3. Metodika návrhu - vývoj špecifikácií procesorov od systémovej úrovne ku RT úrovni

A. Prístup k návrhu
B. Vývoj špecifikácií pre RT úroveň návrhu
C. Príklad vývoja špecifikácie aktívneho procesora CPU

A. Prístup k návrhu

Digitálny systém môže byť veľmi rozsiahly systém a pri návrhu je potrebné vychádzať z jeho **rozkladu** na jednoduchšie **moduly** (**podsystémy**), k čomu dochádza na systémovej úrovni návrhu. V tomto predmete sa budeme venovať podrobne návrhu HW modulov ako **primárnych architektúr** (pozri Kap.1) na úrovni RT. Primárne architektúry predstavujú „základné kamene“ návrhu digitálnych systémov. Pri návrhu primárnych architektúr (á priori) predpokladáme existenciu kompozície jej dvoch častí: operačnej a riadiacej (Kap.1).

Metóda návrhu operačnej časti (OČ) a riadiacej čas5 (RČ) sa **značne odlišuje**. Pri **návrhu OČ** vychádzame z tzv. **toku údajov** (**dát**), v ktorom je explicitne, vhodným spôsobom, vyjadrené sekvencia a súbežnosť exekúcie agentov (alebo operácií nad ich vstupnými a operačnými stavovými premennými) v jednotlivých riadiacich cykloch prebiehajúcich v OČ pri exekúcií jednotlivých procesov. Z toku údajov priamo vyplývajú údajové (vstupno-výstupné) prepojenia agentov (operácií). Z poznania toku údajov a jeho zápisu (obvykle vo forme grafu) možno **zostaviť štruktúru OČ** (t.j. vybrať príslušné funkčné a iné prvky a zostaviť ich prepojenie) a **optimalizovať syntézu** OČ a celej primárnej architektúry.

K **návrhu riadiacej časti** (RČ) sa pristupuje ako k návrhu **stavového stroja**, ktorého vstupné signály sú státusové signály z OČ a výstupné signály sú riadiace signály pre jednotlivé funkčné časti a prepojovacie časti pre údajové cesty o OČ. Základom behavioristickej špecifikácie samotnej RČ je spravidla konečný stavový stroj - FSM (konečný automat) zapísaný v niektorej forme (napr. v prechodovej tabuľke, v návrhovom jazyku VHDL). Prechodová a výstupná funkcia je špecifikovaná v **toku riadenia**, ktorý sa odvodí zo špecifikácie celého systému.

Pri návrhu primárnej architektúry vychádzame z **formálne zapísanej špecifikácie** systému, ktorá (spravidla) vznikne po **vývoji** (**zjemnení**) špecifikácie na systémovej úrovni návrhu. V tejto kapitole uvedieme v podstate dva príklady, na ktorých ilustrujeme vývoj (zjemnenie) špecifikácií prostredníctvom nami zavedeného modelu spracovania informácie v reaktívnych digitálnych systémoch (resp jazyka HSSL). V praxi to prebieha v niektorom **návrhovom jazyku**, v súčasnosti v systémovom návrhom jazyku SSDL (System Specification and Design Language), ako je UML-2, SystemC, HandelC, SpecC, Esterel, HSSL a pod.).

Špecifikácia sa na systémovej úrovni postupne rozvinie (zjemní) do podoby **cieľovej špecifikácie**, ktorú použijeme ako **vstup** pre **syntézu** štruktúrnej implementácie systému na RT a nižších úrovniach návrhu. Špecifikáciu v jazyku SSDL treba obvykle preložiť do vhodného jazyka pre RT úroveň. Takýmito jazykmi sú dnes všeobecne používané jazyky VHDL a VERILOG. Možno povedať, že tieto jazyky sú typické pre aplikácie pri HW návrhu na RT a logickej úrovni. Pre

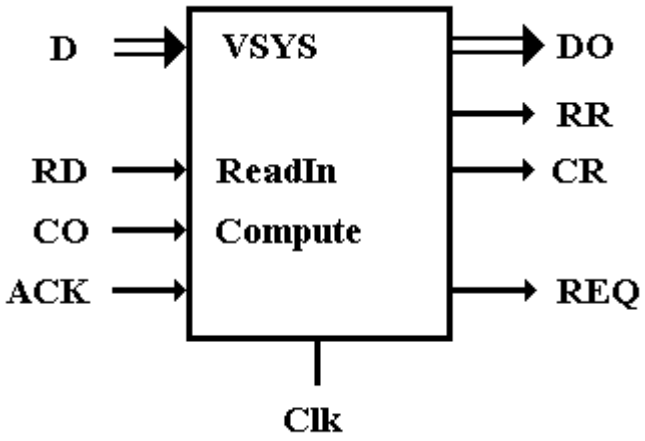
systémovú úroveň ako úroveň najvyššej abstrakcie sú menej vhodné najmä preto, že sú z hľadiska opisu ťažkopádne, nie sú vhodné na opis SW implementovaných modulov a pre simuláciu sú pomalé

Často cieľová špecifikácia obsahuje **mikrooprerácie** a procesy, ktoré ako svoje prvky majú mikrooperácie, t.j. vyjadrujú kompozíciu exekúcie mikrooperácií (tzv. **mikroprogamy**). V časti **B** vysvetlíme vývoj (zjemňovanie) špecifikácií v používanom **všeobecnom modeli spracovania a komunikácie**, na ktorom je založený náš SSDL jazyk HSSL.

B. Vývoj špecifikácie pre RT úroveň návrhu

Vývoj špecifikácie systému vysvetlíme pomocou **príkladu** systému **VSYS** a príkladu **CPU** (pozri Kap.1 a 2). Pri vysvetľovaní pojmov a princípov vývoja (zjemňovania) behavioristickej špecifikácií sa budeme pridržiavať nášho modelu s agentmi a procesmi (Kap 1). Má výhodu (pri didaktickom prístupe) v tom, že ľahko možno predviesť prechod z hrubej špecifikácie príslušného modulu na **systémovej** úrovni na jeho behavioristickú špecifikáciu (pri HW implementácií modulu) na **RT** úrovni návrhu.

PRÍKLAD 1: Aplikačno-špecifický procesor VSYS



Procesor spracováva 16-bitové údaje (celé čísla v doplnkovom kóde), ktoré sa privádzajú cez dátový vstup **D**. Má dve neštrukturované signálové inštrukcie, ktoré sú inicializované z okolia prostredníctvom logických vstupov **RD** (čítanie a prenos poľa údajov do procesora) a **CO** (výpočet súčtu údajov)

RD=1 je aktívny signál, ktorý naštartuje vykonanie inštrukcie READIN, prenesenie 16 čísel d1,..,d16 cez **D** do vnútornej RAM pamäti, reprezentovanej operačnou stavovou premennou **M** typu pole 16 čísel a nastavenie signalizačného výstupu RR=1 (READIN Ready) informujúceho okolie o skončení operácie. Takéto správanie špecifikuje (vykonáva) agent ReadIn alebo proces PrReadIn

CO=1 je aktívny signál aktivujúci inštrukciu COMPUTE, výpočet aritmetického **súčtu** 16 čísel čítaných z vnútornej RAM (poľa **M**), vyslanie výsledku na výstup DO (zároveň operačnej stavovej premennej) a nastavenie signálu **CR**=1 (COMPUTE Ready) informujúceho o skončení operácie. Toto správanie špecifikuje (vykonáva) agent Compute alebo Proces PrCompute.

Premenné **REQ** (REQuest) a **ACK** (ACKnowledge) sa používajú počas vykonávania inštrukcie READIN pri semi-synchrónnom alebo asynchrónnom prenose čísel z okolia cez vstup **D** do procesora (s protokolom typu „hand shaking“- zo vzájomným potvrdením)

Najskôr uvedieme možnú **východiskovú** špecifikáciu systému VSYS na systémovej úrovni návrhu, v ktorej sú dva agenty ReadIn a Compute, ktoré realizujú inštrukcie READIN a COMPUTE sú zadané iba hrubo "**rámcovo**" (t.j. nešpecifikuje sa metóda prenosu údajov zvonku do pamäti M a ani spôsob výpočtu a nie je špecifikovaný ani spôsob definície diskrétneho času, teda ide iba o tzv. presnosť na cyklus). Agenty potom "**zjemníme**", t.j. zapracujeme do nich **podrobnejšie** správanie a takto tvorivo prejdeme ku **cieľovej špecifikácií**, ktorá je východiskovou behavioristickou špecifikáciou systému pre syntézu na RT úrovni. Pri iných subsystémoch implementovaných **HW spôsobom** sa postupuje podobne.

