu každé funkce.
Pro každou funkci proveď 1. Vytvoř prázdný seznam lokálních proměnných. Do seznamu dej parametry funkce a speciální proměnnou pro návratovou hodnotu. 2. Zavolej funkci pro analýzu AST (ASTAnalyze::AnalyzeExpression). Funkci předej seznam lokálních proměnných. 3. Vyzvedni si upravený seznam lokálních proměnných. Ulož si jej pro další zpracování.	

4.2 Výpočet bitových šířek a znamének

Tato funkcionalita má za cíl nastavit atributy m_nBitLength a m_numberSign uzlů typu CASTNodeCCode. Výpočet probíhá ve funkci pro analýzu AST (ASTAnalyze::AnalyzeExpression). Vypočítané bitové šírky a znaménka musejí respektovat normu ANSI C. Bitové šířky musejí udržet hodnotu výsledku beze ztráty přesnosti. Například sčítáním dvou 8bitový čísel bez znaménka obdržíme 9bitové číslo bez znaménka.

Pořadí zpracování operací, grup, expression sekcí a funkcí nemůže být náhodné. Algoritmus postupuje v tzv. topologickém uspořádání. (Poznámka: rekurze není povolena, protože jazyk

ISAC nemá implicitní podporu pro zásobník). Příklad: jestliže fukce A volá funkci B, pak nejprve je vypočítána bitová šířka návratové hodnoty funkce B. Následně je tato bitová šířka použita při analýze AST ve funkci A.

Funkce pro analýzu AST (ASTAnalyze::AnalyzeExpression) je rekurzivní. Prochází AST metodou zdola nahoru. Kromě výpočtu bitových šířek a znamének uzlů provádí kontrolu definice identifikátorů. Funkce umí ignorovat neznámé funkce, které jsou určeny k simulačním účelů (např. printf).

4.3 Výpočet grafu volání funkcí a aktivace operací

Pro účely procesu „Plánování vykonání výpočtů“ (viz obr. 2) je sestaveno pořadí vyhodnocení operací a funkcí. Algoritmus provede topologické setřídění a uloží pořadí pro pozdější použití.

5 Transformace uzlů AST na základní bloky a uzly typu CLBASTNode

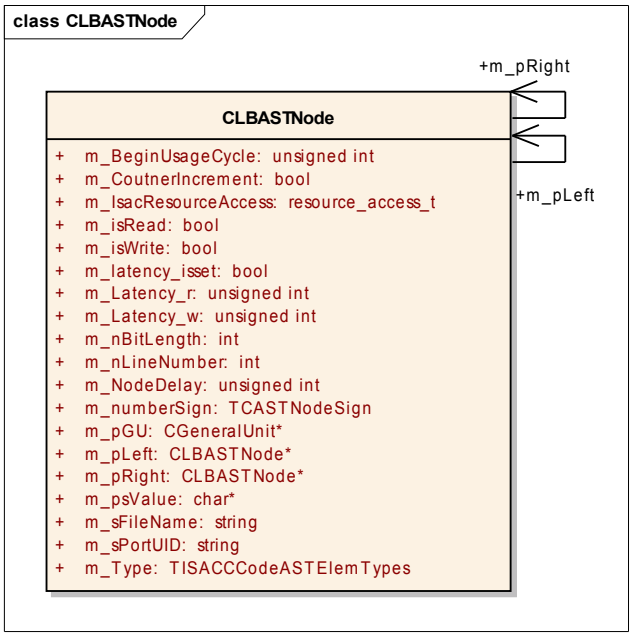
Transformace probíhá v knihovně isac2lbastl. Algoritmus transformuje uzly AST typu CASTNodeCCode na uzly typu CLBASTNode. V dalším kroku rozdělí uzly AST do základních bloku (angl. basic block, viz teorie k překladačům).

Na obrázku 4 je zobrazen diagram třídy pro uzel typu CLBASTNode. Tabulka vedle obrázku popisuje význam jednotlivých atributů. Atribut m_sPortUID bude nastaven až v procesu „Plánování vykonání výpočtu“.

Transformace uzlů typu CASTNodeCCode na uzly typu CLBASTNode obsahuje tyto fáze (pouze významné):

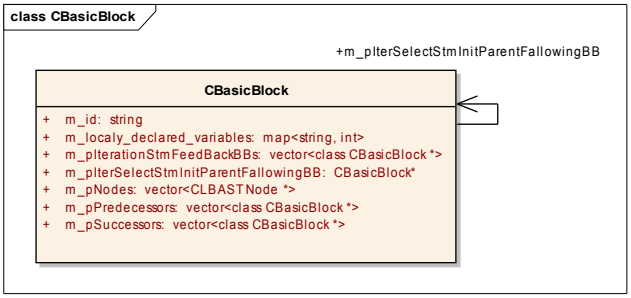
1. Transformace cyklu for na cyklus while.

2. Unifikace příkazů se shodnou sémantikou. Například čtení z paměti lze zapsat jako volání funkce read na objektu paměti nebo jako přístup do pole (mem.read(adrs) vs. mem[adrs]). Volání funkce je převedeno na přístup do pole.
3. Transformace matematických operátorů. Např. „i++“ na „i=i+1“.
4. Transformace výrazu „return hodnota“ na zápis hodnoty do zvláštní proměnné.
5. Transformace aliasů na skutečné zdroje.
6. Propagace konstant.



Obrázek 4: Uzel AST typu CLBASTNode

Název atributu	**Význam**
m_latency_isset	*Nemá význam pro generátor HW.*
m_Latency_r	
m_Latency_w	
m_Type	Typ uzlu
m_psValue	Textová hodnota (např. konstanta)
m_pLeft	Odkaz na levý uzel
m_pRight	Odkaz na pravý uzel
m_pGU	Odkaz na objekt zdroje
m_isRead	Jedná se o čtení?
m_isWrite	Jedná se o zápis?
m_NodeDelay	*Nemá význam pro generátor HW.*
m_BeginUsageCycle	
m_CoutnerIncrement	
m_IsacResourceAccess	Typ přístupu ke zdroji
m_sFileName	Název souboru
m_nLineNumber	Číslo řádku
m_nBitLength	Bitová šířka hodnoty uzlu
m_numberSign	Znaménko hodnoty
m_sPortUID	Identifikátor požadavku. *(Významné pro generátor HW)*



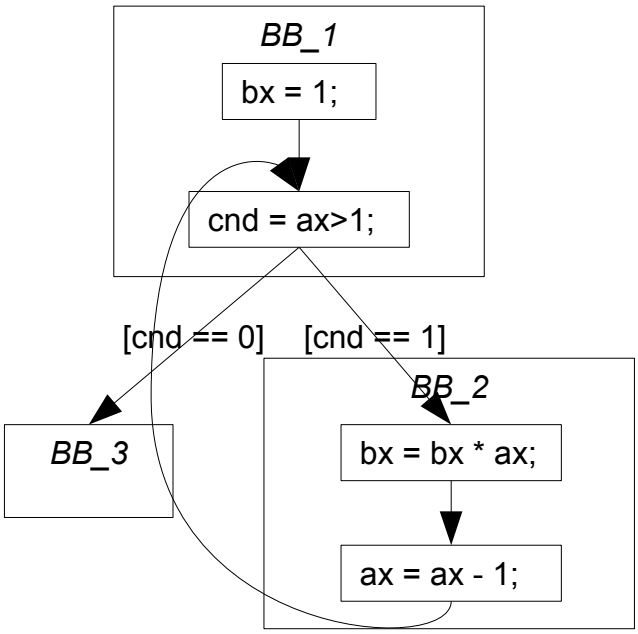
Obrázek 5: Základní blok

Název atributu	**Význam**
m_id	Identifikátor
m_pPredecessors	Přímí předchůdci bloku
m_pSuccessors	Přímí následovníci bloku
m_pNodes	Uzly AST v bloku
m_pIterationStmFeedBackBBs	Zpětné vazby z bloků na konci iterace.
m_pIterSelectStmInitParentFallowingBB	*Nemá význam pro generátor HW.*
m_localy_declared_variables	Proměnné deklarované v bloku. *V gen. HW se nepoužívá.*

Na obrázku 5 je zobrazen diagram třídy pro základní blok. Tabulka vedle obrázku popisuje význam jednotlivých atributů.

Příklad: Následující operace počítá faktoriál čísla v registru ax a ukládá jej do registru bx. Na obrázku 6 je znázorněno schéma základních bloků. Prázdný blok BB_3 je vytvářen implicitně – představuje tzv. koncový blok. Proměnná cnd nese vyhodnocení podmínky cyklu.

```
OPERATION factorial {
     ASSEMBLER { "fact"};
     CODING { 0b100001 };
     BEHAVIOR {
          bx=1;
          while(ax>1)
          {
            bx=bx*ax;
            ax=ax-1;
          }
     };
}
```



Obrázek 6: Schéma základních bloků pro operaci factorial

6 Plánování výpočtu

Proces plánování výpočtu zajišťuje knihovna lbast2rtlal. Algoritmus přijímá schéma výpočtu ve formě základních bloků. Vrací schéma výpočtu ve formě časovaného automatu (tzv. RTL automat).

Knihovna transformuje procedurální zápis výpočtu na control/data-flow graf (CDFG). Nad data flow grafem jsou prováděny optimalizace a plánování. Algoritmus plánuje výpočty metodou as soon as possible (ASAP).

Vnitřní reprezentace se skládá z objektů typu

- CDFGOperation – operace (funkční jednotka),
- CDFGBlock – základní blok,
- CDFGItem – jednotka plánování výpočtu.

Jednotka plánování výpočtu je vytvořena z několika uzlů AST. Dále v textu se pro ně bude užívat pojem „uzly výpočtu“.

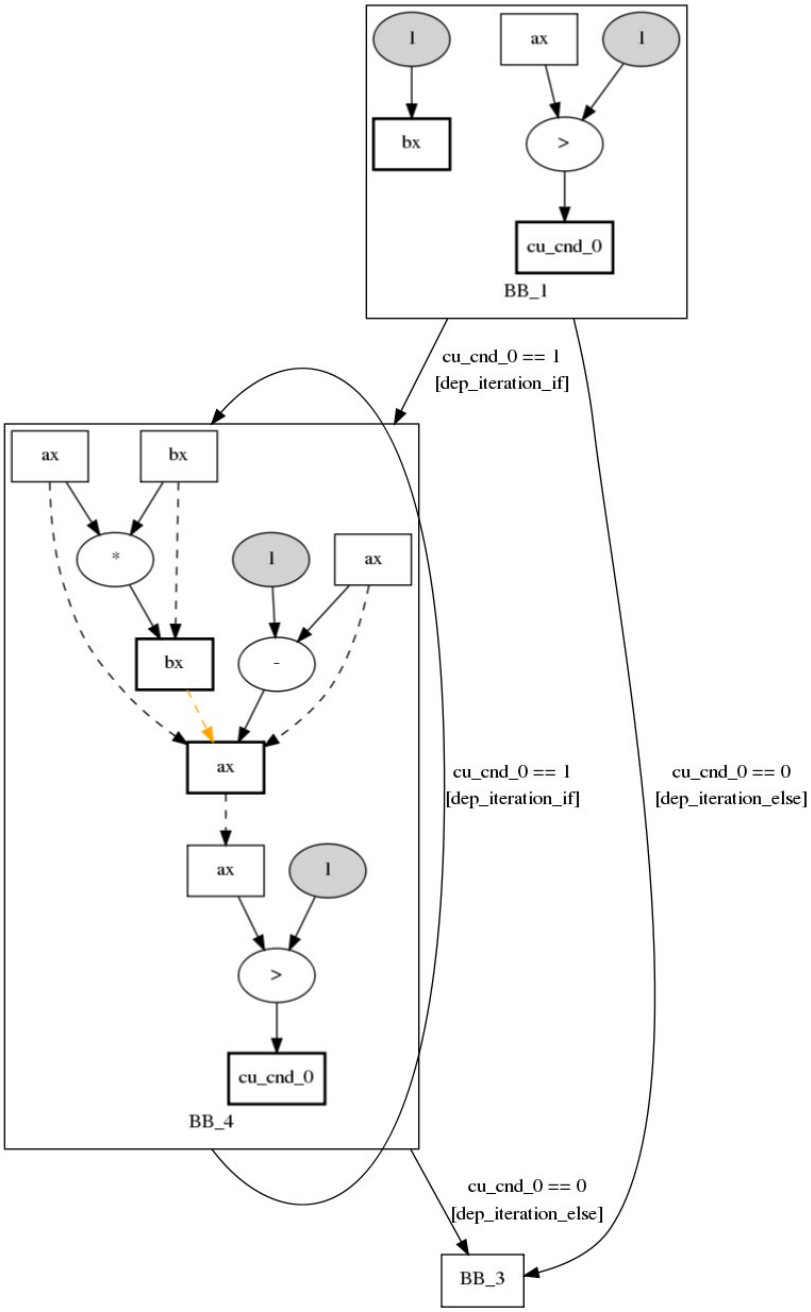
Vztah mezi objekty lze popsat množinově:

$$CDFGItem \in CDFGBlock \in CDFGOperation$$

Na obrázku 7 jsou znázorněny bloky typu CDFGBlock v operaci factorial z předchozí kapitoly. Mezi bloky jsou vytvořeny nové vazby, které chyběly u bloků z knihovny isac2lbastl. Uzly výpočtu jsou uspořádány do rozšířeného data-flow grafu. Vazby mezi uzly jsou znázorněny pomocí orientovaných čar.

