POPIS DIGITÁNÍHO NÁVRHU POMOCÍ VHDL

PRAVIDLA PRO PSANÍ SYNTETIZOVATELNÉHO KÓDU

1. Účel tohoto dokumentu

Následující pravidla se týkají problematiky jak správně popisovat obvod ve VHDL, tak aby se návrhář vyhnul problémům při jeho syntéze. Zároveň však do značné míry pokrývají i problematiku digitálního návrhu obecně. Správné psaní syntetizovatelného VHDL totiž není možné bez důkladného pochopení toho jak obvod bude implementován po syntéze. Ačkoli předpokládám, že čtenář je již seznámen se základními dovednostmi číslicového návrháře, následující kapitolu by neměl pro jistotu přeskakovat. Dalším předpokladem pro pochopení tohoto dokumentu je alespoň základní znalost syntaxe jazyka VHDL.

2. Obecné zásady digitálního návrhu

Zásada č.1 Znát úroveň abstrakce
Číslicové obvody se dnes navrhují metodou popisu v jazyce VHDL. Tento jazyk umožňuje popsat obvod na různých úrovních abstrakce. Toho se při návrhu často využívá. Například při psaní simulačních modelů, kde nás nezajímá konkrétní implementace, můžeme vystačit s plně funkčním popisem do značné míry připomínající klasické programování. Tam je možné použít celé spektrum konstrukcí jazyka VHDL.
Je třeba si též uvědomit, že VHDL původně vznikl jako simulační jazyk, kde mělo být možné postupně přecházet od modelů čistě funkčních k modelům podrobnějším až dospějeme k výslednému popisu struktury obvodu. Číslicová syntéza je vlastně procesem dodaným až později za účelem automatizace přechodu mezi úrovněmi abstrakce. Proto je důležité vědět mezi jakými úrovněmi abstrakce syntéza přechází.
V současné době umí syntéza automaticky převést obvod popsaný v jazyce VHDL na úrovni přesunů mezi registry (Register Transfer Level = RTL) do úrovně hradlové - netlistu základních elementů dané technologie.
Z toho plyne potřeba definovat určitou podmnožinu konstrukcí jazyka VHDL, které jsou podporovány syntézním procesem. Zároveň je nutné aby návrhář pochopil co znamená “navrhovat obvod na úrovni RTL”.
Základní charakteristikou úrovně RTL je striktní vymezení toho, co jsou paměťové elementy (registry) a kombinační logika. Čas na úrovni RTL je diskrétní a je definován “tiky” hodinových signálů.
Pokud návrhář dodrží stanovená pravidla dané úrovně, vyhne se celé řadě problémů. Popisy na vyšší úrovni zpravidla vedou na nesyntetizovatelný kód. (Vyšší úrovně se někdy nazývají “behaviorální” - je to svým způsobem nevhodný název neboť i na úrovni RTL popisujeme chování obvodu - angl. chovat se = to behave).
Popsat obvod na nižší jakési pseudohradlové úrovni je sice také možné, ale kromě zbytečné pracnosti a nepřehlednosti tyto popisy často vedou na neoptimální výsledky po syntéze.

Zásada č.2 Chápat způsob implementace
Jak je uvedeno v předchozí zásadě, popisujeme obvod na úrovni RTL. Proto je nutné aby návrhář myšlenkově chápal jak bude jím popsaný blok implementován - musí tedy být schopen odlišit od sebe registry a kombinační logiku. Při vlastním popisu je možné tyto věci směšovat (např. je možné v jednom procesu popsat část registru a kombinační logiky), návrhář však vždy musí být schopen nakreslit schema daného bloku (s oddělením registrů a kombinační logiky). Proto platí zásada - nakrestlit schema bloku před jeho popisem v jazyce VHDL.
Neboť začátečníci - zejména s praxí programování - velice často mají problémy s tímto způsobem uvažování, doporučuji jim, aby kombinační logiku a registry od sebe oddělovali i při vlastním VHDL popisu (viz doporučení další kapitoly).

Zásada č.3 Navrhovat synchronně jak je to jen možné
Čtenář může nabýt dojmu, že pojem synchronní návrh se stal jistým zaklínadlem. Méně často je však vysvětleno, co se za tímto pojmem vlastně skrývá. Proto zde uvádím tzv. zlatá pravidla synchronního návrhu. Tato pravidla jsou rozvedena v další kapitole podrobněji. Je třeba ovšem uvést, že za všech okolností není

možné dodržovat tato pravidla, případů, kde obvod ***nelze*** navrhnout synchronně je však výrazně méně než si začínající návrhář obvykle myslí.

7 pravidel synchronního návrhu

1. Na hodinové vstupy všech prvků jsou přivedeny pouze hodinové signály.
2. Jsou použity pouze hodinové signály jednofázové s pevným kmitočtem.
3. Všechny klopné obvody jsou hranově řízené (flip-flop ne latch).
4. Asynchronní přednastavení a nulování může být použito pouze k nastavení počátečního stavu klopných obvodů, ne v průběhu jejich běžné funkce.
5. Zpětné vazby v kombinačních obvodech se v návrhu nevyskytují.
6. Výstupy z dekodérů a přenosy čítačů nejsou použity jako hodinové signály.
7. Asynchronní signály jsou synchronizovány pomocí klopného obvodu před jejich rozvedením do více funkčních jednotek .

Pro věčné pochybovače ještě uvedeme několik důvodů pro synchronní návrh.

Proč synchronní návrh ?

- Obvody navržené při dodržení pravidel synchronního návrhu jsou snadno testovatelné. Asynchronní obvody obsahující zpětné vazby způsobují značné problémy pro produkční testování.
- Synchronní obvody jsou odolnější vůči šumu vznikajícím v důsledku přeslechů, odrazů, kombinačních hazardů, nestability zemních potenciálů, vlivům teplot, ...
- Funkčnost návrhu nezávisí na konkrétním rozmístění a propojení, tím je dána pouze dosažená rychlost funkce.
- Při dodržení pravidel synchronního návrhu je k ověření postačující funkční simulace, simulace časová slouží pouze k ověření dosaženého hodinového kmitočtu.
- U čistě synchronních obvodů s jedním hodinovým signálem vždy simulace funkční musí souhlasit se simulací časovou.

Výše uvedené neznamená, že synchronní návrh nemá své nevýhody. Problémem na větších čipech je mimo jiné samotný globální rozvod hodinového signálu, který by měl do každého klopného obvodu dorazit ve stejný okamžik, synchronní obvody také mívají vyšší spotřebu. Proto nejsou tato pravidla stanovena jako dogma. Každý návrhář by však měl důkladně uvážit, je-li nezbytné aby některé z těchto pravidel porušoval. Při respektování pravidel synchronního návrhu se vyvaruje mnoha problémů.

3. Zásady a doporučení

V následujících kapitolách uvedeme sadu pravidel, kterými se řídí popis obvodu v jazyce VHDL pro syntézu. Pravidla se děli do dvou skupin - na zásady a doporučení. Zásady jsou striktní pravidla, ze kterých nelze ustupovat (všimněte si, že skutečných zásad není mnoho). Doporučení jsou méně striktní, ale výjimky by měly být důkladně zváženy.