Hrubú (rámcovú) **špecifikáciu** správania modulu VSYS na systémovej úrovni sme zostavili takto:

System VSYS-0

DT údaj = [cele číslo < -215, + 215-1 >;
"+" v doplnkovom kóde];
pole U(0),...,U(15) údajov;
boolovska hodnota {0,1};

PORTY
vstup D pole D(1),...,D(16) údajov;
RD,CO booleovska hodnota;
výstup DO udaj;
RR,CR boolovska hodnota;

OPER STAV
M pole M(0),...,M(15) udajov;
RR,CR boolovská hodnota

agent ReadIn
SI RR = 0; // všetky globalne stavy z S={<M>,<RR>,<CR> <Cont>},
// pri ktorých je <RR>=0
TE es, ef = up(RR=1);
KM
(D=d_1,..,d_{16}; es; u)(u; ef; RR=1)
//vidno, ze cez D sa cita naraz cele pole D(1),....,D(16)
g M := d_1,...,d_{16};
RR := 1;
TR
del(up(RD=1), up(RR=1), T_{RD});
befo(up(RR=1), up(M=d_1,...,d_{16}), 0);
afto(up(RR=1), ef, 0);
// T_{RD} a v dalsom T_{CO} určujú výkonnosť (rýchlosť spracovania)

```
  agent  Compute
      SI    CR = 0;
      TE    es, ef= up(CR=1);
      KM
          (u; es; u)(u; ef; DO=do,CR=1);
      vs
          do = M(0) + ...+ M(15);
      g   CR := 1;
      TR
         del(up(CO=1), up(CR=1), T_CO);
         befo(up(CO=1), up(DO=do), 0);
         afto(up(CR=1), ef, 0);

START     ReadIn
             Compute   up(CO=1)
      restr   RD nand CO;
               afto(up(CO=1), up(RD=1), T_CO) ;
               afto(up(RD=1), up(CO=1), T_RD) ;
```

Alternatívne možno **VSYS** špecifikovať s inými položkami aj takto:

```
   proces   P1 = [(RD=1): ReadIn, (CO=1): Compute, ((RD=0)and(CO=0)):EA]^ω
START     P1    eon
       restr   RD nand CO
               afto(up(CO=1), up(RD=1), T_CO) ;
               afto(up(RD=1), up(CO=1), T_RD) ;
```

Druhým krokom po zostavení hrubej východiskovej formálnej špecifikácie bude vývoj agentov (procesov) vedený tak, že do nich zapracujeme isté **spresnenia** (zjemnenia do väčších detailov) spôsobov fungovania systému, opísaných menej podrobne vo východiskovej špecifikácií a to :

- spôsob **prenosu údajov** zvonku do vnútra systému a naopak zo systému do okolia (t.j. spôsob prenosu dát podľa určitých komunikačných protokolov).
- spôsob **definície cyklov** systému (synchrónny, asynchrónny,..) teda definície **diskrétneho času** časovacími udalosťami.
- spôsoby – **algoritmy** vnútorného spracovania informácie v module

V druhom kroku pri systéme **VSYS** podrobnejšie špecifikujeme

1. spôsob prenosu 16 celých čísel cez vstupný port **D** do vnútra VSYS, teda do poľa údajov operačnej premennej M zvolíme v agente **ReadIn** ako **sekvenčný** prenos 16 údajov cez vstup D (s údajovým typom "udaj"), s **protokolom** „vzájomného potvrdenia pripravenosti (systému a okolia)“ – „hand shaking“, pričom už na tejto úrovni návrhu anticipujeme, že pole **M** bude v budúcej RTL štruktúre implementované pamäťou typu R/W **RAM**
2. konkrétny spôsob definície cyklov pomocou periodických hodinových časovacích udalostí up(CLK=1, j), j=1,2,….., teda pôjde **o synchrónny** systém a protokol bude „**semi-synchrónny**“
3. **výpočet** sumy 16 celých čísel v agente **Compute** špecifikujeme ako **postupné sčítanie** čísel čítaných z poľa **M** (pamäti RAM) a prenesenie výsledku do výstupného portu DO.

system VSYS-1 // prvá zjemnená verzia špecifikácie

DT udaj = [cele číslo ε < -2^{15}, + $2^{15}-1$ >;
"+" v doplnkovom kóde];
pole U(0),...,U(15) udajv;
boolovska hodnota ε {0,1};
PORTY
vstup D udaj;
RD, CO, ACK, CLK boolovska hodnota;
vystup DO udaj;
RR, CR, REQ boolovska hodnota;
OPER STAV
M pole M(0),...,M(15) udaj;
RR, CR boolovska hodnota;
agent ReadIn;
SI RR = 0;
TE es, up(CLK=1) - default , ef;
KM
(RD=1; es ;REQ=0). [(ACK=1; ;REQ=0)*.(ACK=0; ;REQ=0)
[(ACK=0;;REQ=1)+.(ACK=1, D=d_j, ;;REQ=1).(ACK=1 ; ; REQ=0)]$^{1-16(D)}$.
. (u; ef; RR=1);
g
M := d_1,...,d_{16};
RR := 1;
TR
del(up(RD=1), up(RR=1), T_{RD});
befo(up(RR=1), up(M=d_1,...,d_{16}),0);
afto(up(RR=1), ef, 0);

agent Compute;
SI CR = 0;
TE es, up(CLK=1), ef;
KM
(CO=1; es; u) (u; up(CLK=1); DO=0)*
[(u; up(CLK=1); DO=d_i)]$^{1-16(DO)}$ (u; ef; CR=1, DO=d_V)
vs
d_1= <M(0)>,
FA i >= 2: d_i= d_{i-1} + <M(i-1)>;
d_V = d_{16}= <M(0)> + ...+ <M(15)>;
g
CR := 1;
TR
del(up(CO=1), up(CR=1), T_{CO});
befo(up(CO=1), up(DO=do), 0);
afto(up(CR=1), ef, 0);

START ReadIn (up(CLK=1) and RD);
Compute (up(CLK=1) and CO;

restr

RD nand CO;

afto(up(CO=1), up(RD=1) , T_{CO});

afto(up(RD=1), up(CO=1) , T_{RD});

Alebo **alternatívne**:

proces P1 = [(RD=1): ReadIn, (CO=1): Compute, ((RD=0)and(CO=0)):EA]*****

START

proces P1 (ez, Ø);

restr

RD nand CO; afto(up(CO=1), up(RD=1) , T_{CO});

afto(up(RD=1), up(CO=1) , T_{RD});

Je vidno, že v **tvorivom** procese zjemnenia sme voči VSYS-0 "rozšírili" obidva agenty v zmysle spresnenia definície základných cyklov (d-času), komunikácie s okolím a algoritmu výpočtu. Nezaviedli sme ďalší agent, ani nový proces ani novú operačnú stavovú premennú. Stavový priestor operačných stavov ostal teda nezmenený. Rovnomenne agenty s VSYS-0 a VSYS-1 považujeme za **ekvivalentné** (teda VSYS-1 je korektné voči VSYS-0) práve vtedy, ak v každom agente platí: Pre každé komunikačné slovo reprezentované formulou KM (presnejšie: každé vstupné slovo v každom kom slove z KM), **koncový operačný stav**, **výstupný vektor** vo výstupnom porte na **konci exekúcie** pri rovnomenných agentoch (resp. pre druhú alternatívu na konci exekúcie procesu P1) padne do jednej triedy zlučiteľných stavov v množine operačných stavov.

Práve opísaná "operácia" zjemnenia agentov má všeobecný význam a nazývame ju "**rozšírením**" agenta. Zjemní sa iba komunikácia pri prenose dát, t.j. pri ReadIn sme pridali jeden vstup (ACK) a jeden výstup (REQ), zjemnili sme údajový typ už priradený vo VSYS-0 pre vstupu D (zmenili sme ho v súlade s protokolom z typu „pole údajov“ na typ „údaj“) a vložili sme novú formulu pre opis KM v ReadIn opisujúcu nový kom protokol a upravili sme formulu pre KM v Compute v súlade so spresneným algoritmom výpočtu. Stavové premenné sme nepridali; výstupné premenné sme mohli deklarovať zároveň aj ako operačné stavové (napr. DO sme mohli deklarovať aj ako operačnú stavovú premennú), čo sme tu neurobili.