Vlastnosti data-flow grafu (mezi uzly výpočtu uvnitř základního bloku):

1. Hranatý uzel s tenkou čárou představuje čtení ze zdroje.
2. Hranatý uzel s tlustou čárou představuje zápis do zdroje.
3. Oválný šedý uzel označuje konstantu.
4. Oválný bílý uzel označuje výpočet.
5. Plná čára označuje tok dat.
6. Přerušovaná čára označuje nutnost dodržet pořadí. Například je nutné dodržet pořadí čtení a zápisu do registru ax v bloku BB_4.
7. Přerušovaná oranžová čára je zvláštní typ přerušované čáry. Vynucuje pořadí výpočtu mezi podgrafy data-flow grafu. Na obrázku 7 je tato vazba mezi zápisem do registru bx a zápisem do registru ax. Důvod je následující: čtení z ax ve výrazu bx=bx*ax implikuje, že hodnota ax se nesmí změnit, dokud nedojde k dokončení zápisu do bx. Kdyby zápis do bx následoval až po zápisu do ax ve výrazu ax=ax+1, došlo by k zápisu chybné hodnoty bx.

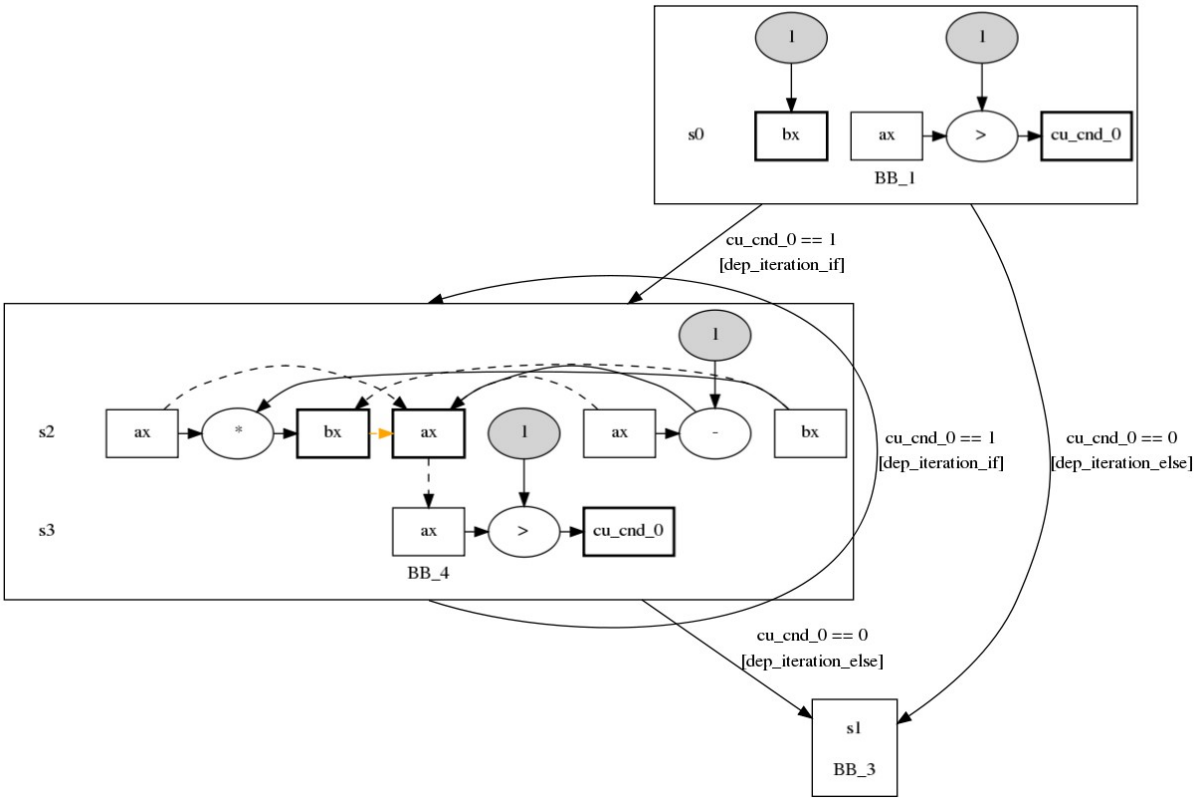


Obrázek 7: Operace factorial ve formě CDFG

Za povšimnutí stojí, že výpočet podmínky opakování cyklu (cu_cnd_0=ax > 1) byl zkopírován z bloku BB_1 na konec bloku BB_4.

Základní bloky tvoří uzly control-flow grafu. Vazby mezi bloky jsou podmíněné nebo nepodmíněné. V obrázku 7 jsou všechny vazby podmíněné.

Na obrázku 8 je vidět výstup algoritmu plánování metodou ASAP. Uzlům výpočtu byly přiřazeny stavy (takty), ve kterých jsou výpočty vykonávány. Stavy jsou v obrázku na levé straně (sN, N je přirozené číslo).



Obrázek 8: Plánování operace factorial metodou ASAP

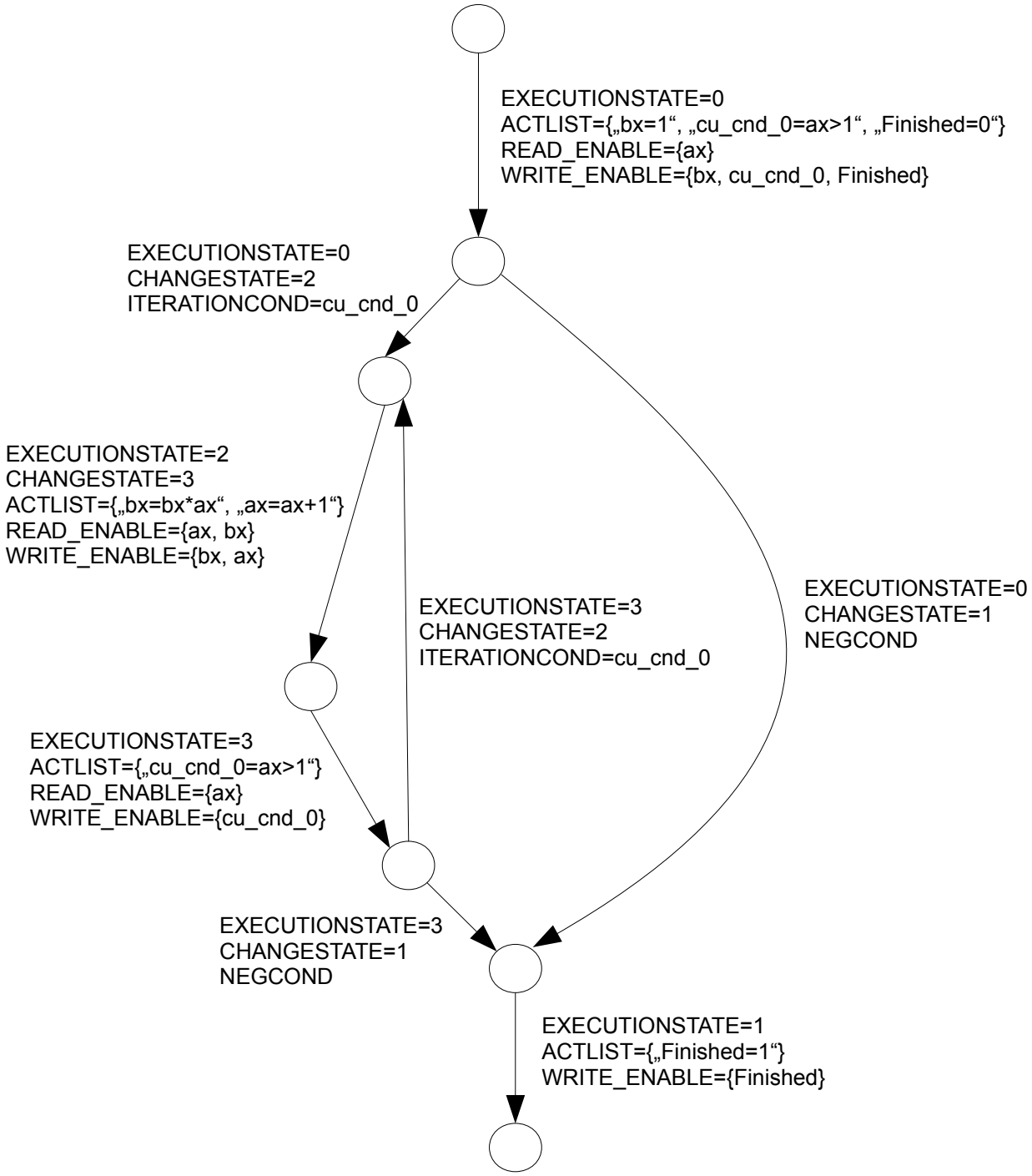
V koncovém bloku (BB_3) je vytvořen zvláštní stav (s1), který je vždy prázdný. Až generátor HW vytvoří odpovídající funkční jednotku, bude tento stav implicitní. Funkční jednotka v něm bude čekat (cyklit), dokud nepřijde aktivační signál. Aktivační signál uvede jednotku do počátečního stavu výpočtu (na obrázku 8 to je stav s0). Po skončení výpočtu se stav nastaví opět na implicitní stav (s1).

Poté, co proběhne plánování, se vytvoří časovaný automat (RTL automat). RTL automat jde na vstup generátoru HW nebo generátoru cycle-accurate simulátoru.

Na obrázku 9 se nachází RTL automat pro operaci factorial. Stavy RTL automatu (kolečka) neodpovídají stavům výpočtu (ASAP). Proto v obrázku nejsou uvedeny jejich názvy. Chování automatu je popsáno na jeho hranách. Hrana může obsahovat atributy.

Pro účely generátoru HW je do automatu dodána speciální proměnná Finished. Proměnná Finished má hodnotu 1, pokud se funkční jednotka nachází v koncovém (implicitním) stavu,

jinak 0. Proměnná Finished bude v HW transformována na 1bitový signál, jímž bude funkční jednotka informovat okolí, že dokončila výpočet.