3.1. Obecné zásady a doporučení

Doporučení č. 1 - Používej standardní popis registrů hranově řízených (flip-flop)
Jako sekvenční elementy v návrhu je doporučeno používat téměř výhradně registry hranově řízené (anglicky flip-flops). Příklad 1 ukazuje doporučený standardní popis, který po syntéze bezpečně vede na hranově řízený klopný obvod typu D tedy DFF. Jiné popisy využívající příkaz WAIT nebo paralelní přiřazení (mimo proces) se nebudou používat v rámci firemních projektů. (Tyto příkazy mají sice stejné chování, jejich podpora u jiných syntézátorů však není zaručena. Mohli bychom mít zbytečné problémy s portabilitou kódu.)
Nedoporučujeme inicializovat signály v deklaraci. Tato počáteční hodnota není podporována syntézou. Jediný způsob jak nastavit registru počáteční hodnotu je použití asynchronního resetovacího signálu.
Některé technologie - např. FPGA poskytují buňky DFF se signálem clock enable. Jedná se o multiplexer předřazený D klopnému obvodu. Syntézní programy se zlepšují v rozpoznávání signálů použitelných jako clock enable, příklad č.2 ukazuje doporučený styl popisu, který by měl vést na obvod DFF se signálem CE.

Příklad č.1 Standardní popis DFF s asynchronním a synchroním resetem

```
SIGNAL q : std_logic:='0';  -- pocatecni hodnota je syntezou ignorovana !

SIGNAL q : std_logic; -- spravna deklarace

clkp: PROCESS(clk,res_asy)
BEGIN
 IF res_asy='1' THEN q<='0';
  -- asynchronni reset - pocatecni nastaveni
 ELSIF clk'event AND clk='1' THEN
   IF res_syn='1' THEN q<='0'; -- synchronni reset (kombinacni logika na vstupu
 DFF)
     ELSE q<=d; -- normalni funkce
    END IF;
END IF;
END PROCESS clkp;
```

Poznámka : Z hlediska návrhu pro snadnou testovatelnost se doporučuje vyvést u každého klopného obvodu asynchronní resetovací signál. (Resetem zde rozumíme asynchronní přednastavení do nuly nebo do jednicky.)

Příklad č.2 Registr DFFCE s asynchronním resetem a signálem clock enable (spec. pro FPGA)

```
clkp: PROCESS(clk,res_asy)
BEGIN
 IF res_asy='1' THEN q<='0';
  -- asynchronni reset - pocatecni nastaveni
 ELSIF clk'event AND clk='1' THEN
   IF ce='1' THEN q<=d; -- clock enable
   END IF;
 END IF;
END PROCESS clkp;
```

Zásada č.1 - Nepoužívej hladinově řízené obvody (latch)
Ačkoli jsou registry typu latch menší než typ flip-flop jejich použití přináší do návrhu další parametr. Kromě hodinové frekvence záleží i na střídě hodinového signálu resp. na délce aktivního pulsu hodinového signálu.
Proto se hladinově řízené obvody při RTL návrhu nepoužívají. Výjimky ovšem existují - jedná se o případy z návrhu velkých ASIC obvodů (např. velká registrová pole, FIFO, 2 fázové synchronní RAM). FPGA návrh by se měl bez obvodů typu latch zcela obejít. O případném použití obvodů typu latch rozhoduje příslušný projektový vedoucí.

Poznámka: Často jsou latche generovány jako **důsledek špatného popisu kombinační logiky**. Syntézní nástroj hlásí to jako varování při spuštění funkce READ. Před vlastní funkční verifikací se doporučuje načíst soubory do syntézního programu a zkontrolovat přítomnost zmíněných “warnings”.

Příklad č.3 Latch syntetizován v důsledku chybějící větve ELSE

```
komblog: PROCESS(a,b)
BEGIN
  IF a='1' THEN
     c<=b;
  -- chybejici vetev  ELSE  znamena, ze c si pri a='0' pamatuje hodnotu !
   END IF;
END PROCESS komblog;
```

Příklad č.4 Syntetizován latch v důsledku chybějícího přiřazení výstupu z

```
komblog: PROCESS(c)
BEGIN
  CASE c IS
    WHEN ´0´ => z<='0'; q<='1';
    WHEN others  => q<='0';
       -- vystup z neni definovan => pamatuje si hodnotu => implementovan latchem
  END CASE
END PROCESS komblog;
```

Příklad č. 5 Syntetizovány latche v důsledku neúplného přiřazení std_logic_vectoru

```
ps_toggle: PROCESS (curr_toggle,
             reop,me,packet_err,
             r_pid,curr_endp)
  BEGIN
    IF (reop = '1' AND me = '1' AND
        packet_err = '0') THEN
      IF (r_pid = DATA0_p) THEN
        next_toggle(conv_integer(curr_endp)) <= '1';
            -- chybi definovat ostatni bity next_togle !!!!
      ELSE
        IF (r_pid = DATA1_p OR r_pid = SETUP_p) THEN
          next_toggle(conv_integer(curr_endp)) <= '0';
            -- chybi definovat ostatni bity next_togle !!!!
        ELSE
          next_toggle <= curr_toggle;
        END IF;
      END IF;
    ELSE
      next_toggle <= curr_toggle;
    END IF;
  END PROCESS ps_toggle;
```

Poznámka : Problémům z příkladů 4 a 5 se lze vyhnout buď důsledným definováním stavu všech signálů ve všech větvích podmínek nebo přiřazením “default” hodnoty na začátku procesu.

Doporučení č.2 : Nevytvářej cykly mezi procesy
Výsledkem cyklu mezi procesy může být asynchronní obvod (implementován z hradel nebo pomocí hladinových obvodů) nebo zpomalení simulace.
Návrhář se snadno vyhne těmto problémům pokud nebude používat nepřehledné vnořené IF-THEN - ELSE sekvence (příklad č.6) a nebude do jednoho procesu sdružovat signály mající odlišné vstupy (viz proces cl1 v příkladu č. 7).

Příklad č. 6 Cyklus mezi procesy vedoucí na asynchronní chování

--a je vstup
--b, c interni signaly

```
cl1: PROCESS (a,b)
BEGIN
  IF a='1' THEN
     IF  b='0' THEN c<='1';
     ELSE c<='0';
     END IF;
  ELSE
     c<='1';
  END IF;
END PROCESS cl1;

cl2: PROCESS(c)
BEGIN
  IF c='1' THEN
    b<='0'; -- c zavisi na a,b ale b take zavisi na c !!!
   ELSE
    b<='1';
  END IF;
END PROCESS cl2;
```

Příklad č.7 Cyklus mezi procesy zpomalující simulaci

```
cl1: PROCESS(a,c)
BEGIN
  IF a='1' THEN
                b<='0';
                IF c='0' THEN d<='1';
                ELSE d<='0';
                END IF;
  ELSE
            B<='1';
            D<='0';
  END IF;
END PROCESS cl1;


cl2: PROCESS(b)
BEGIN
  IF B='1' THEN
      C<='0';
  ELSE
      C<='1';
  END IF;
END PROCESS cl2;

-- b=f(a), c=f(b), d=f(a,b)
-- process cl1 se vykonava vzdy dvakrat !
```

Zásada č.2 Vždy používej úplný a minimální citlivostní seznam (sensitivity list) u procesů
Proces je v simulátoru proveden při změně některého signálu v jeho citlivostním seznamu (neuvažujeme zde procesy se příkazy typu WAIT, které se v syntetizovatelném kódu nepoužívají).
V případě, že proces popisuje **kombinační logiku**, v citlivostním seznamu musí být uvedeny všechny signály na pravé straně přiřazovacích příkazů nebo řídící signály podmínek (IF, CASE), tedy všechny vstupy kombinační logiky.
Chybějící signály jsou zpravidla hlášeny syntézátorem jako varování. Syntézátor si obvykle dokáže chybějící signály "domyslet" na to však nelze spoléhat a je třeba chybu napravit. Varování tohoto typu se nesmějí vyskytovat v logovacím souboru syntézy.