Ďalším krokom (tretím)zjemnenia špecifikácie VSYS bude zjemnenie agenta ReadIn pomocou druhého typu operácie zjemňovania, t.j. zjemnenie "**implementáciou**" agenta ReadIn **procesom** s jednoduchšími agentmi ako jeho prvkami. Vyžaduje to špecifikáciu **tvorivo rozšíriť** o **ďalších agentov** - prvkov tvoreného nového procesu. Ukážeme to na našom príklade.

system VSYS-2 // druhý krok zjemnenia VSYS
// voci predchadzajucej deklaracií pridame iba typ „index“
DT + index = [nezaporne cislo ε <0,15>; + je v module 16];

PORTY // rovnaké ako predtým

TR del(up(RD=1), up(RR=1), T_{RD})
del(up(CO=1), up(CR=1), T_{CO})
afto(up(CO=1), up(RD=1), T_{CO})
afto(up(DD=1), up(CO=1), T_{RD})

OPER STAV // voči predchádzajúcej špecifikácií sem pridáme I „index“
// DO a Req (zaroven vystupy)
I index; DO udaj; Req boolovska hodnota

proces PrReadIn = Reset. [$^{(ACK=1)}$[EA]. SetReq. [EA]$^{(ACK=1)}$. ResReq . WriteM.
.Incrl]$^{(I=0)}$. SetRR ;
proces PrCompute = Reset.[Add. Incrl]$^{(I=0)}$.SetCR ;

// [$^{(ACK=1)}$[EA] značí vlastne cyklus „čakania“ na ACK=0 ak na začiatku tohto cyklu
// ešte nie je ACK=0, resp. [EA]$^{(ACK=1)}$ je čakanie na ACK= 1 na konci cyklu.

agent Reset
TE es, ef → up(Clk=1); // → znaci, ze es a ef sa objavia v bode t(up(Clk))
KM (u; es; u).(u; ef; DO=0,RR=0,CR=0,REQ=0);
g
I:=0; DO:=0; RR:=0; CR:=0; REQ:=0;
TR FA e ε {up(I=0), up(DO=0), dw(RR=1), dw(CR=1), dw(REQ=1) }:
afno(e,ef,0) ;
agent Incrl
TE es, ef → up(Clk=1);
KM (u; es ;u)(u; ef ; u);
g I:=I+1;
TR afto(up(I=I+1),,ef,0);
agent Add
TE es, ef → up(Clk=1);
KM (u; es; u)(u; ef; DO=d);
vs d = DO + M(I);
g DO:=DO + M(I);
TR afto(up(DO=d), ef, 0);
agent SetCR
TE es, ef → up(Clk=1);
KM (u; es; u)(u; ef; CR=1);
g CR:=1;
TR afto(up(CR=1), ef,0);
agent WriteM
TE es, ef → up(Clk=1);
KM (D=d; es; u)(u; ef ; u)
g M(I):=d
TR afto(up(M(I)=d), ef,0)

```
agent   SetReq
     TE     es, ef → up(Clk=1);
      KM   (ACK=0; es; u)(ACK=0; ef; REQ=1);
      g         REQ:=1;
      TR     afto(up(REQ=1), ef, 0);
agent   ResReq
      TE     es, ef →up(Clk=1);
       KM: (ACK=1; es; u)(u; ef; REQ=0);
       g        REQ:=0;
      TR    afto(dw(REQ=1), ef,0);
agent   SetRR
      TE      es, ef → up(Clk=1);
      KM     (u; es; u)(u; ef; RR=1);
      g          RR:=1;
      TR     aftn(up(RR=1), ef, 0);

START    proces   PrReadIn (RD and up(CLK=1));
            proces   PrCompute     (CO and up(CLK=1));
        restr     RD nand CO; afto(up(CO=1), up(RD=1) , T_CO);
                                  afto(up(RD=1), up(CO=1) , T_RD);
       // procesy nemozu bezat subezne jeden s druhým; a ak sa nastartuje jeden, a
       // vykonava sa prave ten isty (rovnaký), tak tento sa taktiež predčasne skončí
```

V predchádzajúcej špecifikácií je možné použiť ako agenty procesov **generické mikrooprerácie**, napr. možno zaviesť "štandardnú" generickú microopreáciu

Meno mikrooperácie (v ; $StavPrem_1:=g_1$, $StavPrem_2:=g_2$,...,$StavPrem_k:=g_k$; h ;
n mikrocyklov)

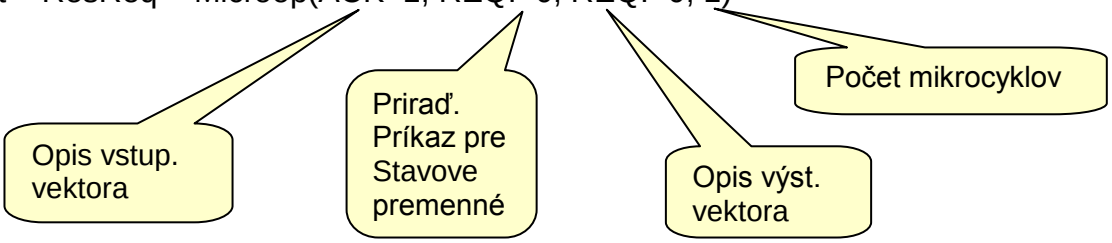
Ide o **opis mikrooperácie**, ktorý má nasledujúcu **interpretáciu**:

```
Agent    meno Microop
    SI;
    TE    es, ef → up(Clk=1);  // „→“ znaci, ze es a ef nastanú v d-bode t(up(Clk=1))
    KM   (v; es; u)(u ; ;u)^(n-1).....(u; ef; h);      // n je počet potrebných mikrocyklov
     g        StavPrem_1 := g_1(v,StavPrem_1,...,StatPrem_k)
                  :
              StavPrem_k := g_k(v,StavPrem_1,...,StavPrem_k)
    TR       afto(up(StavPrem_1=g_1), ef, 0)
                  :
              afto (up(StavPrem_k=g_k), ef, 0)   // ef sa objavi az po nastavení
                                                 // hodnoty operacnej stavovej premenej
```

Hore uvedený zápis mikrooperácie možno často prakticky použiť.

Napr. množinu agentov (mikrooperácií) v ostatnej špecifikácií môžme zapísať takto:

agent Reset (u;I:=0,DO:=0,RR:=0,CR:=0; DO=0,RR=0,CR=0, REQ=0; 1)
agent IncrI(u; I:=I+1; u ; 1)
agent Add (u; DO:=d=DO+M[I],M[I]:=M[I],I:=I; DO=d ; 1)
agent WriteM(D=d; M[I]:=d; u ; 1)
agent SetCR(u; CR:=1; CR:=1;1)
agent SetRR(u; RR:=1; RR:=1; 1)
agent SetReq(ACK=0; REQ:=1; REQ:=1; 1)
agent ResReq = Microop(ACK=1; REQ:=0; REQ:=0; 1)



Všetky agenty – mikrooperácie sú v tomto prípade jednocyklové.

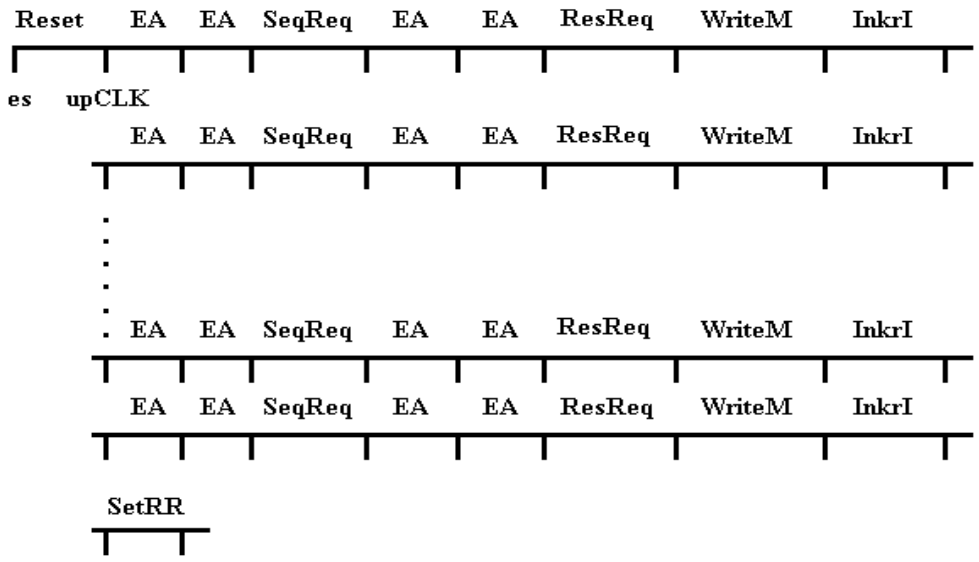
Špecifikáciu VSYS-2 sme zostavili zo špecifikácie VSYS-1, agenty ReadIn a Compute ktorej, predstavujú **špecifikáciu dvoch procesov**. Nazvali sme ich **PrReadIn** a **PrCompute** a zostavili sme ich z jednoduchých jedno-cyklových agentov - mikrooperácií. Týchto 8 agentov sme definovali na základe analýzy možného algoritmu – spriahnutia mikrooperácií pre implementáciu agentov ReadIn a Compute. Robili sme to tvorivo a vychádzali sme z určitej skúsenosti.

Ako vyzerá reťazenie agentov v našich procesoch ? (verifikácia vývoja – zjemnenia špecifikácie)

PrReadIn = Reset.[$^{(ACK=1)}$[EA].SetReq.[EA]$^{(ACK=1)}$
. ResReq.WriteM.Incrl]$^{(I=0)}$.SetRR ;

Tu sme pre jednoduchosť predpokladali dva cykly čakania v oboch slučkách
$^{(ACK=1)}$**[EA]** a **[EA]**$^{(ACK=1)}$.
Ďalej sa vyžaduje 16 opakovaní výrazu v [….], ktorý opisuje istú množinu slov (častí komunikačných slov) v daných hranatých zátvorkách.