Obrázek 9: RTL automat pro operaci factorial

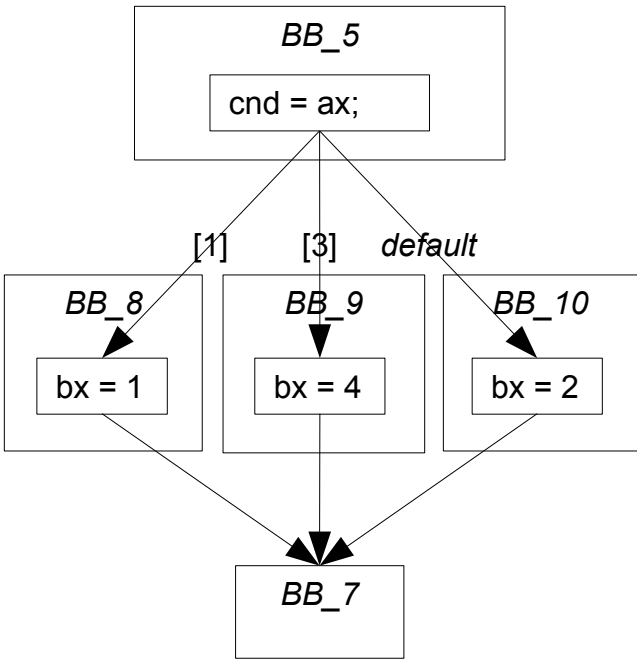
Následující tabulka uvádí význam atributů.

Atribut	**Význam**
EXECUTIONSTATE	Číslo stavu (podle ASAP). Pokud se funkční jednotka nachází v daném stavu, pak probíhá výpočet uvedený v atributu ACTLIST. Více hran může mít stejnou hodnotu atributu EXECUTIONSTATE. To znamená, že jejich výpočty budou provedeny paralelně.
CHANGESTATE	Číslo následujícího stavu (podle ASAP). Po skončení výpočtu dojde k přechodu do následujícího stavu. Pokud má více hran stejnou hodnotu atributu EXECUTIONSTATE a různá čísla následujícího stavu, pak tyto přechody musejí být podmíněné (viz dále).
ACTLIST	Seznam výpočtu v daném stavu.
READ_ENABLE	Seznam zdrojů, ze kterých se v daném taktu čte.
WRITE_ENABLE	Seznam zdrojů, do kterých se v daném taktu zapisuje.
ITERATIONCOND	Obsahuje název podmíněné proměnné. Atribut doplňuje podmínku k přechodu do následujícího stavu (viz CHANGESTATE). Podmínka říká: jestliže podmíněná proměnná (cu_cnd_0) nabude hodnoty 1 (v daném stavu výpočtu), pak dojde ke změně stavu. Duálně existuje atribut NEGCOND, který převádí stav, když podmíněna proměnná má hodnotu 0.
IFCOND *(není v příkladu)*	Má stejný význam jako ITERATIONCOND. Duální atribut se nazývá ELSECOND.
SWITCHEXPR, CASE *(není v příkladu)*	Další typ větvení. Od atributů ITERATIONCOND a IFCOND se liší v tom, že umožňuje výběr z více větví (jako konstrukce switch v jazyku C). Hodnotou atributu SWITCHEXPR je název podmíněné proměnné. Hodnotou atributu CASE je hodnota podmínky. Konstrukci default z jazyka C odpovídá atribut CASEDEFAULT. SWITCHEXPR a CASE (nebo CASEDEFAULT) se na hraně objevují současně.

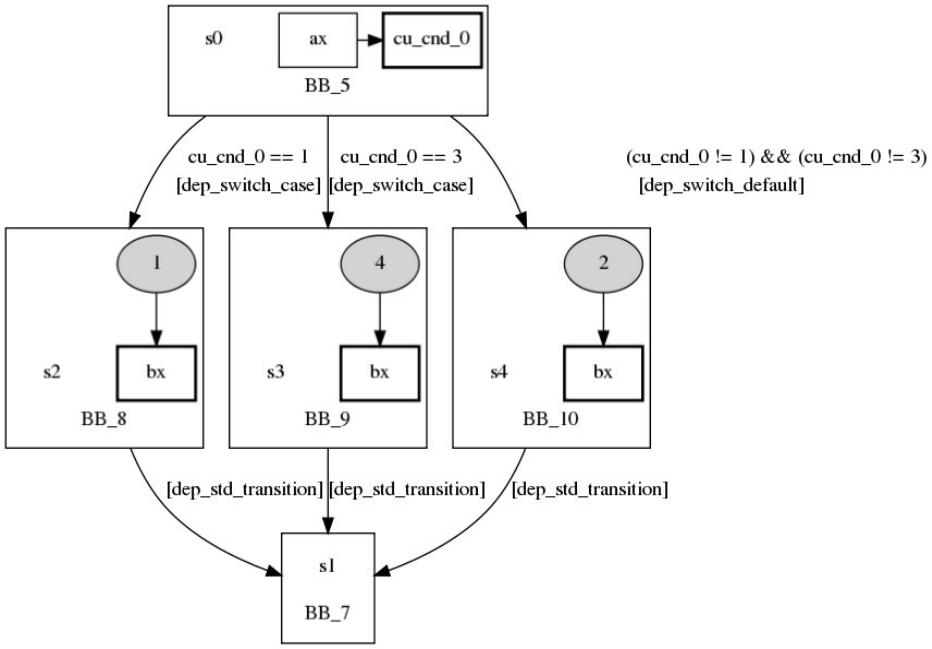
6.1 Příklad funkční jednotky s větvením typu „switch“

Na následujícím příkladu je ukázána transformace příkazu „switch“.

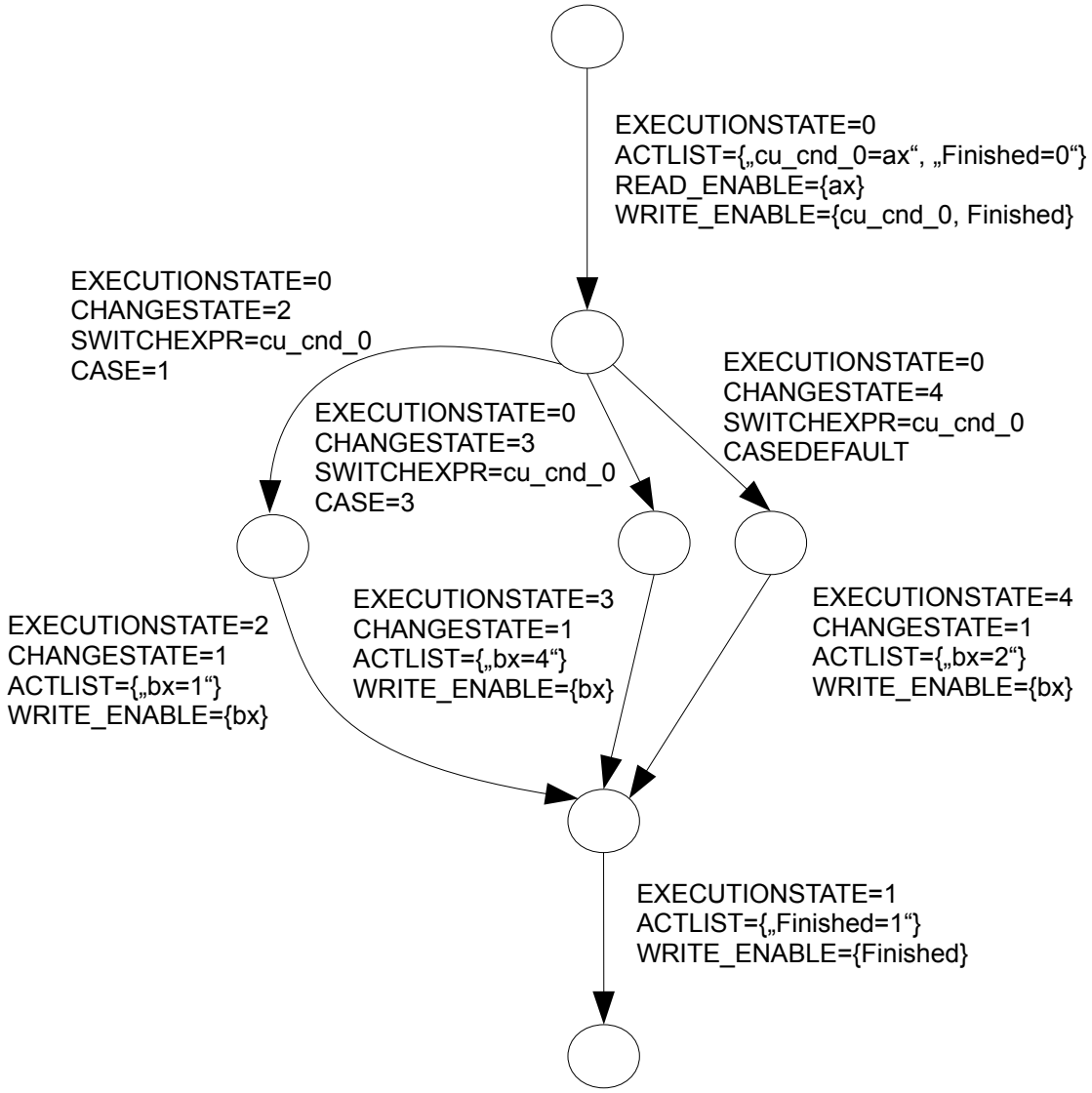
```
BEHAVIOR {
          switch(ax)
          {
          case 1:
               bx = 1;
               break;
          case 3:
               bx = 4;
               break;
          default:
               bx = 2;
               break;
          }
};
```



Obrázek 10: Schéma základních bloků pro konstrukci switch



Obrázek 11: Plánování operace metodou ASAP



Obrázek 12: RTL automat pro příklad operace s konstrukcí switch

6.2 Volání funkcí

Funkce lze volat z operací nebo z jiných funkcí. Musí být dodrženo, že nedojde k rekurzi, protože jazyk ISAC nemá implicitní podporu pro zásobník.

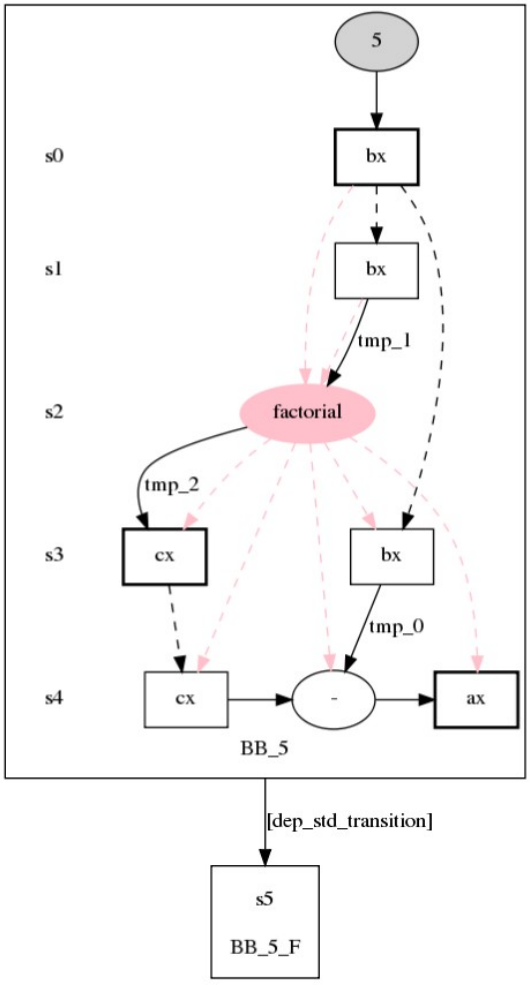
Zvláštností u volání funkcí je to, že volající operace musí čekat na dokončení výpočtu funkce. Doba výpočtu funkce (v taktech) nemusí být konstantní, jak je zvykem např. u přístupu do registru. Variabilní doba souvisí s větvením a cykly.