Příklad č.8 Kombinační logika

```
komblog: PROCESS(a,b,c)
BEGIN
  IF c='1' THEN d<=a AND b;
    ELSE d<='1';
  END IF;
END PROCESS komblog;
```

V případě, že proces popisuje **sekvenční logiku** (registry řízené hranou) obsahuje citlivostní seznam pouze hodinový signál a asynchronní reset - viz příklad č.1. Procesy obsahující v citlivostním seznamu mix hodinových signálů a výstupů kombinační logiky svědčí buď o asynchronním návrhu nebo o nezkušenosti návrháře.

Poznámka : Začínajícím návrhářům se doporučuje striktně oddělovat kombinační a sekvenční procesy, dokud jazyk dostatečně neovládnou.

Doporučení č. 3 Preferuj signály před proměnnými
Proměnné nebývaly v minulosti vždy podporovány při syntéze. Dnes se situace(MTI to umi, LHR). Použití proměnných ovšem šetří paměť a zvyšuje rychlost simulace. **Zkušený návrhář** proto může proměnné použít např. pro odstranění cyklické závislosti procesu sama se sebou (příklad č. 9).

Příklad č.9 Převodník z grayova kódu

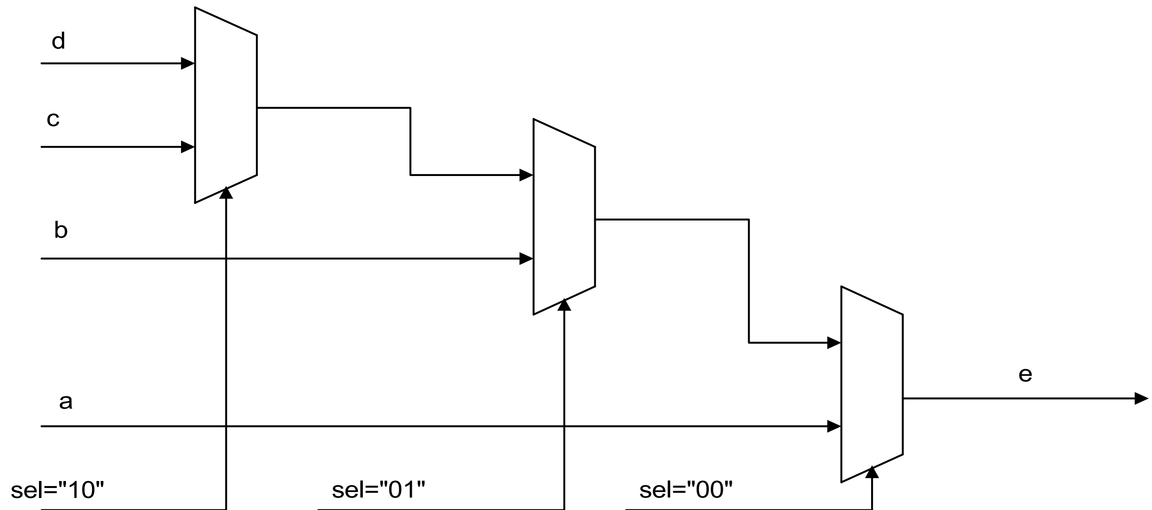
```
-- gray to binary conversion
gray2bin: PROCESS(QINT)
VARIABLE Qbinvar : STD_LOGIC_VECTOR(width-1 downto 0);
BEGIN
   Qbinvar(width-1):=QINT(width-1);
   FOR i IN width-2 downto 0 LOOP
      Qbinvar(i):=Qbinvar(i+1) xor QINT(i);
   END LOOP;
   Qbin<=Qbinvar;
END PROCESS gray2bin;
```

-- použití signálu místo proměnné Qbinvar je možné ovšem vede na zacyklení procesu sama na sebe
-- u jednodušších simulátorů to znamená, že proces je vykonáván $width^2$ -krát místo width-krát.

3.2. Použití multiplexerů a třístavových budičů

Doporučení č. 4 Preferuj CASE před vnořeným IF-THEN-ELSE
Toto doporučení se opírá dnes již o poměrně historickou zkušenost se syntézními programy, které na základě vnořeného IF-THEN-ELSE syntetizovaly kaskádní víceúrovňový multiplexer. Při popisu s využitím příkazu CASE je výsledkem stromová implementace multiplexeru, která je pochopitelně rychlejší.
Experimenty se současnou verzí syntézátoru Leonardo Spectrum tuto zkušenost sice nepotvrzuje, ovšem z důvodů portability kódu zachováváme toto doporučení.
Jiným argumentem ve prospěch příkazu CASE je větší simulační rychlost na některých typech simulátorů.

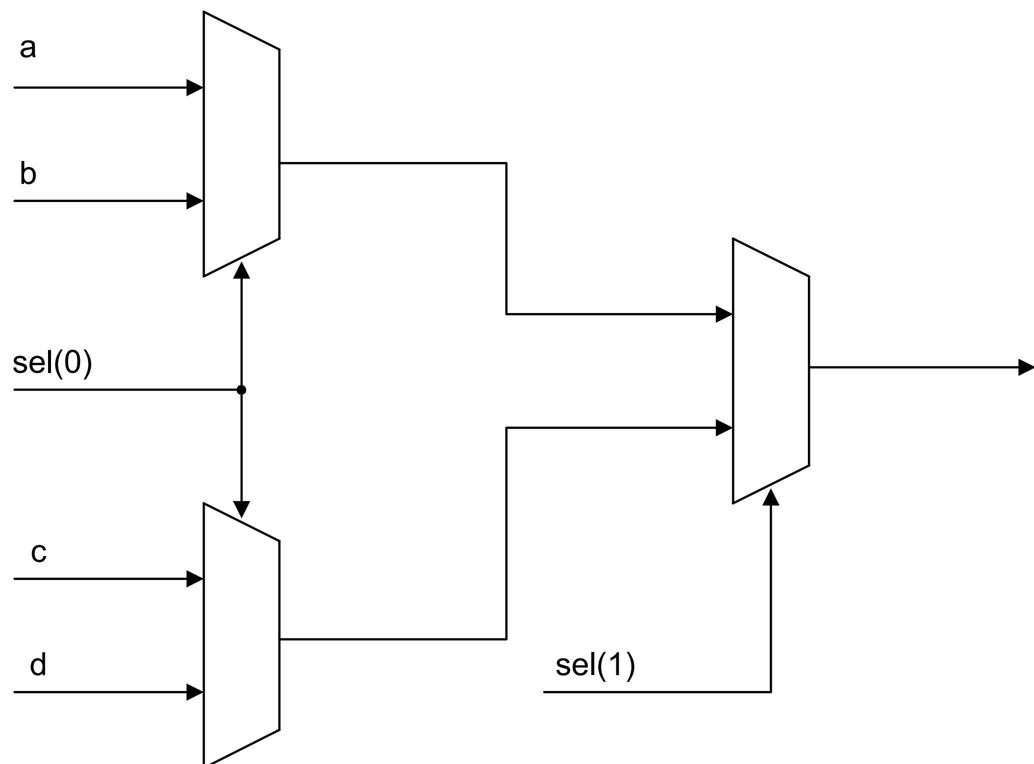
Obrázek 1 Kaskádní multiplexer někdy vznikající syntézou příkladu č.10