Obraz o zreťazení agentov – mikrooperácií poskytuje nasledujúci **časový diagram** s vyznačenými bodmi d-času definovanými udalosťami up(CLK=1)



Uvedieme *konštrukcia komunikačnej množiny* KM procesu **PrReadIn**, ktorá sa vytvára pri jeho exekúcií. Pritom v akciách v **EA** (v prázdnom agente – mikrooperácií) môžme formálne uvádzať aj skutočné hodnoty vstupných portov, ktoré sa pasívne indikujú v príslušných d-bodoch. Pripomíname však, že v žiadnom prípade EA **nemôže meniť operačný stav** ani **hodnoty výstupných portov**. Postupne vyjadríme zreťazenie v procese **PrReadIn** sekvenčne spriahnutých agentov - mikrooperácií.

→ **Reset** KM **(RD=1;es;u)(u;ef; REQ=0, DO=0,RR=0,CR=0)**
EA KM **(ACK=1;es; REQ=0, DO=0, RR=0, CR=0)(u;ef;DO=0,…, CR=0)**
// zrejme vstup Ack je 1, lebo sa čaká na 0, Req=0 je daný Reset
Reset . EA* KM **(RD=1;es;u).(ACK=1; ;REQ=0, DO=0, RR=0, CR=0)***
.(ACK=0; ef ; REQ=0,DO=0,..,CR=0)

→ **SetReq** KM **(u;es;REQ=0)(u;ef;REQ=1)**

Reset . EA*.SetReq

KM

(RD=1;es; u)(ACK=1;;REQ=0,DO=0, RR,..,CR=0)*(ACK=0;;REQ=0, DO=0,..,CR=0)
.(ACK=0; ef; REQ=1, DO=0,..,CR=0)

→ **EA^+ = EA.EA***

KM **(ACK=0; es; REQ=1).(ACK=0;; REQ=1)*(ACK=1;ef ;REQ=1)**

Reset . EA*. SetReq . EA^+

KM

(RD=1;es;u).(ACK=1;; REQ=0,DO=0,..,CR=0)*.(ACK=0;;REQ=0,DO=0,..,CR=0)
(ACK=0; ;REQ=1, RR=0, CR=0).(ACK=0; ;REQ=1, RR=0, CR=0) *
.(ACK=1; ef ; REQ=1, RR=0, CR=0)

→ **ResReq** KM **(ACK=1;es; REQ=1)(ACK=1;ef;REQ=0)**

Reset . EA*. SetReq . EA^+. ResReq

KM

(RD=1;es;u).(ACK=1;; REQ=0, DO=0,..,CR=0)*.(ACK=0;;REQ=0, DO=0,..,CR=0)
(ACK=0; ;REQ=1, RR=0,CR=0).(ACK=0; ;REQ=1, RR=0, CR=0)*
(ACK=1 ; ; REQ=1, RR=0, CR=0). (ACK=1; ;REQ=0, RR=0,CR=0)

→ **WriteM** KM **(D=d;es;u)(u;ef;u)**

Reset . EA*. SetReq . EA^+. ResReq . WriteM

KM

(RD=1;es;u) . (ACK=1;; REQ=0, DO=0,..,CR=0)*· (ACK=0;;REQ=0, DO=0,..,CR=0)
(ACK=0;;REQ=1, DO=0,..,CR=0) . (ACK=0; ;REQ=1, RR=0, CR=0)* .
(ACK=1;;REQ=1, RR=0, CR=0) . (ACK=1,D= d_0; ;REQ=0, RR=0, CR=0).
.(u; ef; REQ=0, DO= d_0, RR=0,CR=0)

→ **Incrl** KM **(u;es;u)(u;ef;u)**

Reset . EA*. SetReq . EA^+. ResReq . WriteM. Incrl

KM

(RD=1;es;u) . (ACK=1;; REQ=0, DO=0,..,CR=0)*· (ACK=0;;REQ=0, DO=0,..,CR=0)
(ACK=0;;REQ=1, DO=0,..,CR=0) . (ACK=0; ;REQ=1, RR=0, CR=0)* .
(ACK=1;;REQ=1, RR=0, CR=0) . (ACK=1,D=d; ;REQ=0, RR=0, CR=0).
(u; ; REQ=0, DO=d_0, RR=0,CR=0) . (u;ef; DO=d_0, RR=1,CR=0)

Po tomto kroku sa komunikácia opakuje vo vonkajšom cykle ešte 15 krát, kým nie je index I=0. V kom slove sa pokračuje s **EA***

→ **EA*** KM **(ACK=1;es; REQ=0, DO=0,..,CR=0)*(u ;ef; REQ=0, DO=0,..,CR=0)**

Reset . EA*. SetReq . EA^+. ResReq . WriteM. Incrl EA*

KM

(RD=1;es;u) . (ACK=1;; REQ=0, DO=0,..,CR=0)*· (ACK=0;;REQ=0, DO=0,..,CR=0)
(ACK=0;;REQ=1, DO=0,..,CR=0) . (ACK=0; ;REQ=1, RR=0, CR=0)* .

(ACK=1;;REQ=1, RR=0, CR=0) . (ACK=1,D=d_{15}; ;REQ=0, RR=0, CR=0).
(u; ; REQ=0, DO=d_{15}, RR=0,CR=0) . (ACK=1;es; REQ=0, DO=0,..,CR=0)*
(u ;ef; REQ=0, DO=0,..,CR=0)
ďalej nasleduje KM pre
→ **SetReq**

Reset . EA*. SetReq . EA^+. ResReq . WriteM. Incrl EA*. SetReq

atď. na konci v každom reťazci výsledných kom slov bude agent :
→ SetRR KM (u;es;u)(u;ef;RR=1)

***Výsledná komunikačná formula pre proces* PrReadIn** *je*

Reset.[$^{(ACK=1)}$[EA].SetReq.[EA]$^{(ACK=1)}$.ResReq. WriteM. Incrl]$^{(I=0)}$.SetRR

Pri zavedení opakovacej slučky **[….]** $^{1\text{-}16\ (D)}$ a pri náhrade iterácie $\mathbf{EA^+}$ výrazom **EA.EA***, ktorý vložíme do **[EA]**$^{(ACK=1)}$ podprocesu, dostaneme **finálnu formulu** pre opis komunikácie. Zodpovedá danej kompozícií agentov v procese PrReadIn, teda komunikačnej množine agenta *PrReadIn*, ktorý špecifikuje daný proces (uvedieme ho neskôr).
KM

(RD=1;es;u) . [(ACK=1;; REQ=0, DO=0,..,CR=0)*· (ACK=0;;REQ=0, DO=0,..,CR=0)
(ACK=0; ;REQ=1, RR=0, CR=0)$^+$. (ACK=1;;REQ=1, RR=0, CR=0) .
(ACK=1,D=d_j; ;REQ=0, RR=0, CR=0). (u; ; REQ=0, DO=d_j, RR=0,CR=0)] $^{1\text{-}16(D)}$.
(u;ef; DO=dv, RR=1,CR=0)

Komunikačná výraz pre **KM** agenta ReadIN (zostaveným vo VSYS -1, ktorý sme nahradili procesom **PrReadIn** v VSYS-2) je:
KM

(RD=1; es ; REQ=0). [(ACK=1; ; REQ=0)*. (ACK=0; ;REQ=0).
(ACK=0; ; REQ=1)$^+$.(ACK=1 D=d_j; ; REQ=1) . (ACK=1, D=d_j; ; REQ=0)]$^{1\text{-}16(D)}$.
.(u; ef; RR=1)

Je vidno, že výrazy **nie sú rovnaké**. To však nemusí byť zlé**!!** Pri úprave posledného výrazu "KM" agenta **ReadIn** Ďalej môžme spokojne do akcií doplniť agentom **Reset** nastavené – **vynulované** - hodnoty operačných stavových premenných **RR**, **CR**, ktoré sú zároveň deklarované ako **výstupné porty**. Takisto možno dourčiť aj výstupný port a vložiť **DO=0** do tých akcií, kde **zotrvá** agentom **Reset** nastavená nulová hodnota operačnej a zároveň výstupnej premennej **DO**. Ako sme už videli, do KM v **EA** možno doplniť špecifikované hodnoty vstupných premenných, avšak v EA **nesmie dochádzať k zmene** hodnôt výstupov. Teda výraz KM pre agent ReadIn môžme upraviť takto :
KM

(RD=1;es;u). [(ACK=1;; REQ=0, DO=0,..,CR=0)*· (ACK=0;;REQ= 0, DO=0,..,CR))
(ACK=0; ;REQ=1,RR=0,CR=0)$^+$(ACK=1; ; REQ=1, RR=0, CR=0) .
(ACK=1, D = d_j ; ; REQ=0, DO= d_j , RR=0, CR=0)] $^{1\text{-}16(D)}$.
(u; ef; RR=1, REQ=0,DO=dv,CR=0)

Túto komunikačnú formulu môžme teraz **porovnať** s nasledujúcou formulou pre KM agenta ***PrReadIn***, odvodeným priamo z implementujúceho procesu **PrReadIn**.