Oproti teorii překladačů se náš přístup k funkcím liší v tom, že volání funkce neukončuje základní blok. V budoucnu se počítá s tím, že této vlastnosti bude využito při plánování.

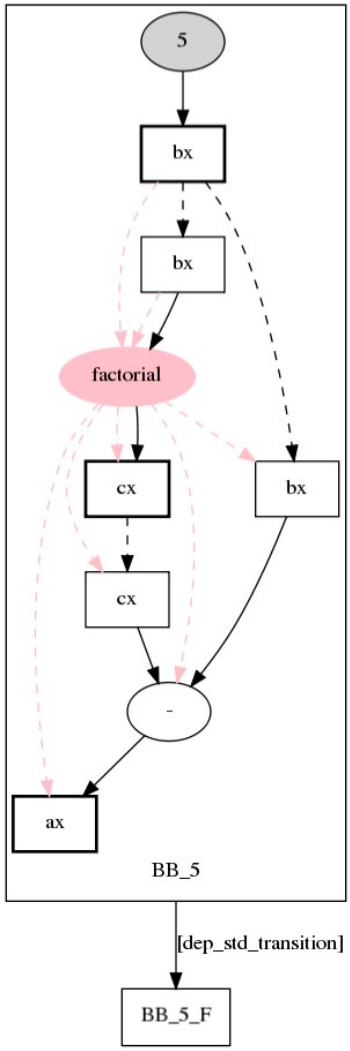
V následujícím příkladu je použito volání funkce factorial:

```
BEHAVIOR {
          bx=5;
          cx=factorial(bx);
          ax=cx-bx;
};
```

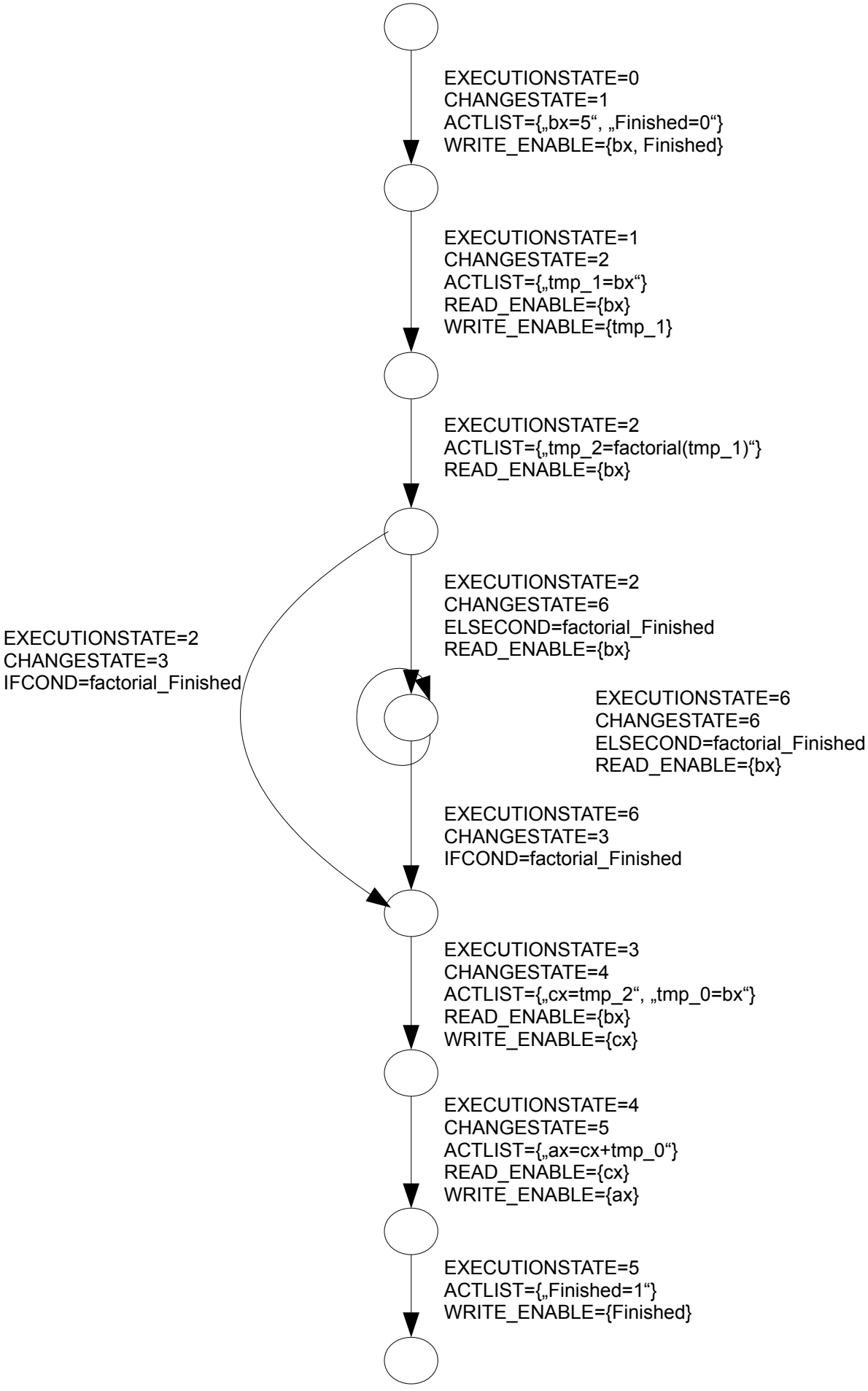
Na obrázku 13 je CDFG této behaviorální sekce. Volání funkce factorial je v růžovém oválu. Pomocí růžových hran je zajištěno, že volání funkce bude po naplánování metodou ASAP ve svém vlastním taktu (viz obrázek 14).



Obrázek 14: Plánování volání funkce metodou ASAP



Obrázek 13: CDFG pro volání funkce



Obrázek 16: RTL automat s voláním funkce

Na obrázku 16 se nachází RTL automat pro behaviorální sekci s voláním funkce. V automatu jsou obsaženy některé záležitosti, které nebyly diskutovány v předchozích kapitolách:

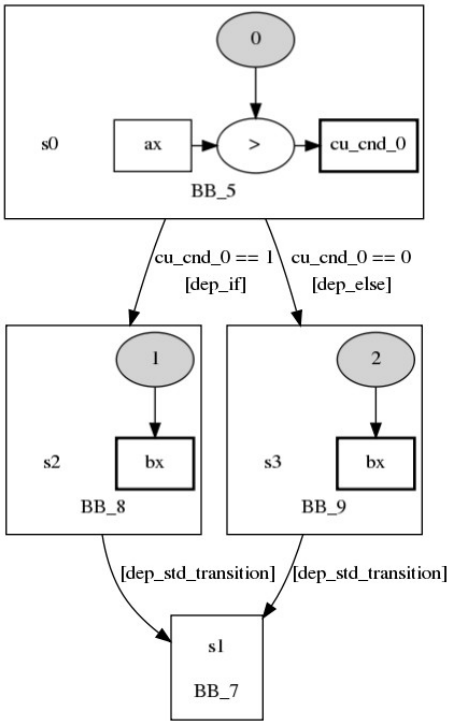
- Automat obsahuje pomocné proměnné. Pokud je potřeba přenést hodnotu do následujícího taktu, pak se hodnota uloží do pomocné proměnné. V hardwaru bude pomocná proměnná transformována na signál (tedy nedojde k dodatečné režii – vytvoření dodatečného registru).
- Pomocné proměnné nejsou obsaženy v seznamech READ_ENABLE a WRITE_ENABLE. Nejedná se totiž o fyzické zdroje, které by mohly mít povolovací vstup.
- Ve stavu 2 je volána funkce factorial. V hardwarové implementaci bude volající funkční jednotka pozorovat signál Finished od funkce factorial (factorial_Finished). Funkce factorial nastaví factorial_Finished na 1 při skončení výpočtu. Volající funkční jednotka tím zjistí, že výsledek výpočtu funkce factorial je k dispozici. Čekání na factorial_Finished je v automatu implementováno čekacím cyklem. Čekací cyklus obsahuje podmínku „if“. Poznámka: některé funkce mohou skončit okamžitě (ve stejném taktu, ve kterém byly aktivovány). Proto je před vstupem automatu do čekacího stavu (6) testován signál Finished funkce factorial.

6.3 Optimalizace control-flow grafu

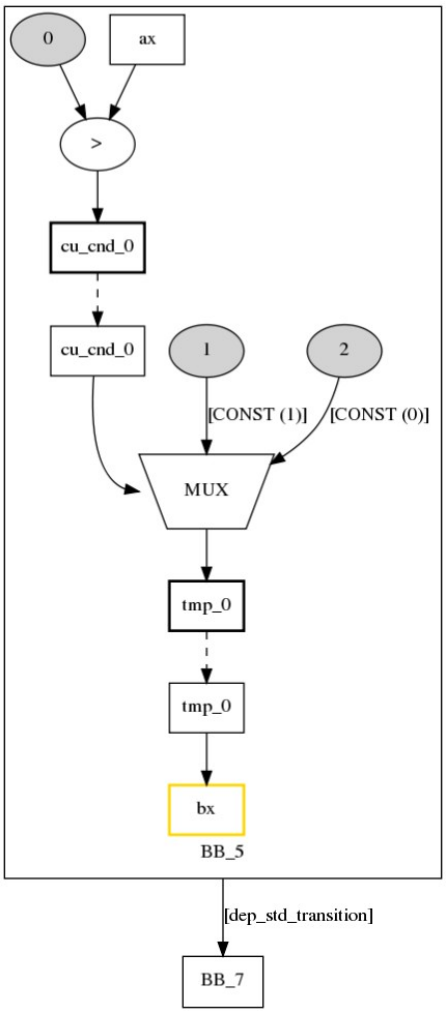
Tato podkapitola se věnuje optimalizaci control-flow grafu z pohledu doby výpočtu. Doba výpočtu je měřena v taktech. Snížení doby výpočtu je dosaženo pomocí optimalizace větvení.

Optimalizace je naznačena na příkladu operace s podmínkou „if“:

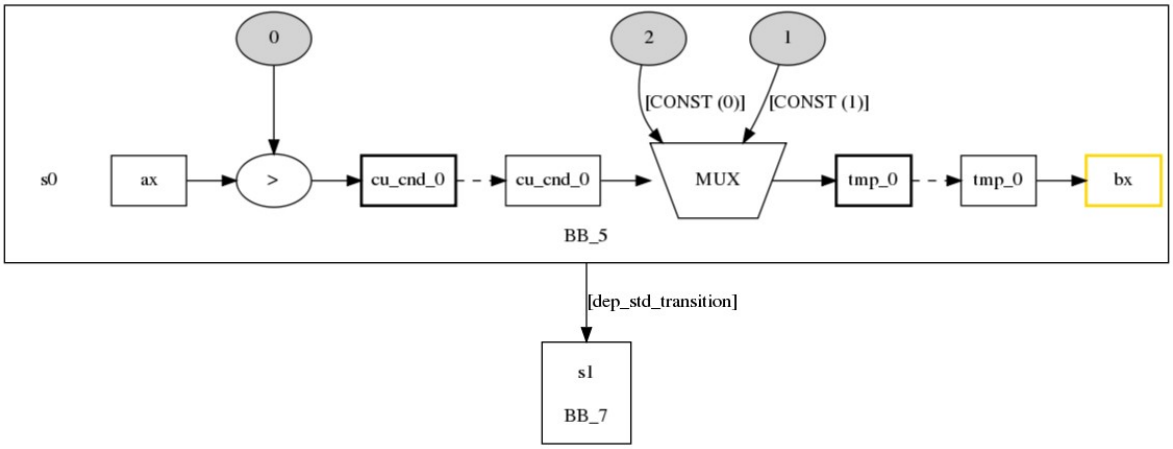
```
BEHAVIOR {
     if(ax>0)
       bx=1;
     else
       bx=2;
};
```



Obrázek 17: Neoptimalizovaná verze (ASAP)



Obrázek 18: Optimalizovaná verze (nenaplánovaná)



Obrázek 19: Optimalizovaná verze (ASAP)

Na obrázku 17 je zachycen CDFG pro neoptimalizovanou verzi výpočtu. Pro vykonání výpočtu jsou potřeba 2 takty: s0 + (s2 nebo s3). Na obrázku 19 je optimalizovaná verze téže operace naplánována do 1 taktu. Na obrázku 18 se nachází optimalizovaná verze před plánováním.