Příklad č. 10 Vnořený IF THEN ELSE vedoucí údajně na kaskádní multiplexer

```
pr1: PROCESS(a,b,c,d,sel)
BEGIN
  IF (sel="00") THEN
    e<=a;
  ELSIF (sel="01") THEN
    e<=b;
  ELSIF (sel="10") THEN
    e<=c;
  ELSE
    e<=d;
  END IF;
END PROCESS pr1;
```

Obrázek 2 Stromový (neprioritní) multiplexer vzniklý syntézou příkazu CASE (vždy)



Příklad č. 11 Funkčně ekvivalentní logika z příkladu 10 zapsána příkazem CASE

```
pr1: PROCESS(a,b,c,d,sel)
BEGIN
 CASE sel IS
  WHEN "00" => e<=a;
  WHEN "01" => e<=b;
  WHEN "10" => e<=c;
  WHEN "11" => e<=d;
  WHEN others => e<='-';
 END CASE;
END PROCESS pr1;
```

Multiplexer lze rovněž popsat pomocí podmíněného signálového přiřazení jako v příkladě č. 12. Jako nevýhoda tohoto zápisu je uváděn fakt, že na některých typech simulátoru je pomalejší. Návrhář může uvážit použití tohoto zápisu jako alternativy klasického IF-THEN-ELSE. Výhodou je, že odpadá psaní hlavičky procesu.
Pro rozvětvenější podmínky se doporučuje používat procesu a příkazu CASE.

Příklad č. 12 Multiplexer potřetí tentokrát jako podmíněné přiřazení

```
e<=a WHEN sel=”00” ELSE
      b WHEN sel=”01” ELSE
      c WHEN sel=”10” ELSE
      d;
```

Doporučení č. 5 Třístavové budiče používej s rozvahou (speciálně pro Xilinx FPGA návrh)
Třístavové budiče jsou nabízeny některými technologiemi např. FPGA Xilinx. Jejich použití je v zásadě dvojí - jako alternativa multiplexeru a k realizaci obousměrných sběrnic. Budiče jsou ideální ve sběrnicově orientovaných architekturách, kde potřebujeme realizovat spojení každý s každým pro větší počet bloků.
Návrháři může přijít výhodné používat třístavové budiče protože nekonzumují běžnou logiku a zdá se, že nevnášejí zpoždění jako rozvětvený multiplexer (to může být často jen zdání neboť dlouhé sběrnice s velkou kapacitní zátěží jsou pomalé).
Zkušenosti ukazují, že existuje trade-off mezi použitím třístavových sběrnic a multiplexerů (a dvoubodových spojů). Hlavní nevýhodou velkého použití třístavových budičů je fakt, že v FPGA konzumují “long lines”, kterých je samozřejmě limitovaný počet. Zejména při návrhu obsahujícím hodně vzájemně izolovaných třístvových sběrnic (ale třeba i jednu 32 a více-bitovou sběrnici) se potom může stát, že ačkoli FPGA není příliš zaplněno, obvod se nepropojí. (FPGA Virtex posouvají hranici opět dále, ovšem ani tam se nevyplatí problém ignorovat.)
Optimální se jeví i u sběrnicových systémů sdružovat několik vysílačů přes multiplexer ke společnému třístavovému budiči (multiplexer 1-2 nebo 1-4 příliš zpoždění nepřidá).

Pro popis třístavového budiče je osvědčené použít stručný zápis z příkladu č. 13.

Příklad č.13 Třístavový budič

```
bus<= data_out WHEN oe=’1’ ELSE (OTHERS=>’Z’);
```

3.3. Použití konečných automatů (FSM)

Zásada č. 3 Při popisu automatu používej tři procesy - dva kombinační procesy - pro přechodovou funkci automatu a pro výstupní funkci automatu a jeden sekvenční proces pro popis paměťových elementů.
Některý z kombinačních procesů se může redukovat na podmíněné nebo prosté signálové přiřazení - typicky jde o popis výstupní funkce automatu.
Důvodem je jednak přehlednost popisu a jednak zaručená syntéza očekávané struktury.

Doporučení č.6 Používej výčtový (enumerated) typ pro reprezentaci stavu
Použití výčtového typu jednak zpřehledňuje funkční simulace, jednak umožňuje měnit kódování automatu při syntéze - viz dokument o postupu při syntéze.

Zásada č.4 Automaty separuj v samostatném bloku od jiné logiky
Separace automatů do samostatného bloku umožňuje jejich separátní optimalizaci při syntéze - více viz doporučení pro dekompozici obvodu do bloků.

Příklad č. 14 Popis automatu metodou 3 procesů

```
TYPE machine2_state_type IS (
    LOCKED,
    FREE
  );

SIGNAL machine2_current_state, machine2_next_state : machine2_state_type ;

BEGIN

machine2_clocked : PROCESS (clk, res)
BEGIN
    IF (res = '1') THEN
      machine2_current_state <= FREE;
      -- Reset Values
    ELSIF (clk'EVENT AND clk = '1') THEN
      machine2_current_state <= machine2_next_state;
      -- Default Assignment To Internals
    END IF;
  END PROCESS machine2_clocked;

machine2_nextstate : PROCESS (FRAME, Hit,LOCK, L_lock,machine2_current_state)
BEGIN
    CASE machine2_current_state IS
    WHEN LOCKED =>
      IF ((FRAME AND LOCK)='1') THEN
        machine2_next_state <= FREE;
      ELSE
        machine2_next_state <= LOCKED;
      END IF;
    WHEN FREE =>
      IF (((NOT FRAME AND LOCK AND Hit AND (STATE_IDLE OR STATE_TURN_AR)) OR
(L_lock AND Hit AND STATE_B_BUSY))='1') THEN
        machine2_next_state <= LOCKED;
      ELSE
        machine2_next_state <= FREE;
      END IF;
    WHEN OTHERS =>
      machine2_next_state <= FREE;
    END CASE;
  END PROCESS machine2_nextstate;

machine2_output : PROCESS (machine2_current_state)
BEGIN
    -- Default Assignment
    -- Default Assignment To Internals
    -- State Actions
    CASE machine2_current_state IS
    WHEN LOCKED =>
      STATE_LOCKED<='1';
      STATE_FREE<='0';
    WHEN FREE =>
      STATE_FREE<='1';
      STATE_LOCKED<='0';
    WHEN OTHERS =>
      NULL;
    END CASE;
 END PROCESS machine2_output;
```

3.4. Dekompozice obvodu do bloků z hlediska syntézy

Problém dekompozice obvodu do bloků je základní metodou používanou v průběhu standardní metodologie návrhu “shora-dolů”. Je obtížné dát univerzální doporučení podle jakých hledisek obvod dekomponovat. Smyslem této kapitoly je stanovit pravidla na blokovou dekompozici z hlediska snadné a optimální syntézy.

Základní algoritmus syntézy z úrovně RTL provádí převod abstraktního popisu kombinační a sekvenční logiky do konkrétní obvodové reperezentace - hradel a klopných obvodů dané technologie. S výjimkou některých speciálních bloků (jako např. násobiček) nemění polohu registrů vůči kombinační logice. Taková operace by vedla na obvod s odlišným časovým rozvrhem operací - obvod RTL nenekvivalentní. Jedinou věcí, kterou syntéza optimalizuje je kombinační logika mezi registry (také ovšem eliminuje registry s trvalou konstantní hodnotou výstupu, provádí vyrovnávání zátěže výstupů pomocí replikace logiky a pod.). Registry tvoří přirozenou bariéru pro optimalizaci kombinační logiky.
Na hranice mezi bloky se většina návrhářů dívá jako na virtuální hranice rozčleňující obvod na části myšlenkově pochopitelné části. Syntézátor ovšem implicitně chápe hranice mezi bloky jako další umělé bariéry pro optimalizaci kombinační logiky. Z těchto dvou faktů plyne následující doporučení a zásada.