Pri porovnaní výrazov (po spomenutej úprave) je vidieť, že sú **ekvivalentné** (pri dourčení vstupov a výstupov, ktoré **zachováva zlučiteľnosť** sebe zodpovedajúcich akcií). V KM agenta ***PrReadIn*** je pridaná posledná akcia**:**
(u ;;REQ=0, DO= d_j, RR=0, CR=0) pred zátvorkou "]". Táto akcia z hľadiska požadovanej funkcie neprekáža. Predĺži sa iba exekúcia agenta ***PrReadIn*** (implementovaného procesom PrReadIn) o **jeden** hodinový cyklus voči požiadavke vo VSYS-1. Je to preto, že vo východiskovom agente ReadIn vo VSYS-1 sme **mlčky** predpokladali **súbežné** vykonávanie agentov **WriteM a IncrI**, t.j. paralelnú kompozíciu (exekúciu) (**WriteM || IncrI**), ktorú sme v procese PrReadIn nepoužili.

Ako je to z koncovým stavom v procese* PrReadIn *?

Z procesu priamo vyplýva, že na jeho konci bude vďaka opakovanom vykonávaní (16 razy) mikrooperácie WriteM zo svojim načítaným údajom „d_J"(j=1..,16) a za exekúciou mikrooperácie **SetRR** na konci agenta ReadIn sa dosiahne nasledujúci operačný koncový stav, ktorý je **z hľadiska korektnosti najdôležitejší**

M := d_1 + ... + d_{16} =dv

RR := 1

Je vidno že zjemnená špecifikácia má rovnakú **vlastnosť**, ako východisková na systémovej úrovni, pokiaľ ide o natavenie operačného stavu na konci **Readin**. Po exekúcií agenta ReadIn a tiež aj zjemňujúceho procesu **PrReadIn**, ktorý má **ReadIn** implementovať (pomocou zostavenej množiny jednoduchších agentov) dávajú **rovnaký operačný stav**.

Z predchádzajúcej analýzy (verifikácie, ktorú sa dá formalizovať) môžme zostaviť agent ***PrReadIn***, ktorý zodpovedá procesu **PrReadIn**, a o ktorom hovoríme, že **je špecifikáciou** procesu **PrReadIn**, alebo naopak, že proces **PrReadIn** je **implementáciou** pôvodného východiskového agenta ReadIn (z predchádzajúcej - vyššej úrovni vývoja).

agent *PrReadIn*
SI:; //začína s akéhokoľvek stavu globálnej množiny stavov systému VSYS
TE es, up(CLK=1), ef;
KM
(RD=1;es;u) . [(ACK=1;; REQ=0, DO=0,..,CR=0)*.
(ACK=0;;REQ=0,DO=0,..,CR=0).(ACK=0; ;REQ=1, RR=0, CR=0)$^+$.
(ACK=1;;REQ=1, RR=0, CR=0) . (ACK=1,D=d_j; ;REQ=0, RR=0, CR=0).
(u; ; REQ=0, DO=d_j, RR=0,CR=0)] $^{1\text{-}16(D)}$. (u;ef; DO=dv, RR=1,CR=0);
g M := d_1,...,d_{16};
RR := 1;
TR del(up(RD=1), up(RR=1), T_{RD}); // opísali sme v agente ReadIn
befo(up(RR=1), up(M=d_1,...,d_{16}),0);
aftno(up(RR=1),ef,0);

Podobne možno postupovať a verifikovať východiskovú špecifikáciu danú agentom Compute a jeho implementujúcim procesom **PrCompute**. Pri verifikácií ide vlastne o porovnanie dvoch agentov: východiskového agenta A s agentom *PrA*, ktorý odvodíme z finálneho implementujúceho procesu PrA (pozri nasledujúci obr.). V

prípade východiskového agenta Compute, tento agent porovnáme s agentom *PrCompute*, ktorý odvodíme z procesu PrCompute podobne ako pri PrReadIn.
Tu na jednoduchom praktickom prípade vidno, ako možno pristúpiť k **formálnej verifikácií**, teda verifikácií striktným **dôkazom**, že tvorivo vyvinutá (zjemnená) špecifikácia (ktorá opisuje alebo urobí exekúciu určitého čiastkové správania, je v podstate taká, ako východisková špecifikácia (povedzme) na systémovej úrovni. Predvedenú procedúru transformácie konečného procesu na agent, ktorý ho špecifikuje možno **algoritmicky riešiť a implementovať programom**.

Nakoniec uvedieme ešte výsledok transformácie procesu **PrCompute**, ktorý sme tvorivo zostavili vo vývoji (pri zjemňovaní) behavioristickej špecifikácie, na príslušný agent *PrCompute*, ktorý je jeho špecifikáciou.

PrCompute = Reset.[Add.IncrI]$^{(I=0)}$.SetCR ;

Odvodením z tohto procesu dostaneme tento komunikačný výraz procesu, vlastne agenta *PrCompute*
KM
(CO=1;es;u)(u;;DO=0,RR=0,CR=0,REQ=0) [(u;;DO=d_i)(u;;u)]$^{1\text{-}16(DO)}$.
(u;ef;CR=1,DO=d_V)
Vo východiskovom agente Compute vo VSYS-1 sme mali:
Compute
KM
(CO=1;es;u)(u;;DO=0)* [(u;;DO=d_i)]$^{1\text{-}16(DO)}$ (u;ef;CR=1,DO=d_V)

V výraze KM pre *PrCompute* je medzi zátvorkami **[**..**]** navyše akcia **(u; ;u)**, ďalej vo výraze sú dourčené navyše niektoré výstupy a akcia **(u;;DO=0)** sa vyskytuje práve jeden raz, čo z hľadiska korektnosti **všetko nevadí**.

Koncový stav agenta *PrCompute* po 16 razy opakovanej mikrooperácií **Add** (pri ktorom sa vždy obnoví obsah DO podľa priraďovacieho príkazu DO := DO + M(I))
a po vykonaní **SetCR** na konci procesu bude:

DO := d_V pričom **d_V = d_{16} = M(0) + ... + M(15)**
CR := 1

POZNÁMKA (dôležitá): Pri vývoji (zjemňovaní) zložitých behavioristických špecifikáciách sa požaduje **verifikovať každý krok** zjemnenia (t.j. každá novo-zostavená špecifikácia), pokiaľ sa transformácia na zjemnenú špecifikácia rieši tvorivo, nealgoritmickou transformáciou,. Ak sa takýto spôsob príjme a v návrhovom systéme zavedie, netreba už verifikovať cieľovú špecifikáciu (napr. VSYS-2) voči východiskovej (napr. VSYS-0).

V našom príklade je cieľová špecifikácia VSYS-2 korektná i voči VSYS-0. Tu máme na mysli **ekvivalenciu koncových** stavov pri daných začiatočných stavoch, ktorú možno formálne dokázať

Vlastnosti komunikačných FSM **KSM**, ktoré možno algoritmickou transformáciou zostaviť z výrazov (formúl) komunikačných množín agentov a procesov (resp. agentov, ktoré procesy špecifikujú) – pozri Kap. 2, možno samotné podrobiť

testovaniu vlastností („model checking“ a tým **formálne**, dôkazom potvrdiť, že dva porovnávané komunikačné výrazy majú rovnaké vlastnosti, ktoré návrhár požaduje a sú pre dosiahnutie východiskovej špecifikácie relevantné.
Teraz, v osobitnej sub-4asti C uvedieme druhý príklad, ktorý je venovaný jednoduchému aktívnemu procesoru CPU, ktorý sme už uviedli predtým

C. Príklad vývoja špecifikácie aktívneho procesora „CPU“

Uvedieme zjemnenie špecifikácie procesora s menom "CPU". Použijeme pritom už uvedenú východiskovú špecifikáciu CPU-0 (z kap.2, PRÍKLAD 4). Agentov východiskovej špecifikácie CPU-0 implementujeme procesmi v CPU-2. Prejdeme pritom hneď na verziu s procesmi s mikrooperáciami, ktoré predstavujú už špecifikáciu správania CPU na RT úrovni, teda na úrovni RT (mikroarchitektúry).

```
Systém    CPU-0    // východisková špecifikácia
DT     data.      [cele cislo   <-2^31,+2^31- 1>; + ,-  v
                   dvojkovom doplnkovom kode];
       adresa   nezaporné cislo    < 0, 2^30- 1 >; + v module 2^30;
       opkod    symbol ϵ  {AD,BRN,LD,ST};
       instrukcia = záznam
                     opkod:  opkod,
                     adresa: adresa;
       signal = boolovska hodnota {0,1}
PORTY
  vstup            D    data, instrukcia,
                   Res  signal
  výstup           D    data,
                   A    adresa;
OPER STAV          PC   adresa;
                   AC    data;

 proces    Comp = ResOp.[GlobOp]^ω
  agent   GlobOp
   TE   es, up(A=PC), up(D=d1),up(A=d1.adresa), up(D=d2),
        up(D=AC), ef;
   KM     (u; es; u)(u ; ; A=PC)(D=d1 ; ; u).
          [d1.opkod  ε {ADD,LD} : (u; ; A=d1.adresa)(D=d2; ; u)(u; ef; u),
           # d1.opkod = BRN:       (u; ef; u),
           # d1.opkod = ST:        (u ; ; A=d1.adresa)(u ; ; D=AC)(u; ef; u) ];
   g    pre prípad  d1.opkód
                    AD        AC := AC + d2,  PC := PC+ 1;
                    LD        AC := d2, PC := PC+1;
                    ST        PC := PC+ 1;
                              // údaj <AC>:sa má zapísať do
                              // vonkajšej RAM na adresu
                              // d1.adresa
                     BRN      PC := ak <AC> < 0 tak  d1.adresa  inak  PC+1;
 agent   ResOp
   TE   es, ef;
    KM   (u;es;u)(u;ef;u);
    g       PC : =  0;
```

START Comp (ez or up(Res=1);

Ako **alternatívu** uvedieme špecifikáciu variantu CPU s **prerušením** (Int -interupt). Uvedieme ako sa môže doplniť východiskovú špecifikáciu CPU-0.