Princip optimalizace:

1. Optimalizátor nalezne větvení výpočtu („if“ nebo „switch“)
2. Optimalizátor zjistí, zda lze přesunout některé výpočty z větví do rodičovského bloku. V příkladu bylo možné přesunout zápisy konstant do registru bx do rodičovského bloku.
3. Optimalizátor přesune vybrané výpočty, doplní tzv. multiplexor zapisované hodnoty a doplní podmínku zápisu (nebo čtení) do zdroje.

Význam multiplexoru spočívá ve výběru hodnoty, která bude předána na jeho výstup. V příkladu má multiplexor tuto funkci: „Jestliže hodnota podmíněné proměnné cu_cnd_0 nabývá konstanty 1, pak na výstup pošli konstantu 1. V opačném případě (cu_cnd_0 je 0) pošli na výstup konstantu 2.“ Výstup multiplexoru je zapsán do pomocné proměnně tmp_0. Z pomocné proměnné se v HW stane signál (tedy nedojde k dodatečným nárokům na zdroje).

Zdrojům je ještě potřeba doplnit podmínku zápisu. Podmínka zápisu pro registr bx v příkladu zní: „Zápis je povolen, pokud proměnná cu_cnd_0 má hodnotu 0 nebo 1.“ Při generování HW bude k této podmínce připojena podmínka stavu, tedy „Zápis je povolen, pokud se funkční jednotka nachází ve stavu s0 a proměnná cu_cnd_0 má hodnotu 0 nebo 1.“ Oranžové orámování na obrázku označuje přítomnost podmínky.

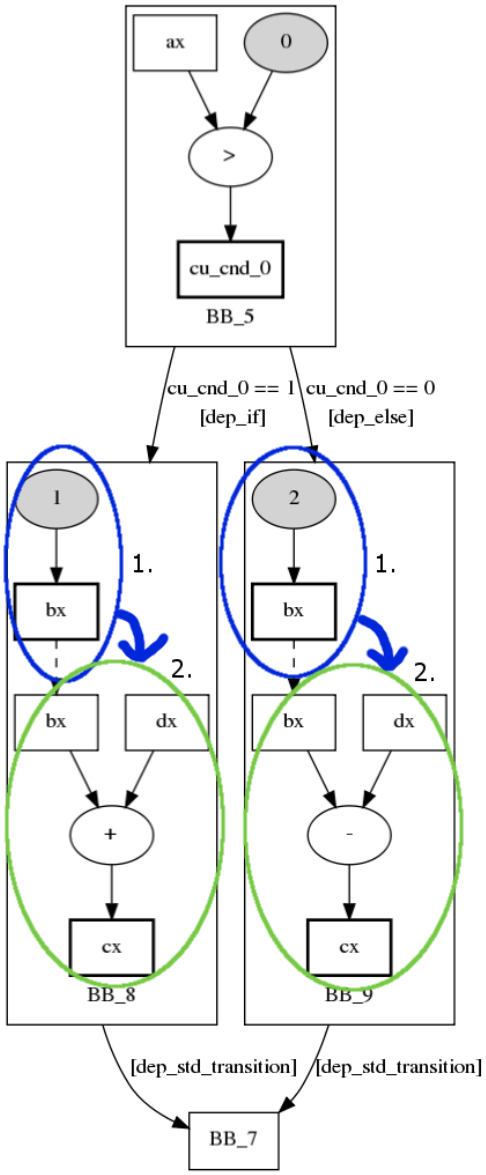
Některé zdroje potřebují také podmínku čtení. Například čtení z paměti musí být explicitně povolováno (na rozdíl od čtení z registru). V takovém případě je podmínka čtení doplněna také.

6.4 Pokročilé možnosti optimalizace

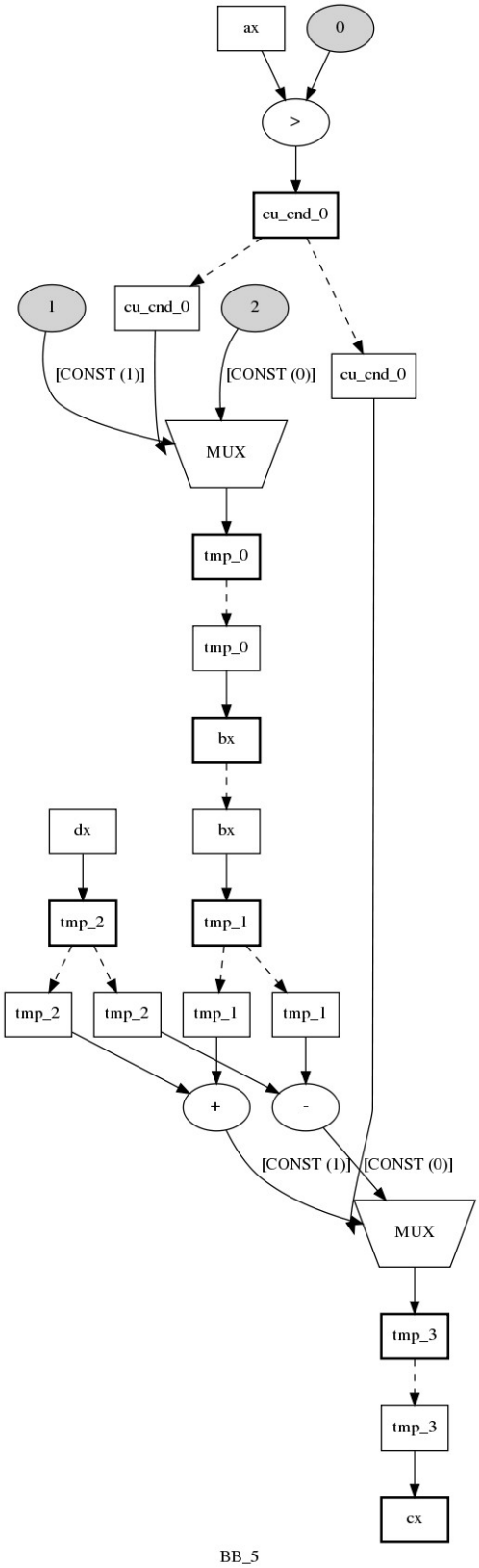
V předchozí kapitole je uveden jednoduchý příklad, ve kterém je přesunut zápis do registru bx do rodičovského bloku. Optimalizátor umí přesunout i více zápisů z podmíněných větví (bloků).

Tato vlastnost je demonstrována na následujícím příkladu:

```
BEHAVIOR {
     if(ax>0)
     {
      bx=1;
      cx=bx+dx;
     }
     else
     {
      bx=2;
      cx=bx-dx;
     }
};
```



Obrázek 20: Před optimalizací



Obrázek 21: Blok BB_5 po optimalizaci bloků BB_8 a BB_9

Algoritmus pracuje takto:

1. Podmíněné bloky rozdělí na subbloky. Na obrázku 20 jsou subbloky označeny barevnými elipsami. Subblok je tvořen uzlem zápisu do zdroje (nebo volání funkce) a uzly, na kterých je datově závislý (tranzitivně). Na obrázku: uzly zpětně-dosažitelné ze zápisového uzlu po plných šipkách.
2. Mezi subbloky vytvoří závislosti. Závislost mezi subblokem A a subblokem B existuje tehdy, když existuje závislost mezi nějakým uzlem v A a nějakým uzlem v B. Na obrázku 20 jsou závislosti vyznačeny modrou šipkou – zelené subbloky jsou závislé na modrých subblocích.
3. Uspořádá subbloky podle topologického třídění. (Subbloky bez závislostí jsou první v pořadí.) Na obrázku 20 je pořadí zapsáno vedle barevných elips.
4. Algoritmus se pokouší přesunout jednotlivé subbloky do rodičovského bloku. Provádí to v pořadí, které si vypočítal v předchozím kroku. Subblok je přesunut do rodičovského bloku, pokud tam byly přesunuty všechny subbloky, na kterých je závislý.

Výsledek algoritmu na uvedeném příkladu je zobrazen na obrázku 21. Z obrázku je patrné, že původní závislosti jsou dodrženy, což je klíčové.

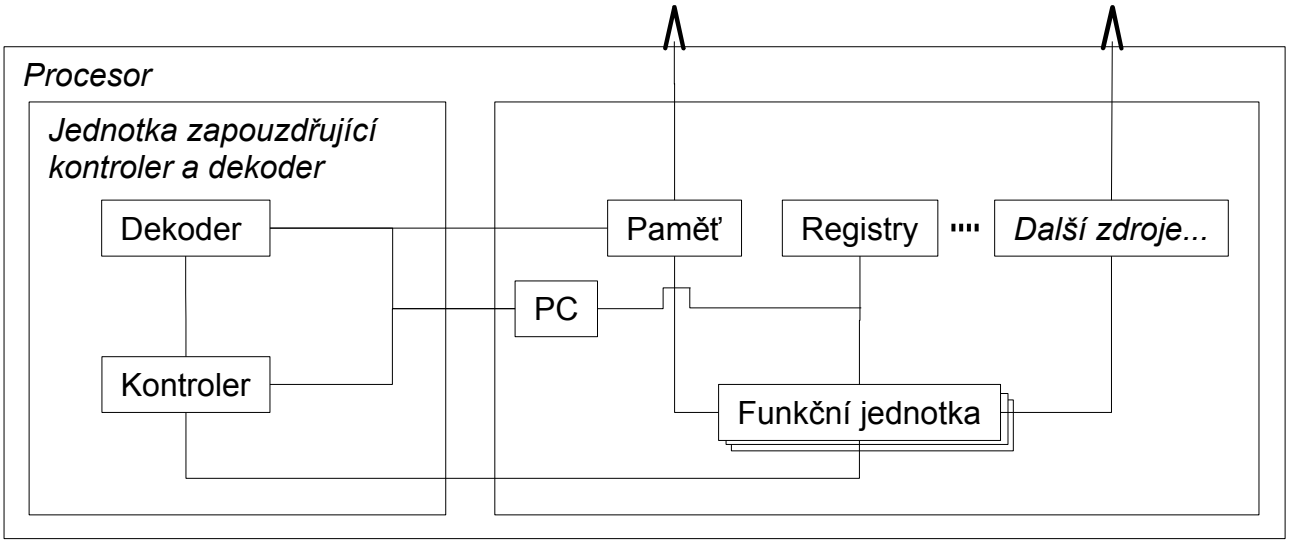
7 Generátor hardware

Tato kapitola popisuje zadní část překladače – tzv. generátor HW. Generátor HW je implementován v knihovně isac2hwl.

Vstupy generátoru HW jsou:

- zdroje,
- časované RTL automaty vytvořené v knihovně lbast2rtlal (funkční jednotky),
- datové vazby mezi operacemi s EXPRESSION sekcí,
- porty pro jednotku zapouzdřující kontroler a dekoder

Generátor vytvoří popis HW v jazyku VHDL. Popis je hierarchický. Schéma je znázorněno na obrázku 22.