Doporučení č. 7 Snaž se aby výstupy z bloků byly registrované
Pokud chceme umožnit syntéze optimalizovat kombinační logiku je nevhodné vkládat do ní umělou bariéru v podobě hranice bloku. Jinými slovy - výstupy z bloků by měly být výstupy z registrů - viz obrázek č. 3.
Obvod dobře dekomponovaný dává potom lepší výsledky při syntéze s přepínačem “preserve hierarchy” než “flatten”. Dalším důvodem registrovaných výstupů je usnadnění tvorby časových “constraints”.
Je zřejmé, že dodržovat toto doporučení striktně by značně omezovalo užitečnou blokovou dekompozici. Proto moderní syntézní programy jako např. Leonardo Spectrum umožňují rozpouštět hranice mezi bloky (disolve hiearchy). U větších obvodů je však nutné provádět optimalizaci na rozumně velkých blocích - při rozpuštění hierarchie (flattening) celého obvodu dávají optimalizační algoritmy špatné výsledky nebo běží příliš dlouho.
Proto zavádíme následující zásadu.

Zásada č. 5 Bloky obsahující 1 tisíc a více ekvivalentních hradel musí mít registrované výstupy
Dodržení této zásady nám umožní při syntéze rozpustit hierachii až po učitou mez, kde je možno provádět optimalizaci na blokové úrovni. Leonardo Spectrum sice umožňuje spojovat bloky i mimo vlastní hierarchii, nelze na to však spoléhat u jiných syntézních nástrojů.
Při správném postupu dekompozice je vhodné též provádět “time-budgeting” - vymezení dílu zpoždění na daný blok. Bloky s registrovanými výstupy mají tento proces usnadněn.

Doporučení č. 8 Snaž se držet kombinační logiku pohromadě v jednom bloku společně s příslušnými registry
Ve spojení s předchozími doporučeními se snažíme naznačit, aby se návrhář snažil vyvarovat samostatných bloků čistě kombinační logiky. Zejména by se neměly tyto bloky tzv. “glue logic” vyskytovat na nejvyšší úrovni hierarchie. Důvodem je opět omezení efektivity optimalizace a obtížné časové rozpočtování (time-budgeting).

Doporučení č. 9 Separuj časově kritickou logiku od časově nekritické
Toto rozdělení umožní pouštět odlišné optimalizační algoritmy na kritickou a nekritickou logiku. Typicky kritická logika je optimalizována na rychlost a nekritická na plochu. Volba typu algoritmu je možná pouze na blokové úrovni proto je vhodné takto logiku separovat.

3.5. Asynchronní logika

Doporučení č. 10 Snaž se vyhnout asynchronní logice
Asynchronní logika je daleko obtížněji analyzovatelná a je mnohem obtížnější navrhnout ji bezchybně a verifikovat ji. Implementace je obecně technologicky závislá z hlediska časových parametrů a to značně omezuje její portabilitu.

Zásada č.6 O případné asynchronní implementaci rozhoduje vedoucí projektu a musí své rozhodnutí obhájit před týmem starších návrhářů.

Doporučení č.11 Je-li asynchronní logika nezbytná,umísti jí do zvláštního bloku
Separace asynchronní logiky umožní snadnější analýzu kódu. Asynchronní logika musí být analyzována mnohem pečlivěji z hlediska funkčnosti a časových parametrů.

3.6. Aritmetické operátory

Zásada č. 7 Pro aritmetické operátory používej knihovny IEEE.std_logic_arith nebo IEEE.std_logic_unsigned nebo IEEE.std_logic.signed.

Výběr se řídí následujícími pravidly :

a) Pokud blok pracuje pouze s čísly bez znaménka používej knihovnu IEEE.std_logic_**unsigned**
b) Pokud blok pracuje pouze s čísly se znaménkem (v doplňkovém kódu) používej knihovnu IEEE.std_logic_**signed**
c) Pokud blok pracuje s čísly bez znaménka a se znaménkem použij knihovnu IEEE.std_logic_**arith**

Správný zápis aritmetické operace záleží na použité knihovně !!!
V knihovně **signed** a **unsigned** jsou definovány příslušné aritmetické operace přímo pro typ **std_logic_vector** (Ten je chápán jako číslo se znaménkem nebo bez znaménka v závislosti na použité knihovně). V knihovně **arith** je nutno explicitně přetypovat std_logic_vector na **SIGNED** nebo **UNSIGNED.** (*Některé simulátory vyžadují i zpětné přetypování na std_logic_vector.*) Výsledkem syntézy je příslušná jednotka (kombinační logika) pro čísla v daném formátu. Příklad č. 5 ukazuje zápis sčítačky.

Příklad č. 15 Popis sčítání

a) Pro knihovny std_logic_unsigned a std_logic_signed :

SIGNAL a,b,c,d,z : std_logic_vector (...)

```
IF ctl=´1´ THEN
      z<= a + b;
ELSE
      z<= c + d;
END IF;
```

b) Pro knihovnu std_logic_arith

SIGNAL a,b,c,d,z : std_logic_vector (...)

```
IF ctl=´1´ THEN
      z<= unsigned(a) + unsigned(b);
ELSE
      z<= unsigned(c) + unsigned(d);
END IF;
```

Další příklady ukážeme pro knihovnu **std_logic_arith**.

Důležité je i správné přetypování na ***integer*** , zde při použití SIGNED místo UNSIGNED dochází k odlišnému výsledku - viz příklad číslo 16. Využití typu ***integer*** je nejčastější v adresových dekodérech.
Příklad č. 17 ukazuje popis obousměrného dvojkového čítače. Všimněte si, že jednička je u inkrementace a dekrementace uvedena bez apostrofů - jde o operátor + 1 resp. -1, který je v knihovně "přetížen" - definován pro typy integer, signed a unsigned.

Příklad č. 16 Přetypování na integer a zpět

```
SIGNAL a : std_logic_vector (3 downto 0);
SIGNAL a_int : integer RANGE 0 to 15
SIGNAL b : std_logic_vector (3 downto 0);

BEGIN

a_int<= conv_integer(unsigned(a));

b<= to_std_logic_vector(a_int,4);

END;
```

Příklad č. 17 Popis čítače

```
IF up='1' THEN
    nxt_cnt<= unsigned(cnt) + 1;
ELSE
    nxt_cnt<= unsigned(cnt) - 1;
END IF;
```

Doporučení č. 12 : Neseparuj aritmetické operátory do zvláštních bloků
Charakteristickou vlastností aritmetických jednotek je možnost jejich sdílení pro více operací. Popis na příkladě č. 15 může vést buď na syntézu dvou sčítaček s multiplexerem na výstupu nebo jedné sčítačky s multiplexery na vstupech. Syntézní program má tedy možnost optimalizace. V případě, že sčitání je popsáno v separátním bloku, optimalizace je limitována hranicemi bloku a proto nedojde ke sdílení sčítaček.