Do špecifikácie CPU pridáme nasledujúce entity

1. Operačnú stavovú premennú STACK, ktorá má známy komplexný dátový typ „stack“ s operáciami push((d,d d,..,d), STACK) a pop (STACK), kde (d,d d,..,d) je vektor konkrétnych dát.
2. Vstup CPU „IntVektor“ // určuje adresu skoku na obslužný program prerušenia
3. agent IntEx // agent vykonávajúci výnimočné (excepčné) správanie prerušenia
   IS (<PC>, <AC>)
   TE es, ef
   KM (u, es, u).(u ; ef; u)
   g pusch((<PC>,<AC>), STACK)
   PC := IntVector
4. zostavíme nový proces CPU CompEx = IntEx . $[GlobOp]^{\omega}$
5. START zmeníme takto
   START Comp (ez or up(Res=1);
   CompEx Int and (“f “ of GlobOp)

Treba si všimnúť, že po prijatí signálu prerušenia, ktorý sa **testuje** pri **koncovej akcií** „f“ agenta **GlobOp**, ak je f a Int=1 (pozri štartovací výraz), tak sa naštartuje proces **CompEx** a pretože nie je v reštrikcií procesný predikát **c**(Comp,Compex), bežiaci proces **Comp** sa skončí. Skončí sa však na konci **GlobOp** a teda riadne sa zakončí exekúcia inštrukcie. Po vykonaní agenta **IntEx** potom opäť pokračuje nekonečné opakované vykonávanie procesu $\mathbf{[GlobOp]^{\omega}}$, ktorý začína tentoraz od adresy **<IntVektor>** dodanej z okolia CPU cez pridaný vstup CPU **IntVektor.** Táto adresa je asociovaná s daným prerušovacím signálom.

Operačný stav na konci **GlobOP**, ktorý tvorí tzv. **kontext** (<PC>=a, <AC> d) prerušeného programu sa uchová pomocou **IntEx** v pamäti **STACK** a do **PC** sa vloží nová adresa **<IntVector>**. Ak príde ďalšie prerušenie, tak sa opísaná situácia opakuje a GlobOP skočí na inú adresu **<IntVektor>** asociovanú s daným prerušovacím signálom vo vstupe Int. Hĺbka vnorenia prerušovaných procesov záleží od **hĺbky** pamäti **STACK**. Adresa návratu do prerušeného sa získa operáciou pop(STACK) t.j. pop(STACK).<PC> = a. V univerzálnych procesorov sa to prakticky rieši zavedením osobitnej inštrukcie, realizujúcej operáciu pop, pri ktorej sa vyberie celý kontext z pamäti STACK a v danom prípade CPU vložia sa pôvodné obsahy registrov „a“ a „d“ do PC a AC. Návrat predchádzajúceho kontextu zabezpečuje v CPU spravidla interpretovaný obslužný program prerušenia.

ÚLOHA na riešenie: Vypracujte špecifikáciu procesora CPU^{+} na úrovni CPU-0, ktorý má pôvodný súbor inštrukcií (ADD, BRN, LD, ST) a reaguje na jednoduché prerušenie s hĺbkou 1. Zaveďte prídavné inštrukcie PU, PO, a NPP, ktoré realizujú operácie pusch a pop v externej pamäti (kde sa uchová jednoduchý kontext), resp. návrat z obslužného programu.

Teraz však pokračujeme v o vývoji (zjemnení) pôvodne opísanej východiskovej špecifikácie CPU-0 (bez systému prerušenia) a v ďalšom kroku zostavíme špecifikáciu CPU-1.

```
Systém  CPU-1              // prvá zjemnená špecifikácia
DT                         // detto ako pri CPU-0
PORTY  vstup          D    udaj,
                      Wait signál;
          vystup      D    udaj,
                      A    adresa,
                      Req  signál;
OPER STAV             PC   adresa,
                      IR   inštrukcia,
                      AC   údaj,
                      Contr r_stav;
proces          InstrCy = RESoper.[InstrRead.( (IR.opkod=BRN): BRNop,
                                               (IR.opkod=ADD): ADDop,
                                               (IR.opkod=LD):  LDop,
                                               (IR.opkod=ST):  STop )]*;
 agent InstrRead
   SI    Contr = IF;
   TE    es, up(CLK=1), ef;
    KM    (u; es; u).(Wait=0; ;u).(Wait=0; ; u)*.(Wait=1 ; ; A=PC, RdWr=1, Req=1)+ .
          .(Wait=0,  D=d ; ; Req=1).(Wait=0; ; Req=0)*.(Wait=1;ef; Req=0).
   g:     IR  := d;    //IR obsahuje inštrukciu
          PC  := PC+1;
         Contr := OD; // OD „Operation Decode“
 agent   BRNoper
   SI   Contr = OD;
   TE: es, up(CLK=1), ef;
   KM  (u; es; u)(u; ef; u);
   g:    Ak AC < 0 tak PC := IR.adresa inak
                  PC := PC + 1;
                  Contr := IF;
 agent    ADDoper
   SI  Contr = OD;
   TE  es,up(CLK=1),ef;
   KM:
       (Wait=1`;es; u) (Wait=1; ;A=IR.adresa,Req=1,Rd/Wr=1)+.
      .(Wait=0, D=d ; ; Req=1).(Wait=0; ; Req=0)*.(Wait=1 ; ef ; Req=0);
   g:    PC  := PC + 1;
         AC  := AC + d;
         Contr := IF;
agent     Ldoper
   SI  Contr = OD;
   TE  es, up(CLK=1), ef;
   KM   (Wait=1; es; u).(Wait=1 ; ; A=IR.adresa, Req=1, Rd/Wr=1)+
        .(Wait=0, D=d ; ; Req=1). (Wait=0; ; Req=0)*.(Wait=1 ; ef ; Req=0)
   g      PC  := PC + 1;
          AC  := d;
          Contr := IF;
```

```
agent     Stoper
   SI  Contr = OD;
   TE  es, up(CLK=1), ef;
    KM  (Wait=1; es; u).(Wait=1 ; ;A=IR.adresa, D=AC, Rd/Wr=0, Req=1)+
        .(Wait=0 ; ; D= d, R/W=0, Req=1). (Wait=0; ; D=d, R/W=1, Req=0)*.
        .(Wait=1 ; ef ; D=d; Req=0)     //udaj d sa zapise do pamti pri up(R/W=1)
   g       PC  := PC + 1;
           Contr := IF;
 agent    ResOper
   TE  es, ef;
    KM   (u; es; u)(u; ef; u);
    g       PC  := 0;
            Contr := IF;
START
   InstrCy   (ez or (up(Clk=1)) and Res)
```