Obrázek 22: Schéma procesoru

Zdroje (deklarované v RESOURCE sekci) jsou:

- Registry. Mezi významné registry patří program counter (PC).
- Registrová pole. Od registrů se liší tím, že z nich lze číst z omezeného počtu portů. Podobně jsou omezeny zápisové porty. Typicky bývá počet portů pro čtení 2, pro zápis 1. V jednom taktu by šlo číst nejvýše ze 2 registrů v poli a zapisovat do jediného.
- Paměti jsou uvažovány dvojího typu. Jednoportové a víceportové.
- Porty procesoru. Portem se rozumí vstup či výstup procesoru.
- Signály. Funkční jednotky si mohou předávat data pomocí signálů. Signál není paměťový prvek, a proto obě komunikující funkční jednotky musejí běžet paralelně a být ve stavu umožňujícím komunikaci.

Všechny paměti a porty mají své rozhraní vyvedé do rozhraní procesoru. U portů to je zřejmé. U pamětí se předpokládá, že při inicializaci procesoru dojde k nahrání programu nebo dat do pamětí.

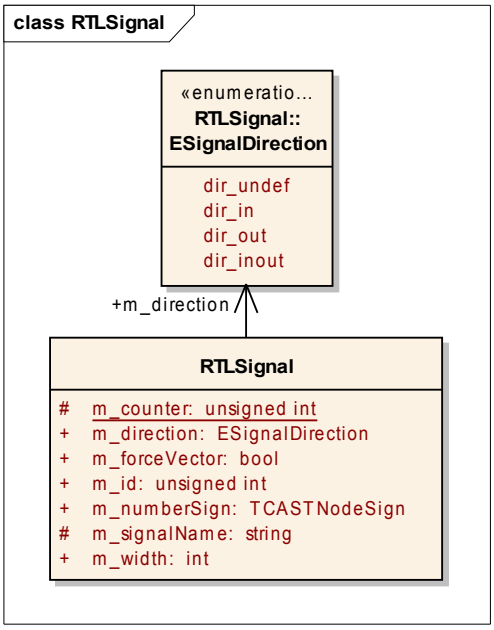
Na obrázku 23 je zobrazen postup generování HW. V následujících kapitolách jsou popsány komponenty HW a souvislost mezi nimi. Propojení funkčních jednotek a jednotky zapouzřující kontroler a dekoder popisováno nebude. Jedná se totiž o pouhé propojení signálů.

7.1 Komponenty popisu HW

Pro abstraktní zdroje definované v RESOURCE sekci jazyka ISAC jsou vytvořeny objekty, které odpovídají HW reprezentaci.

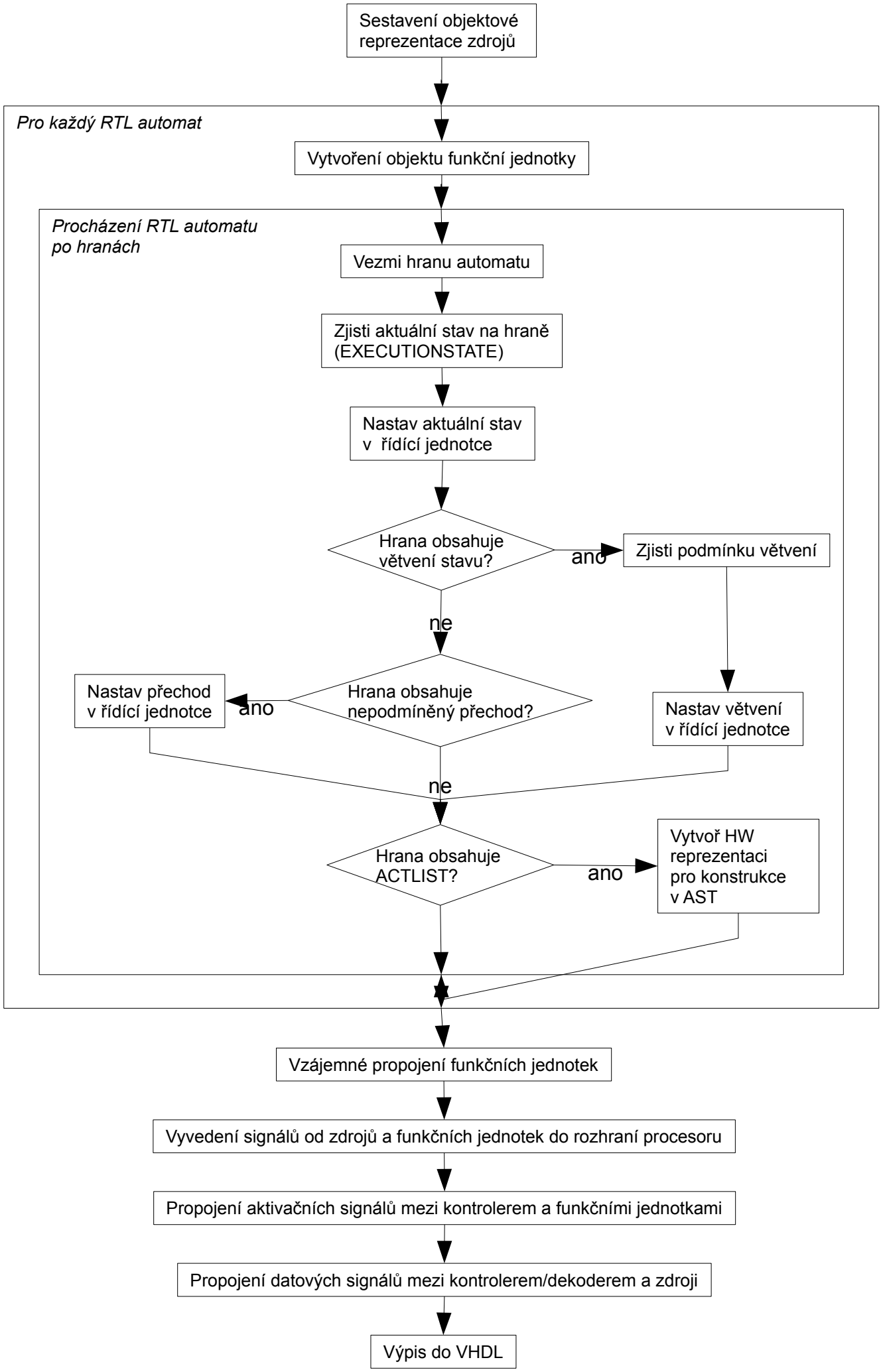
7.1.1 Signál

Na obrázku 24 je zobrazena třída pro signál. Komponenty HW jsou spojeny pomocí signálů.



Obrázek 24: Třída RTLSignal

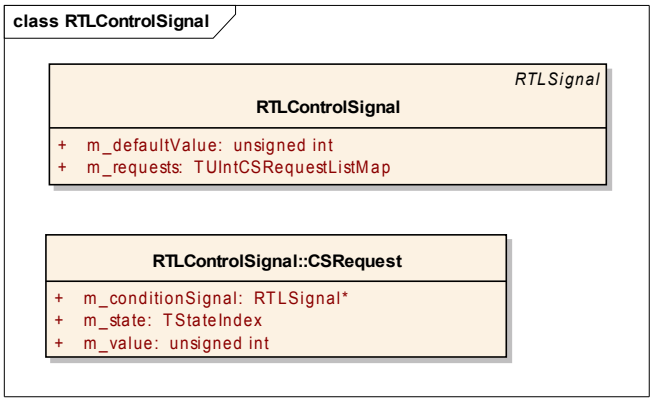
Název atributu	**Význam**
m_counter	Číslování pomocných signálů.
m_signalName	Název
m_id	Identifikátor
m_width	Bitová šířka
m_direction	Směr signálu vzhledem k rodičovské komponentě
m_numberSign	Znaménko hodnoty
m_forceVector	Pro VHDL: 1b signál lze zapsat jako signál nebo vektor. Pokud je true, pak 1b signál bude vektor.



Obrázek 23: Postup generování hardware

7.1.2 Výběr hodnoty na základě stavu (varianta multiplexoru)

Třída RTLControlSignal implementuje výběr hodnoty na základě stavu řídící jednotky. Zvolená hodnota je zapsána do signálu (na výstup). V HW je tato komponenta implementována pomocí multiplexoru nebo vyhledávací tabulky. Objekt třídy RTLControlSignal si udržuje seznam požadavků. Požadavky jsou trojice (stav řídící jednotky, hodnota, dodatečná podmínka). Diagram třídy je na obrázku 25.



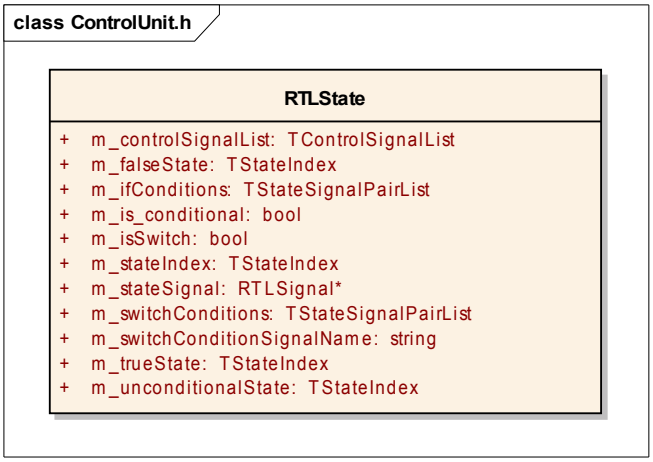
Obrázek 25: Výběr hodnoty signálu podle stavu řídící jednotky

Název atributu	Význam
m_requests	Seznam požadavků
m_defaultValue	Implicitní hodnota

Název atributu	Význam
m_state	Stav řídící jednotky
m_value	Požadovaná hodnota
m_conditionSignal	Dodatečná podmínka (nemusí být nastavena)

7.1.3 Řídící jednotka

Řídící jednotka je tvořena grafem stavů. Jedná se o stavy vypočítané plánovačem pomocí algoritmu ASAP. Na obrázku 26 je diagram třídy.

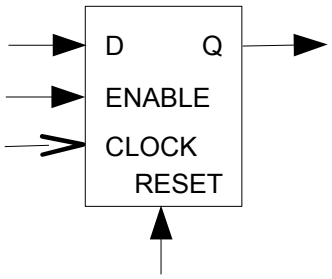


Obrázek 26: Stav řídící jednotky

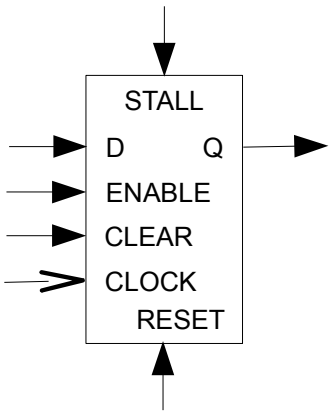
Název atributu	Význam
m_is_conditional	Větví se tento stav na více stavů?
m_isSwitch	Jde o větvení typu switch?
m_stateIndex	Číslo stavu
m_trueState	Číslo stavu pro větvi then.
m_falseState	Číslo stavu pro větvi else.
m_unconditionalState	Číslo stavu pro nepodmíněný přechod
m_switchConditionSignalName	Název signálu, který nese porovnávanou hodnotu pro switch.
m_switchConditions	Seznam dvojic: (následující stav, signál 0/1 indikující, zda dojde k přechodu)
m_ifConditions	dtto
m_controlSignalList	*nepoužívá se*
m_stateSignal	Signál, který informuje o aktuálním stavu řídící jednotky.

7.1.4 Registr

V HW implementaci existuje několik druhů registrů. Generátor HW vytváří D-klopný obvod pro registr. Podle použití se D-registr liší v počtu vstupních signálů, které jej ovládají. Na obrázku 27 se nachází běžný registr. Na obrázku 28 se nachází registr, který byl použit v lince zřetězení (pipeline). Linka zřetězení vyžaduje pozastavování (stall) a synchronní vymazání obsahu (clear). Popis chování registrů v jazyku VHDL je pod obrázky.