Závěrem ještě několik poznámek : Všechny syntezátory předpokládají použití řádové mřížky celých čísel. V případě DSP aplikací se častěji používají jiné řádové mřižky - např. tzv. zlomkový tvar. Návrhář proto musí mít elementární znalosti v oblasti číslicové aritmetiky (viz např. Pluháček A. : Projektování Logiky Počítačů).

Několik otázek na které by si měl umět odpovědět každý návrhář aritmetiky:

1. Lze použít stejnou sčítačku/odčítačku pro čísla **SIGNED** a **UNSIGNED** ?
2. Lze použít stejnou násobičku pro čísla **SIGNED a UNSIGNED ?**
3. Jak je třeba modifikovat sčítačku aby šla použít jako odčítačka ?
4. Lze použít stejnou násobičku pro čísla celá a ve zlomkovém formátu ?

3.7. Dekompozice z hlediska rychlé syntézy a tvorba "constraints"

V minulosti se doporučovalo používat k popisu relativně malé bloky k dosažení rozumné doby syntézy. Dnes vzhledem k růstu rychlosti syntézních algoritmů i pracovních stanic jsou hlavními hledisky dekompozice logická funkce a časové a plošné požadavky. Vhodná dekompozice společně s constraints má větší vliv na dobu syntézy než vlastní velikost bloku. Důležité je separovat logiku kritickou na rychlost od logiky kritické na plochu. To umožňuje používat odlišné optimalizace pro každý blok a výrazně snižuje dobu syntézy.
Tvorba vhodných "constraints" je relativně komplexní problém, který není možné postihnout v celé šíři v rámci tohoto dokumentu. Důležité je vyhnout se nerealistickým požadavkům - tzv. "overconstraining". Pokud výsledkem běhu programu bez "constraints" je obvod o frekvenci 50 MHz je realistické očekávat, že při vhodných "constraints" lze dosáhnout 60 - 70 MHz ovšem již velmi nepravděpodobně 100 MHz. (Pokud je cílem dosáhnout 100 MHz, je primárně nutné změnit RTL návrh s využitím pipeliningu a pod.)
Metodika tvorby constraints u větších obvodů se nazývá časové rozpočtování ("time budgeting"). V ranných stádiích návrhu je vhodné rozpočíst zpoždění jednotlivých bloků a na tomto základě vytvořit vhodné constraints. Stejné constraints potom slouží i při verifikaci s využitím statické časové analýzy.

3.8. Falešné a vícecyklové cesty

Doporučení č.13 Vyhýbej se použití vícecyklových cest
Vícecyklovou cestou (Multicycle path, point to point exception) rozumíme cestu v kombinační logice mezi zdrojovým a cílovým registrem, kterou se signál může šířit po dobu více než jednoho hodinového taktu než je zapsán do cílového registru.
Použití vícecyklových cest s sebou nese problémy při tvorbě constraints, statické časové analýze a také při pochopení funkce návrhu. Proto je vhodné se vícecyklových cest vyvarovat.

Doporučení č. 14 Pokud jsou použity vícecyklové cesty, umísti je do jednoho modulu a dobře je zdokumentuj. Modulem rozumíme blok na vyšší úrovni hierarchie. U návrhu obsahující jak vícecyklové, tak jednocyklové cesty je vhodné tyto cesty od sebe oddělit.

Doporučení č. 15 Vyhýbej se vzniku falešných cest
Falešná cesta (False path) je cesta v kombinační logice nesplňující časový limit, která však při reálné fukci obvodu nikdy není aktivována. Tyto cesty vznikají často ve sběrnicově orientovaných systémech a tam, kde mezi dvěma registry existuje více než jedna možná cesta (typicky zpětné vazby v ALU). Falešné cesty je nutné specifikovat do constraints neboť vedou na dlouhé časy při optimalizaci a statická časová analýza jinak dává špatné údaje. Návrh obsahující více falešných cest je nutno verifikovat časovou simulací a proto je vhodné se falešným cestám vyhýbat.

3.9. Návrh obsahující paměti

Paměti používáme v návrhu FPGA a ASIC velice často. Ať už k realizaci registrových polí nebo jako vyrovnávací a jiné paměti. Paměti ovšem představují problém při snaze o přenositelnost návrhu. Důvodem je fakt, že typ paměti je značně technologicky závislý. Typicky jsou podporovány buď tzv. ***synchronní*** nebo ***asynchronní*** RAM s různým rozhraním.

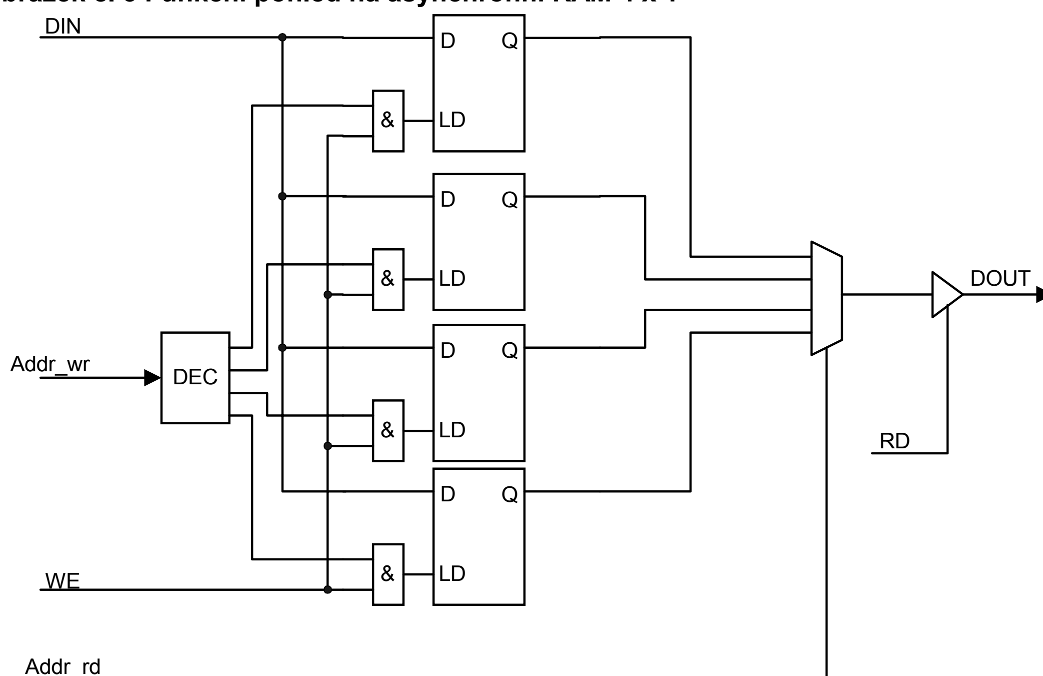
Nejprve provedeme malý úvod do terminologie, která je často velmi zmatená. Připomínám, že se budeme zabývat pouze STATICKÝMI pamětmi RAM, které jsou integrovány na čipech s logikou. Integrace DRAM představuje stále ještě značný technologický oříšek.

3.9.1. Přehled typů RAM

1. **Asynchronní RAM**

Asynchronní RAM je možné z hlediska funkce považovat za pole hladinově řízených klopných obvodů LATCH, se selektivním zápisem a čtením.