Prejdeme na ďalšie zjemnenie behavioristickej špecifikácie CPU a to implementáciou jednotlivých agentov InstrRead, …., SToper procesmi. V tomto prípade v CPU-2 prejdeme už na procesy s mikrooperáciami.

```
System  CPU-2

DT      // detto ako v CPU-1
PORTY
  vstup      D     údaj,
             Wait  signál;
             Clk   signal
  výstup     D     udaj,
             A     adresa,
             Req   signál;

OPER STAV   PC    adresa,
            IR    inštrukcia,
            AC    údaj,
            MAR   adresa    // „Memory Address Register“
            MBR   údaj      // „Memory Buffer Register“

proces   PrInstRead =  (LdMAR || IncrPC).[EA](Wait=1)[SetMRd](Wait=0).
                          [LdIR](Wait=1);                  // "||" značí súbežnosť
proces      PrST        (MovIR-MAR || LdMBR).[SetMWr](Wait=0);
proces      PrBRN       (AC<0): EA. MovIR-PC, (AC >=0):EA;
proces      PrLD        MovIR-MAR.[SetMRd](Wait=0).LdAC;
proces      PrADD       MoveIR-MAR.[SetMRd](Wait=0).AddAC;
proces      PrGlobal    Reset.[ PrInstRead.((IR.opkód=BRN): PrBRN,
                                            (IR.opkód=ADD): PrADD,
                                            (IR.opkód=LD):  PrLD,
                                            (IR.opkód=ST):  PrST)  ]ω;
```

```
agent  LdMAR
       TE  es,ef;
        KM: (u;es;u)(u;ef;u);
        g     MAR := PC
agent  IncrPC
       TE  es, ef;
        KM  (u;es;u)(u; ef;u);
        g:     PC := PC + 1
agent  SetMRd
       TE  es, ef;
       KM  (Wait=0,D=d; es; u)(Wait=0, D=d; ef; A=MAR, Req=1, Rd/Wr=1);
       g     MBR := d
 agent  LdIR
       TE   es, ef;
       KM  (Wait=0; es;u)(u; ef; Req=0);
       g      IR := MBR
agent  LdAC
      TE  es, ef;
       KM  (u; es; u)(u; ef; u);
       g       AC := MBR
agent  LdMBR
     TE   es, ef;
      KM   (u; es; u)(u; ef; u);
      g:       MBR := AC
agent  SetMWr
     TE   es, ef;
     KM   (Wait=1; es; u)(u; ef; A=MAR, D=MBR, Rd/Wr=0, Req=1);
     g;
 agent  MovIR-PC
      TE  es,ef;
      KM   (u; es; u)(u; ef; u);
      g:       PC := IR.adresa
 agent  MovIR-MAR
     TE   es, ef;
     KM.. (u; es; u)(u; ef; u);
     g       MAR := IR.adresa
agent  AddAC
    TE  es,ef;
     KM  (u; es; u)(u; ef;u);
     g       AC := AC + MBR
agent  Reset
    TE  es, ef;
    KM   (u; es; u)(u; ef; u);
    g        PC := 0;

START
            PrGlobal   (ez or (up(Clk=1) and Res);
            Reset       (up(Clk=1) and Res
```

Verifikácia CPU-1 oproti CPU-0

Potrebujeme **zostaviť** agentov, ktoré zodpovedajú jednotlivým procesom (špecifikujú jednotlivé procesy) v CPU-1 a **porovnať** s agentmi v CPU-0. Ukážeme to iba na procesoch (1) **PrInstRead** a (2) **PrBRN**, teda zostavíme agentov ***PrInstRead*** resp. ***PrBRN***.

(1) PrInstRead = (LdMAR || IncrPC).[EA]$^{(Wait=1)}$ [SetMRd]$^{(Wait=0)}$.
.[LdIR]$^{(Wait=1)}$;

Mikrooperácie LdMAR a IncrPC sa **realizujú súbežne**. V danom prípade sa exekujú v tom istom mikrocykle (mikrotakte) Pretože v akciách ich KM sú vstupné a tiež výstupné vektory nešpecifikované, možno KM pre ich paralelnú kompozíciu vyjadriť a opísať bez operátora „||“ ako spoločný výraz. Priraďovacie príkazy pre jednotlivé stavové premenné v súbežných mikrooperáciách sa potom združia do jedného súboru (pozri nižšie).

LdMAR KM (u;es;u)(u;ef;u); g ==> MAR:= PC;
IncrPC KM (u;es;u)(u;ef;u); g ==> PC:= PC+1;
IncrPC || LdMAR KM (u;es;u)(u;ef;u); g ==> MAR:= PC,
PC:= PC+1;

(1) Zostavenie KM generovanej procesom PrInstRead a zodpovedajúceho agenta

PrInnstRead = (LdMAR || IncrPC).[EA]$^{(Wait=1)}$ [SetMRd]$^{(Wait=0)}$.
[LdIR]$^{(Wait=1)}$;

→Vezmeme prvý agent, teda kompozíciu (LdMAR || IncrPC) a zreťazíme s EA^{+}:

(IncrPC || LdMAR) KM (u;es;u)(u;ef;u)
EA KM (Wait=0;es;u) (u;ef;u)
E^{+}= EA.EA* KM (Wait=0;es;u).(u; ;u)*.(Wait=1;ef;u)
//EA nemení oper stav ani výstup,ide o čakanie na vstup Wait=1

Zreťazením dostaneme čiastočný výraz :
(u; es; u).(Wait=0;;u).(Wait=0, ,u)*.(Wait=1;ef;u)

→Zreťazíme ďalší sufix SetMRd^{+}:

SetMrd KM (Wait=1; es u).(Wait=1; ef ;A=MAR,Req=1,Rd/Wr=1)
SetMRd^{+} = SetMRd . SetMRd*
KM (Wait=1;es;u).(Wait=1 ; ; A=MAR, Req=1, Rd/Wr=1)*.
.(Wait=0, D=d$_j$; ef; A=MAR, Req=1, Rd/Wr=1)

Zreťazením s výrazom v predchádzjúcom kroku „→“ dostaneme čiastočný výraz :
KM
(u; es; u)(Wait=0;;u)(Wait=0, ,u)*
(Wait=1; ; A=MAR,Req=1,Rd/Wr=1)$^{+}$
(Wait=0,D=d$_j$; ef;A=MAR, Req=1, Rd/Wr=1)

→ Pridáme zreťazenie so sufixom LdIR^{+}

LdIR KM (Wait=0; es ;u).(u; ef; Req=0)
LdIR^{+}= LdIR.LdIR* KM (Wait=0; es ;u).(Wait=0; ;Req=0)*.
.(Wait=1, ef, Req=0)

Zreťazením ostaneme kompletný výraz pre KM, ktorú generuje PrInstRead :
KM
(u; es; u).(Wait=0;;u).(Wait=0;;u)*.(Wait=1;; A=MAR,Req=1,Rd/Wr=1)$^{+}$.
.(Wait=0, D=d; ; A=MAR, Req=1, Rd/Wr=1).(Wait=0, ,Req=0) *
.(Wait=1; ef ; Req=0)
Minimálne komunikácia trvá 4 mikrocykly (hodinové cykly):

(LdMAR || IncrC). EA . SetMRd . LdIR
1 2 3 4
(minimálne je 5 bodov d-času pre 4 mikrocykly.)

Výsledný komunikačný výraz agenta *PrInstRead* je teda :

KM
(u; es; u).(Wait=0; ;u).(Wait=0 ; ;u)*.(Wait=1; ; A=MAR, Req=1, Rd/Wr=1)$^{+}$.
.(Wait=0, D=d ; ; A=MAR, Req=1, Rd/Wr=1).(Wait=0, ef, Req=0) *.
.(Wait =1;ef; Req=0)

V tejto formule môžme subvýrazy typu (Wait=0; ;u).(Wait=0 ; ;u)*nahradiť výrazom (Wait=0; ;u)$^{+}$. Dostaneme :

(u; es; u).(Wait=0; ;u)$^{+}$. (Wait=1 ; ; A=MAR, Req=1,Rd/Wr=1) $^{+}$.
.(Wait=0, D=d ; ; A=MAR, Req=1, Rd/Wr=1).(Wait=0 ; ; Req=0)*
.(Wait =1 ; ef; Req=0)

Pre **porovnanie** uvádzame KM **východiskového** agenta **InstRead** z CPU-1 :
KM
((u; es; u).(Wait=0; ;u).(Wait=0; ; u)*.(Wait=1 ; ; A=PC, RdWr=1, Req=1)$^{+}$.
.(Wait=0, D=d ; ; Req=1).(Wait=0; ; Req=0)*.(Wait=1;ef; Req=0).

Výraz KM z východiskového agenta InstVyb v CPU-0 je fakticky ekvivalentný s výrazom KM agenta ***PrInstVyb*** až na to, že v 3. akcií v KM v agente ***PrInstVyb*** sú (prípustne) dourčené niektoré predtým nastavené hodnoty.

Ďalej je zrejmé, že po vykonaní procesu PrInstVyb (agenta *PrInstRead*)sa dostane CPU do vyžadovaného
koncového stavu

IR := d // d je inštrukcia
PC := PC + 1
MAR := PC
MDR := d

Teraz môžme zostaviť agent ***Pr InstRead***
agent *PrInstRead*
SI Contr=IF0
TE es,up(Clk=1),ef;
KM (u; es; u).(Wait=0; ;u)$^{+}$. (Wait=1 ; ; A=MAR, Req=1,Rd/Wr=1) $^{+}$.
.(Wait=0, D=d ; ; A=MAR, Req=1, Rd/Wr=1).(Wait=0 ; ; Req=0)*
.(Wait =1 ; ef; Req=0)

g IR := d;
PC := PC + 1;
MAR := PC;
MDR := d;
Contr:=OD;

(2) Zostavenie KM generovanej procesom* PrBRN *a zodpovedajúceho agenta

PrBRN = (AC<0): MovIR-PC, (AC >=0): EA;

→ EA KM (u;es;u)(u;ef;u)
MovIR-PC KM (u; es;u)(u; ef; u)
(AC<0): MovIR-PC, (AC >=0):EA
KM [AC >=0: (u; es;u) # AC < 0: (u; es ; u)] (u; ef; u);

Agent *PrBRN* vyzerá takto:

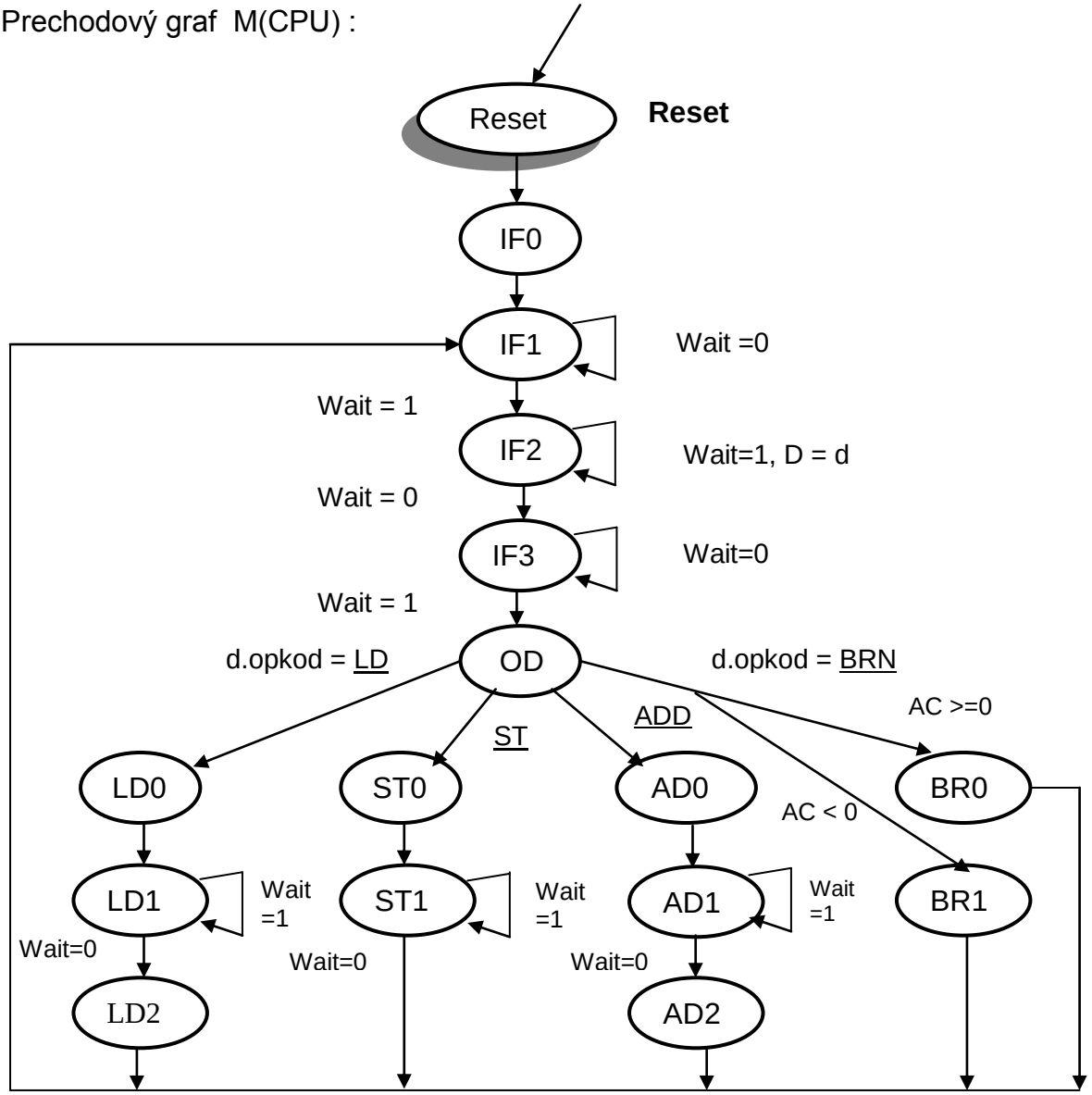
agent *PrBRN*
SI (Contr=OD, <AC> = d)
TE es,ef; → up(Clk =1)
KM [AC >=0: (u; es;u) # AC < 0: (u; es ; u)] (u; ef; u);
g Ak AC < 0 tak PC := IR.adresa inak
PC := PC + 1;
Contr := IF0;

Z globálneho procesu PrGlobal
PrGlobal = Reset.[PrInstVyb.((IR.opkód=BRN): PrBRN,
(IR.opkód=ADD): PrADD,
(IR.opkód=LD): PrLD,
(IR.opkód=ST): PrST)]$^{\omega}$;
a z jednotlivých agentov *PrInstRead*, *PrBRM*, *PrADD*, *PrLD* a *PrST*, ktoré zodpovedajú procesom PrInstVyb, PrBRM, PrADD, PrLD a PrST možno odvodiť **konečný stavový stroj** (FSM), ktorý vyjadruje riadenia a teda aj funkciu riadiacej časti CPU (ako uvidíme v ďalšom). Tu uvedieme Moorov FSM
RS = (DV,R,MO,p,v),
kde DV sú vstupné vektory pre riadenie, R sú stavy riadenia, MO sú mikrooperácie (jedno-cyklové agenty) v procesoch a p, v sú: prechodová a výstupná funkcia.
p : R x DV → R, v: R → MO,
teda v (q) = M je mikrooperácia, ktorá sa má vykonať v stave q.

Prechodová tabuľka FSM M(CPU) (v riadkovej forme) zo začiatočného stavu RES, do ktorého sa dostane vždy po vykonaní agenta, (mikrooperácie) Reset (naštartovanej z vnútra procesu PrGlobal, pri naštartovaní procesu z ST) z akéhokoľvek stavu (čo nie je explicitne zapracované v nasledujúcej tabuľke) je:

Stav q	Hodnoty vstupov vstupný.vektor „v“	Nasl. stav p(q,v)	Mikrooperácie v(q)	podprocesy v PrGlobal
RES	U	IF0	Reset	
IF0	U	IF1	LdMAR || IncrPC	
IF1	Wait=0	IF1	EA	
IF1	WAIT=1	IF2		
IF2	Wait=1	IF2	SetMRd	
IF2	Wait=0,D=d	IF3		PrIstVyb
IF3	Wait=0	IF3	LdIR	
IF3	Wait=1	OD		
OD	d.opkod=ST	ST0	EA	
OD	d.opkod=ADD	AD0		
OD	d.opkod=BRN, c=0	BR0		
OD	d.opkod=BRN, c=1	BR1		
OD	d.opkod=LD	LD0		
LD0	Wait=1	LD1	MovIR-MAR	
LD1	Wait=1	LD1	SetMRd	PrLD
LD1	Wait=0,D=d	LD2		
LD2	u	IF0	LdAC	
ST0	u	ST1	MovIR-MAR || LdMBR	PrST
ST1	Wait=1	ST1	SetMWr	
ST1	Wait=0	IF0		
AD0	u	AD1	MovIR-MAR	
AD1	Wait=1	AD1	SetMRd	PrADD
AD1	Wait=0,D=d	AD2		
AD2	u	IF0	AddAC	
BR0	u	IF0	EA	PrBRN
BR1	u	IF0	MovIR-PC	

Prechodový graf M(CPU) :



Výstupné symboly uvedeného Moorovho FSM (pozri prechodovú tabuľku - v grafe sme ich kvôli lepšej prehľadnosti vynechali) sú mená agentov - **mikrooperácií** pridelené na vykonanie v OČ pri jednotlivých stavoch v príslušných hodinových cykloch podľa výstupnej funkcie „ v “ daného FSM. Tak napr. v stave Reset je to agent **Reset**, pri IF0 je to **LdMAR || IncrPC**, v stave IF1 – **EA**, v stave IF2 – **SetMRd**, atď**.** Tieto výstupy (mimo agenta **Reset**) sme v prechodovom grafe vynechali. Možno si ich pozrieť v nasledujúcej prechodovej tabuľke.

Z tohto modelu CPU daným FSM môžme dobre rozlíšiť množinu **riadiacich stavov R** od množiny **operačných stavov Q**. Množina **R =** {reset,IFO,IF1,…BR1} vystupuje v Moorovom FSM, ktorý modeluje správanie riadiacej časti CPU. Množinu **Q** tvorí množina vektorov hodnôt operačných stavových premenných v danom modeli CPU, teda množina **Q** = {(<MBR>, <MAR>, <PC>, <IR>, <AC>) }.

Agenty – **mikrooperácie**, ktoré sú ako výstupné symboly priradené riadiacim stavom FSM, v systéme CPU majú funkciu **nastavovania hodnôt** operačných stavových premenných z **Q** (v operačnej časti CPU) podľa akceptovaných hodnôt primárnych vstupov CPU a elementárnych funkcii v operačnej časti CPU (v danom prípade súčet „+“). Uvedený FSM je vlastne vyššie zavedený komunikačný stavový stroj (KSM) opisujúci „nekonečnú komunikáciu“ generovanú v nekonečnom procese **PrGlobal** v cieľovej špecifikácií CPU-2 uvedenej vyššie.

Špecifikácia v danej prípadovej štúdií CPU obsahuje iba jeden globálny proces **PrGlobal**. Ide o čisto sekvenčný systém bez súbežných globálnych procesov. Teda ide o jednu primárnu architektúru s jednou sekvenčnou riadiacou jednotkou