Obrázek 27: D-registr (varianta 1)



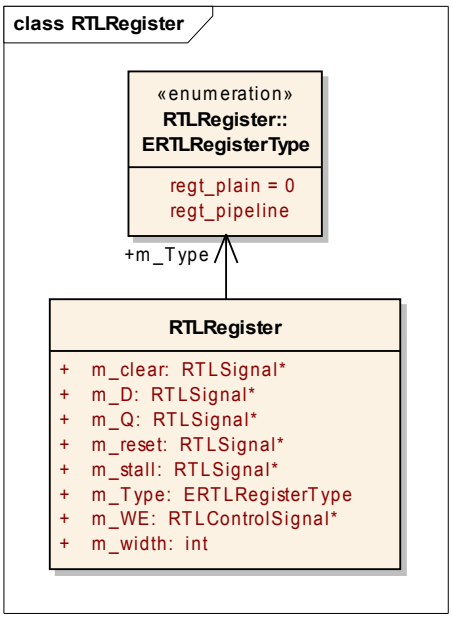
Obrázek 28: D-registr (varianta 2)

```
PROCESS(Clock, Enable, Reset, D)
BEGIN
  IF (Reset = '0') THEN
    Q <= (others => '0')
  ELSE
    IF (rising_edge(Clock)) THEN
      IF (Enable = '1') THEN
        Q <= D;
      END IF;
    END IF;
  END IF;
END PROCESS;
```

```
PROCESS(Clock, Enable, Reset, Clear,
        Stall, D)
BEGIN
  IF (Reset = '0') THEN
    Q_local <= (others => '0')
  ELSE
    IF (rising_edge(Clock)) THEN
      IF (Clear = '1') THEN
        Q_local <= (others => '0');
      ELSIF (Stall = '1') THEN
        Q_local <= Q_local;
      ELSIF (Enable = '1') THEN
        Q_local <= D;
      END IF;
    END IF;
  END IF;
END PROCESS

Q <= Q_local;
```

Na obrázku 29 se nachází diagram třídy registru (s vybranými atributy). Pro obě varianty existuje jediná třída.



Obrázek 29: Třída registru

Název atributu	Význam
m_Type	Varianta registru
m_width	Počet bitů
m_D	Vstupní signál
m_Q	Výstupní signál
m_WE	Povolení zápisu
m_reset	Asynchronní vymazání (aktivní v 0)
m_clear	Synchronní vymazání (aktivní v 1)
m_stall	Zakázání zápisu

7.1.5 Registrový soubor (register file)

Registrový soubor lze deklarovat v `RESOURCE` sekci jazyka ISAC. Rozdíl oproti paměti je v tom, že každý požadavek na čtení nebo zápis, je vyřízen v 1 taktu.

Příklad:

```
// registrové pole o 8 prvcích
REGISTER     bit[8] gprs [8]
{
      DATAPORT (2,1);
};
```

V příkladu bylo deklarováno registrové pole `gprs` o 8 registrech. Každý registr má šířku 8 bitů. Z registru lze paralelně číst na 2 portech. Zapisovat lze pomocí 1 portu.

Ve funkčních jednotkách lze z registrového pole číst nebo zapisovat. Adresa registru se zadává do hranatých závorek.

Příklad:

```
OPERATION OP1
{
      ASSEMBLER { "OP1" };
      CODING { 0b00000011 };
      BEHAVIOR
      {
            cx = gprs[ax] + gprs[2];
            gprs[bx] = 4;
      };
}
```

7.1.5.1 Souběžný přístup k registrovému poli

Funkční jednotky mohou do registrového pole přistupovat nezávisle na sobě. Vznikají tím dva problémy:

1. souběžný zápis do stejného registru,
2. více požadavků na čtení (resp. zápis) než kolik je dostupných čtecích (resp. zápisových) portů.

Problém souběžného zápisu do stejného registru je vyřešen při simulaci celého procesoru. Jedná se o chybu návrhu procesoru. Tato chyba je odhalena během simulace.

Problém více požadavků o přístup lze rozdělit na 2 podproblémy:

1. Funkční jednotka požaduje víc hodnot registrů, než kolik je portů registrového souboru. Například: `cx = gprs[ax] + gprs[2] + gprs[3];` odpovídá 3 paralelním čtení ze souboru s pouze 2 čtecími porty. Tento typ chyby je odhalen při překladu.

2. Jednotlivé funkční jednotky nepožadují přístup k více portům zároveň, jako v bodu 1. Avšak při běhu procesoru dojde k situaci, kdy součet požadovaných čtení (resp. zápisů) ze všech funkčních jednotek je větší než počet čtecích (resp. zápisových) portů registrového souboru. Tento typ chyby je odhalen při simulaci. Jedná se o chybu návrhu procesoru.

Ostatní přístupy k registrovému souboru jsou korektní. V následujících kapitolách je popsán efektivní způsob řešení.

7.1.5.2 Návrh řešení

Analýzou behaviorální sekce operace se zjistí, ke kolika portům pro čtení (resp. zápis) je v 1 taktu nejvýše přistupováno.

Příklad:

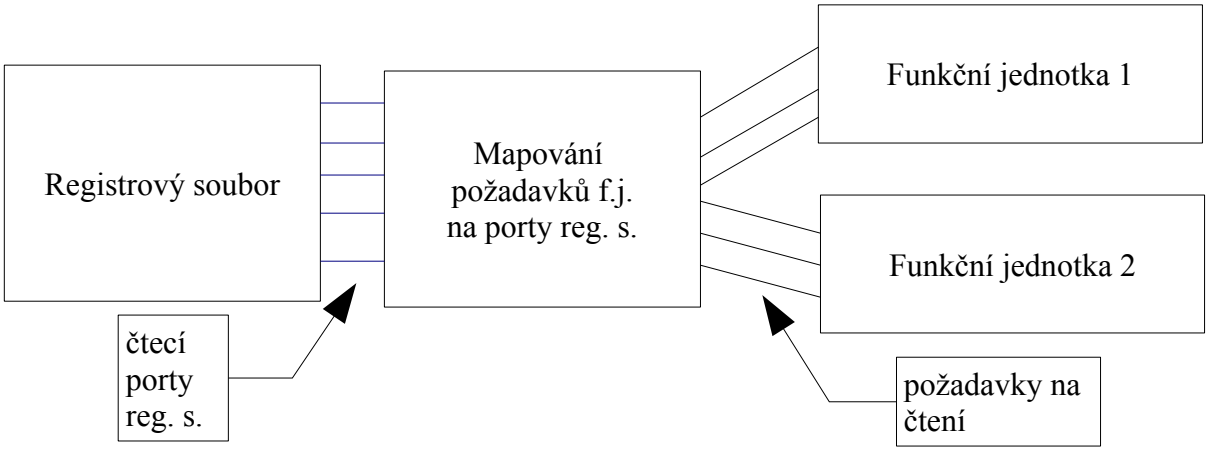
(Předpokládejme, že registrový soubor má celkem 5 čtecích portů)

```
OPERATION OP3
{
      BEHAVIOR
      {
            cx = gprs[ax] + gprs[2];
            dx = gprs[bx] + gprs[cx] + gprs[3];
      };
}
```

Nejvýše je přistupováno ke 3 portům v 1 taktu (na 2. řádku). Funkční jednotka pro operaci `OP3` tedy potřebuje přístup ke 3 čtecím portům registrového souboru.

Nastává ovšem problém: nelze předem určit, ze kterých portů registrového souboru má být čteno. V jiné funkční jednotce může být totiž požadováno čtení na stejných portech, ale z jiných registrů. Tím by došlo ke kolizi.

O tom, na jakém portu registrového souboru bude požadavek funkční jednotky vyřízen, se rozhodne dynamicky za běhu. V ohled se vezmou všechny požadavky funkčních jednotek v daném taktu. Na obrázku 30 je znázorněn způsob komunikace funkčních jednotek s registrovým souborem. Požadavky funkčních jednotek jdou přes blok Mapování... . Stejně jako u běžného přístupu k registrovému souboru je i nyní požadováno, aby všechny požadavky byly vyřízeny v 1 taktu.



Obrázek 30: Logické schéma přístupu k registrovému souboru

7.1.5.3 Blok Mapování požadavků funkčních jednotek na porty registrového souboru

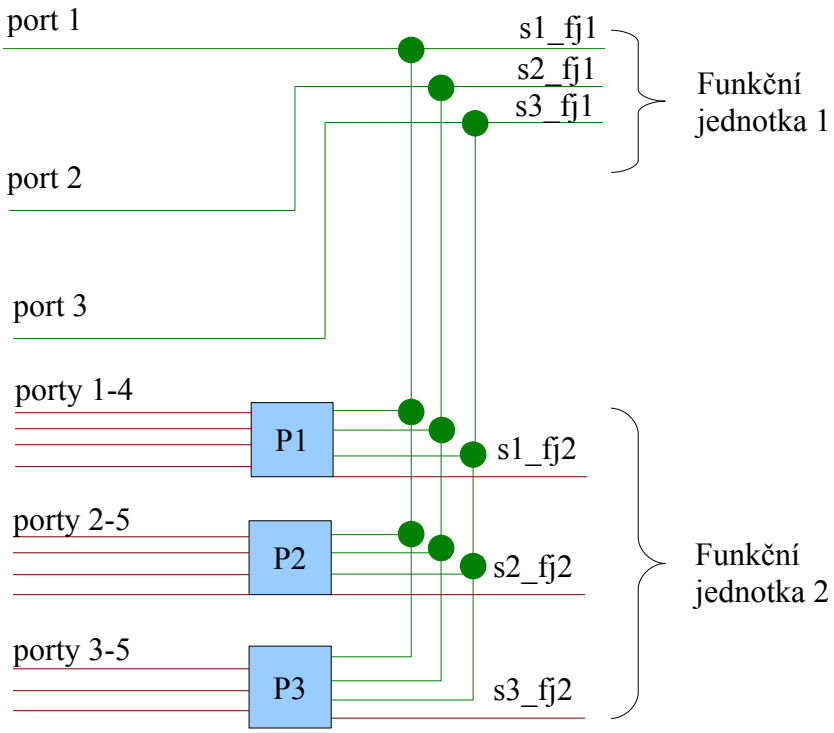
Způsob zadávání požadavků ve funkčních jednotkách:

1. Z funkční jednotky vedou signály odpovídající požadavkům (viz předchozí obrázek).
2. Signály odpovídající požadavkům ve funkční jednotce mají dáno pořadí od `1` do `k`.
3. Funkční jednotka umisťuje požadavky na signály tak, že nejprve obsadí signál `1`, pak `2` atd. (Díky tomu bude možné optimalizovat zapojení bloku Mapování...).

Následující obrázek zobrazuje optimalizované schéma bloku Mapování... . Zelenou barvu mají signály z funkční jednotky 1, červenou mají signály z jednotky 2. Registrový soubor má 5 portů, o které se funkční jednotky dělí. Uvažujme o tom, že jde o požadavky o čtení z registrového souboru. Požadavky o zápis se vyřizují stejným způsobem.

Signál `s1_fj1` je mapován na port 1. Podobně signály `s2_fj2` a `s3_fj1` jsou mapovány na porty 2 a 3. Signál `s1_fj2` může být namapován na porty 1 až 4, podle toho, zda funkční jednotka 1 žádala na signálech `si_fj1`. Signál `s2_fj2` může být namapován na porty 2 až 5. Na port 1 jej zcela jistě namapovat nelze, protože jej obsadí signál `s1_fj2` (viz bod 3 v předchozím seznamu).

Bloky P1, P2, P3 přijmou na vstupu signály požadavků. Vyhodnotí počet nastavených požadavků. Výstupem bloku je svazek signálů. V tomto svazku je nastaven nejvýše jeden signál na 1 ostatní na 0. Signál s hodnotou 1 označuje port, který je přiřazen odpovídajícímu červenému signálu (požadavku funkční jednotky 2).