Obrázek č. 3 Funkční pohled na asynchronní RAM 4 x 1

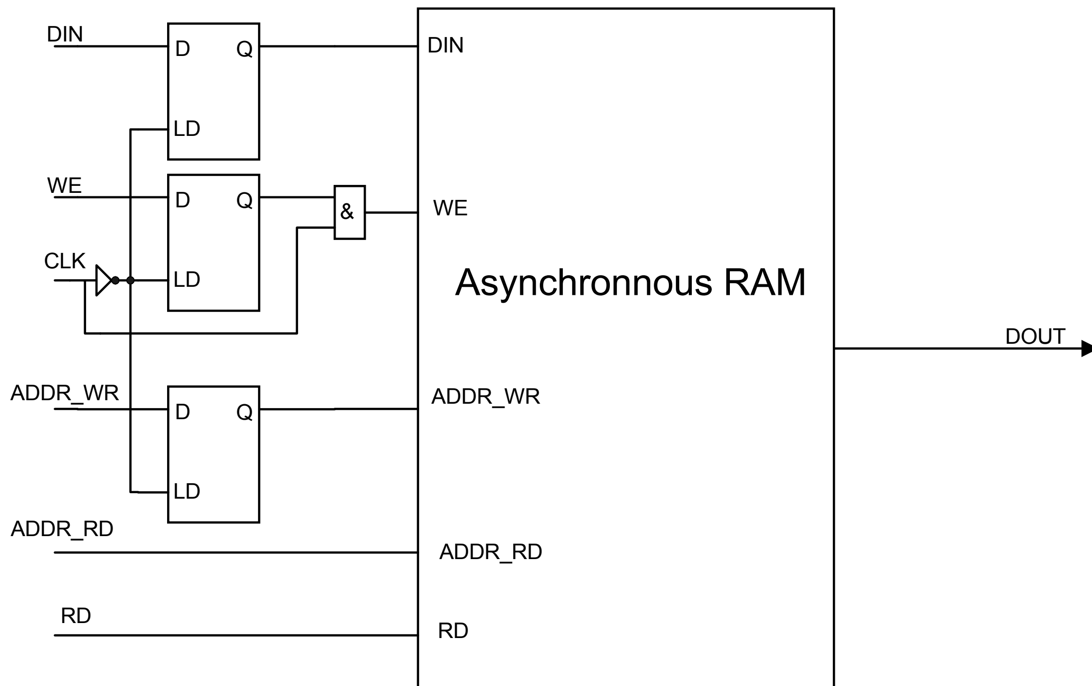


Pro čtení se chová jako kombinační logika (signál RD a třístavový budič může umožnit spojení DOUT a DIN a společnou adresu pro čtení a zápis). Pro zápis je důležité vygenerovat příslušný zápisový puls WE a dodržet stabilní signály Addr_wr a DIN. Výhodou asynchronní RAM je malá plocha, nevýhodou je obtížnější generování řídicích signálů.

2. **Synchronní RAM (též RAM se synchronním zápisem)**

Synchronní RAM je možné z hlediska funkce považovat za pole hladinově řízených klopných obvodů DFF, se selektivním zápisem a čtením.
V praxi se z důvodu úspory plochy realizuje jako Master / Slave, kde opět můžeme využít výše zmíněnou asynchronní RAM.
Na obrázku č. 4 je ukázka principiálního zapojení synchronní RAM s použitím asynchronní RAM. Všimněme si, že nyní přibyl hodinový signál CLK. Z hlediska čtení se paměť opět chová jako kombinační logika.

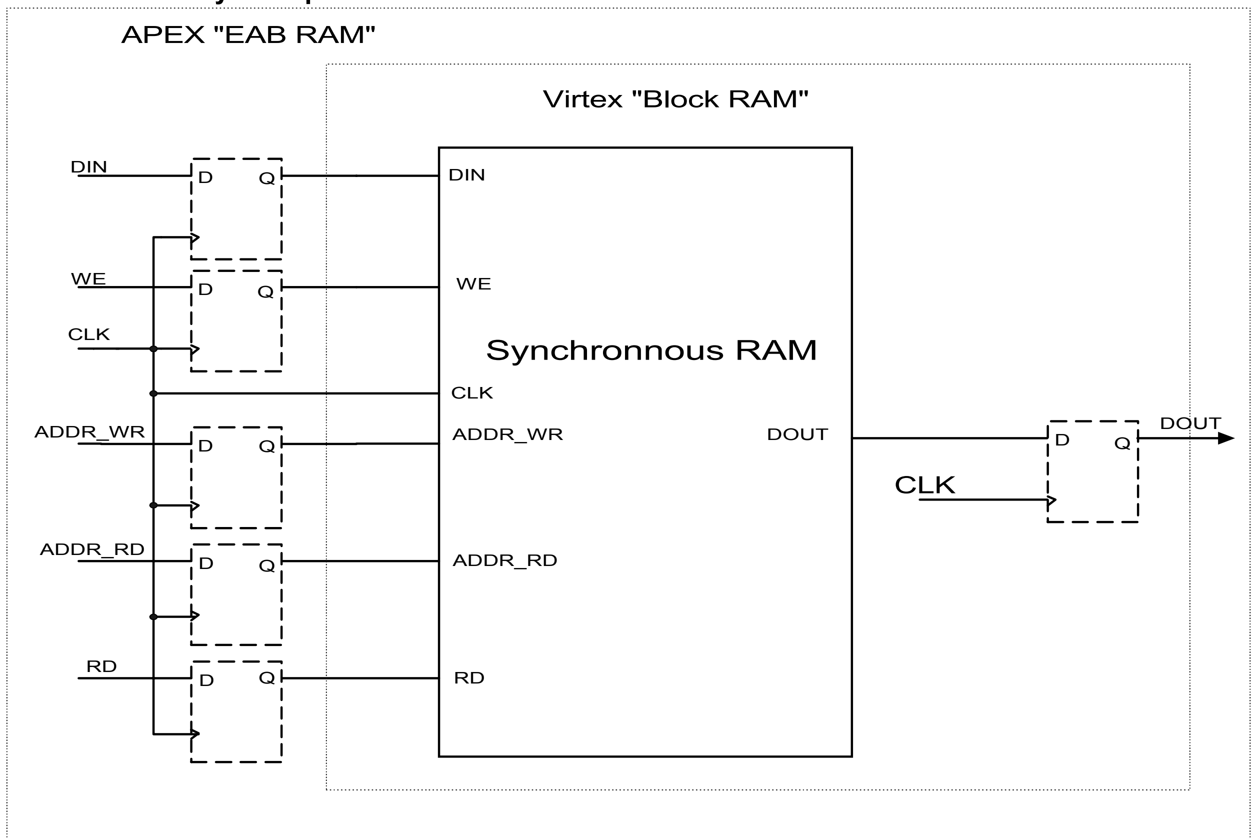
Obrázek č. 4 Realizace synchronní RAM s použitím asynchronní RAM



3. **Vícecyklové paměti**

Potřeba zajistit prediktabilní časování vede výrobce FPGA a ASIC k umísťování dalších registrů na adresové a datové signály. Pro návrháře to znamená, že pro čtení i zápis je nutno často počítat s dodatečnou latencí.
Obrázek č. 5 ukazuje příklad takové vícecyklové paměti. Čerchované čáry naznačují možná umístění dodatečných registrů.

Obrázek č. 5 Vícecyklové paměti



Jednotliví výrobci FPGA označují jako paměť nejrůznější kombinace dříve uvedených typů. Např. Xilinx řada 4000 podporovala asynchronní distribuovanou RAM, řada 4000E synchronní a asynchronní distr. RAM, Spartany a novější pouze synchronní distribuovanou RAM (náš typ č. 2). Tento typ je označován jako Select RAM. Kromě toho řada Virtex přináší tzv. Block RAM, která má ovšem registrovaný výstup a kde adresy musí být generovány z registrů (tyto registry jsou implementovány na poli v CLB). Co označuje Xilinx jako Block RAM je ukázáno na obrázku č. 5.
Konkurenční výrobce Altera používá pouze synchronní blokové paměti RAM v tzv. EAB (Embeded Array Block). Altera jako RAM označuje dokonce i registry na adresních vstupech, registr na výstupu naopak být nemusí.