Obrázek 31: Schéma implementace bloku Mapování požadavků na fyzické porty

7.1.5.4 Implementace v jazyku VHDL

V předchozích příkladech bylo přistupováno k souboru registrů `gprs`. Do souboru `gprs_RF.vhd` je vygenerována entita a architektura pro registrový soubor `gprs_RF` s parametry specifikovanými v `RESOURCE` sekci jazyka ISAC.

Výpočet prováděný jednotkami `P1, P2, P3` v předchozí kapitole je zajišťován generickou entitou `PortPositionResolver` v souboru `PortPositionResolver.vhd`.

V souboru `gprs.vhd` se nachází entita `gprs`, která obsahuje:

1. logiku bloku Mapování...,
2. instance registrového souboru a
3. instance jednotek `PortPositionResolver` (tj. `P1, P2, P3`).
4. Rozhraní entity `gprs` obsahuje signály požadavků funkčních jednotek.

Požadavek na čtení se skládá ze signálů

1. Read enable (RE),
2. Read address (RA),
3. Port.

Požadavek na zápis se skládá ze signálů:

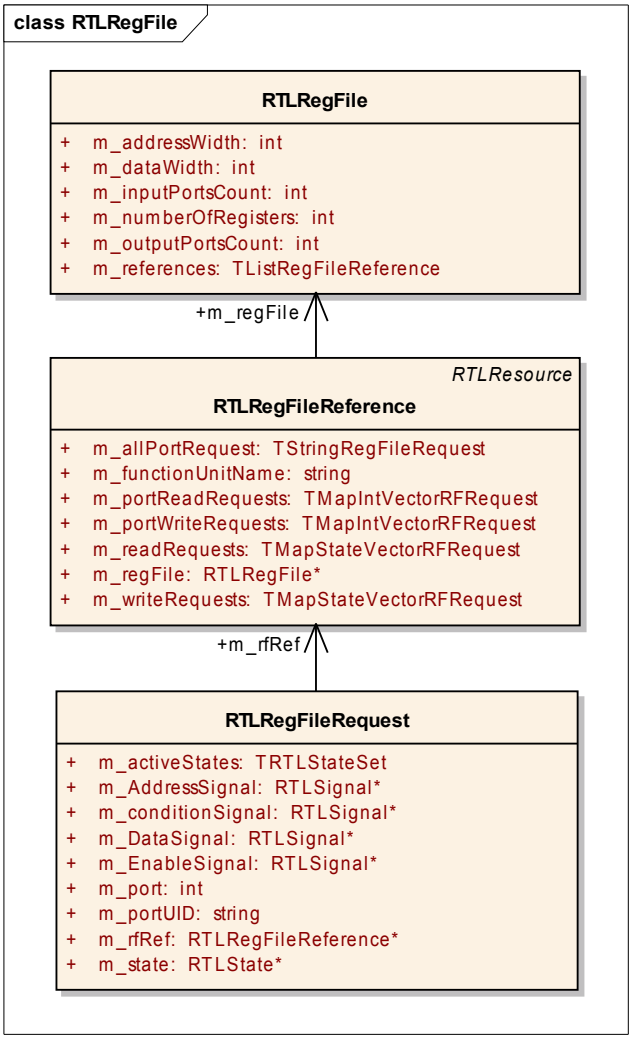
1. Write enable (WE),
2. Write address (WA),
3. Data (D).

7.1.5.5 Objektová reprezentace registrového souboru

Objekt registrového souboru je instancí třídy RTLRegFile. Instance je sdílená všemi funkčními jednotkami, které k registrovému souboru přistupují. Každá funkční jednotka si udržuje referenci na objekt registrového souboru. Reference je instance třídy RTLRegFileReference. Každý požadavek na přístup k registrovému souboru z funkční jednotky je evidován v instanci třídy RTLRegFileRequest. Diagram tříd (s významnými atributy) je zobrazen na obrázku 32.

Před výpisem registrového souboru (a ostatního propojení komponent) je sestavena konečná podoba registrového souboru z instancí tříd RTLRegFile, RTLRegFileReference a RTLRegFileRequest.

- RTLRegFileRequest zastupuje čtení nebo zápis do registrového souboru.
- RTLRegFileReference sdružuje přístupy k registrovému souboru z funkční jednotky.
- RTLRegFile sdružuje instance tříd RTLRegFileReference.



Obrázek 32: Diagram tříd registrového souboru

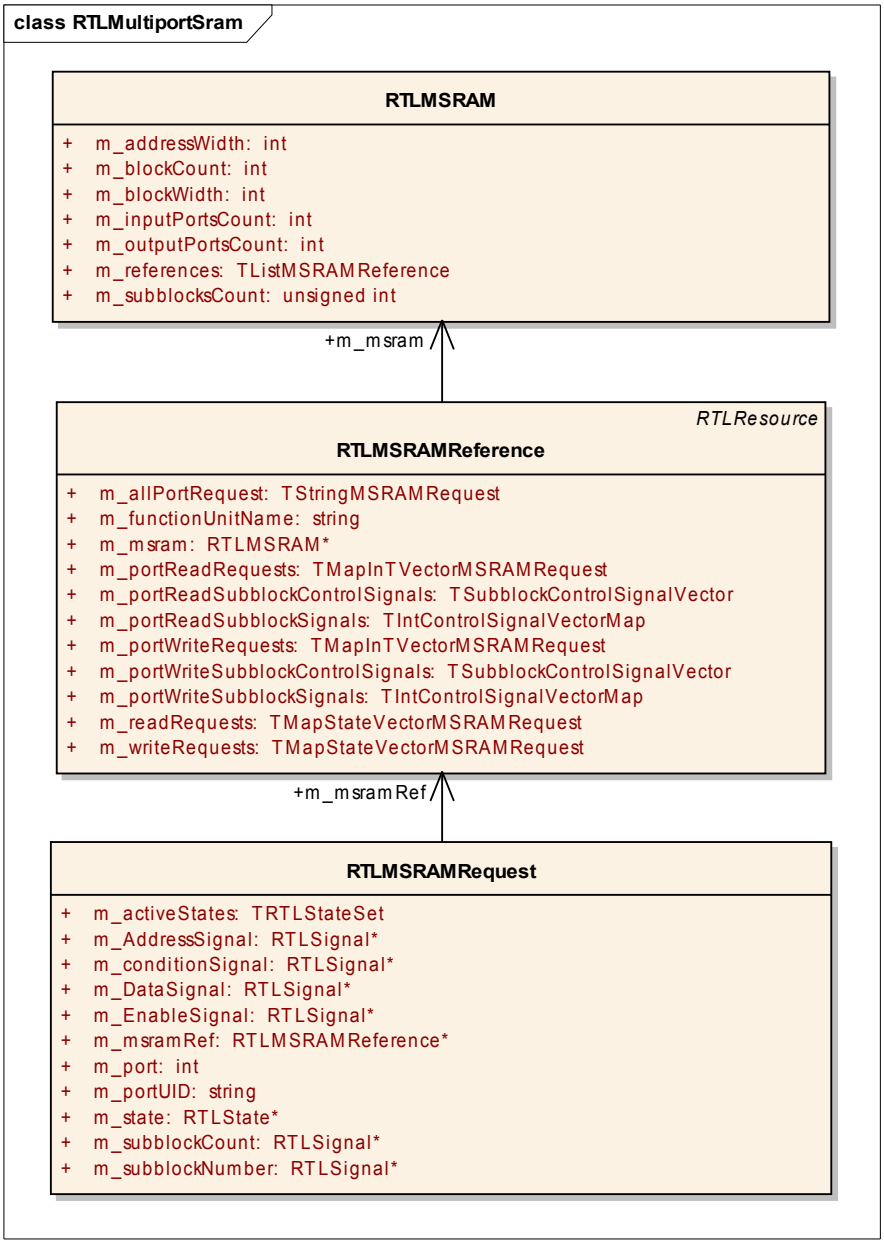
7.1.6 Paměť

Paměť se od registrového souboru liší v těchto bodech:

- lze adresovat subbloky,
- latence přístupu může být větší než 1 takt.

Pro vnitřní reprezentaci se používá stejné logiky, jako pro registrový soubor (viz předchozí kapitola). Požadavky na čtení a zápis jsou navíc obohaceny o adresování subbloků.

Diagram tříd (s významnými atributy) je zobrazen na obrázku 33.



Obrázek 33: Diagram tříd pro paměť

7.1.7 Funkční jednotka

Funkční jednotka se skládá z části výpočtové – datapath a z části řídící – řídící jednotka. Na obrázku 34 je zobrazeno schéma.

Přístupy ke zdrojům a výpočty nad získanými hodnotami probíhá v datapath. Vstupy datapath jsou:

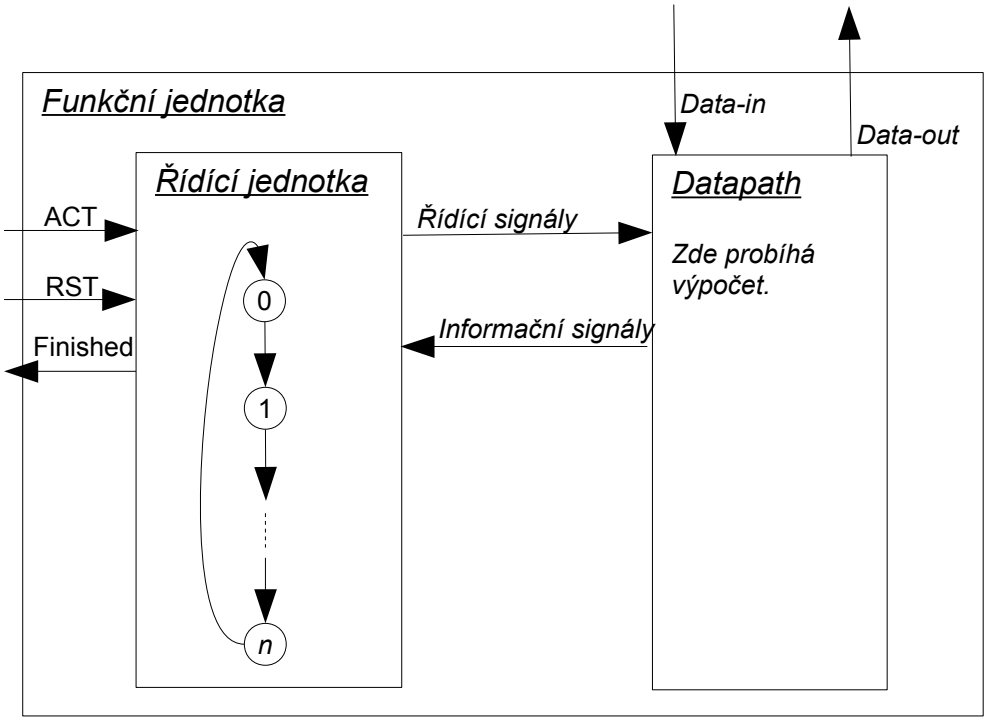
- Řídící signály – nastavují cesty v multiplexorech, povolují čtení a zápisy z/do zdrojů.
- Data-in – vstupní hodnoty z externích zdrojů a operací.

Výstupy datapath jsou:

- Informační signály – informují řídící jednotku o výsledcích komparací. Řídící jednotka podle nich mění stav výpočtu.
- Data-out – zapisovaná data do zdrojů a data předávaná cizím operacím.

Řídící jednotka obsahuje přechodový automat. Automat vychází z RTL automatu. Signály řídící jednotky jsou:

- ACT – aktivační signál. Převádí jednotku z koncového (idle) stavu do počátečního stavu výpočtu.
- RST – resetovací signál. Uvádí jednotku do koncového (idle) stavu.
- Finished. Jednotka informuje okolí, že dorazila do koncového stavu (výpočet skončil).



Obrázek 34: Schéma funkční jednotky

Literatura

[1] Hruška, T., Masařík, K., Zámečníková, E.: ISAC manual (version 2), Fakulta informačních technologií VUT, Brno, 2009