Z výše uvedeného je zřejmé, že navrhnout obvod nebo core přenositelné mezi různými FPGA a ASIC je značně obtížné.

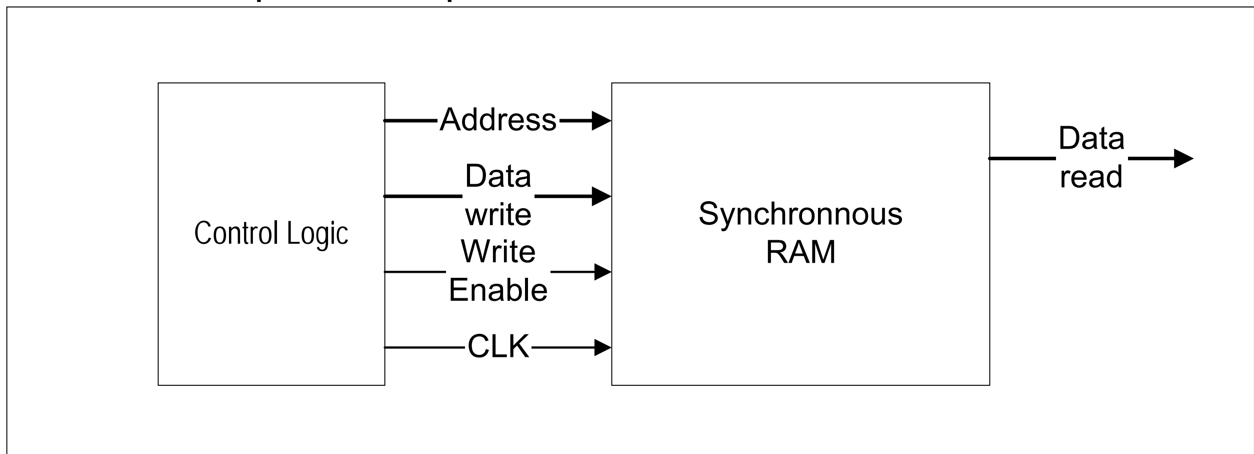
3.9.2. Paměti z hlediska syntézy

Tradiční cesta syntézy používá pro RAM technologicky specifické RAM generátory. Syntézní nástroje potom s RAM zacházejí jako s black boxem. To ovšem znamená, že RAM je bariérou statické časové analýzy.
Syntezátor Leonardo Spectrum je schopen FPGA RAM rozpoznat přímo z RTL popisu.

Pro snadnou přenositelnost návrhu mezi technologiemi i syntézními programy je vhodné dodržovat následující doporučení.

Doporučení č. 16 Odděl řídící logiku od paměti do samostatného modulu
Obrázek č. 6 ukazuje správnou dekompozici bloku s pamětí. Řídící logika by vždy měla být umístěna v separátním bloku od vlastního popisu paměti. To umožňuje obě cesty syntézy RAM. Blok paměťové matice může být chápán jako black box (syntéza se Synopsys, ASIC syntéza s Leonardem) nebo nahrazena RTL popisem a inferována z kódu (FPGA syntéza s Leonardem).

Obrázek č. 6 Dekompozice bloku s pamětí



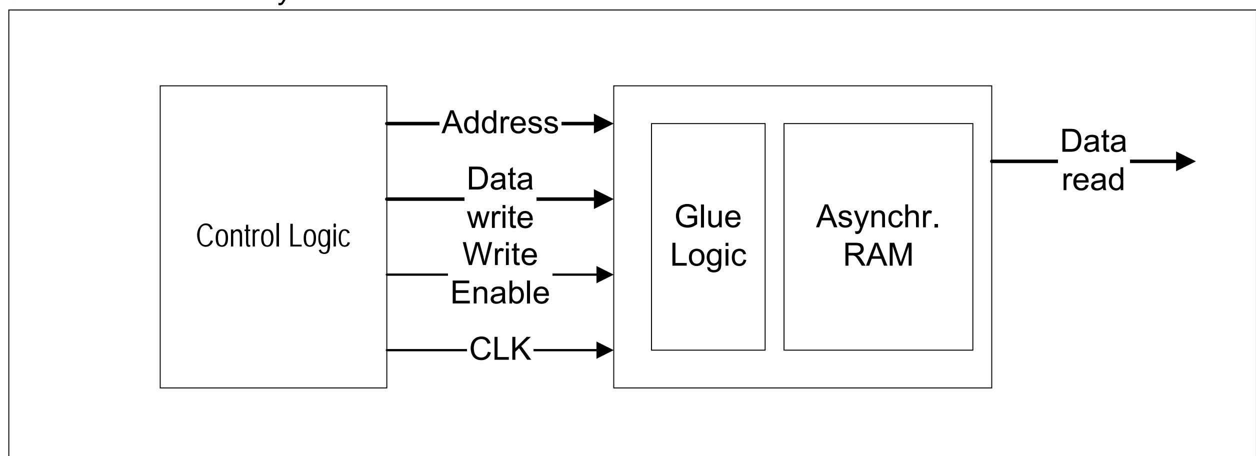
Doporučení č. 17 Používej synchronní paměti RAM
Ve spojení s předchozím doporučením to znamená, že řídící logika by vždy měla předpokládat synchronní paměť. V případě, že daná technologie nepodporuje synchronní RAM přímo, může být tato vytvořena s použitím glue logiky a asynchronní RAM. Tento příklad je ilustrován na obrázku č. 7. Standardní popis vedoucí na inferenci synchronní Select RAM pro Xilinx je uveden v příkladě č. 18.

Doporučení č. 18 Snaž se, aby adresa byla generována z registru a/nebo aby data byla čtena přímo do registru
Toto doporučení ve svém důsledku znamená, latenci 1 až 2 takty pro čtení. Paměť splňující plně tento požadavek je však bez problémů přenositelná mezi jednotlivými typy FPGA a ASIC obvodů i syntézního software.

Příklad č. 18 Standardní popis synchronní jednoportové Select RAM

```
ENTITY spram IS
GENERIC(widthA : integer := 4;
            widthD : integer := 8);
PORT(
     addr  :  IN std_logic_vector(widthA-1 downto 0);
     din   :  IN std_logic_vector(widthD-1 downto 0);
     ce_wr :  IN std_logic;
     wr_clk:  IN std_logic;
     dout  :  OUT std_logic_vector(widthD-1 downto 0)
     );
END spram;

ARCHITECTURE rtl OF spram IS

TYPE t_memory IS array (0 to 2**widthA-1) OF std_logic_vector(widthD-1 downto 0);

SIGNAL memory : t_memory;

BEGIN

write_proc: PROCESS(WR_CLK)
-- note : there is no reset in memory write description !
BEGIN
 IF wr_clk'event AND wr_clk='1' THEN
  IF CE_WR='1' THEN memory(conv_integer(unsigned(addr)))<=din;
  END IF;
 END IF;
END PROCESS write_proc;

read_proc: PROCESS(addr,memory)
-- note there is no clock for read, memory is combinatorial for read !
BEGIN
  dout <= memory(conv_integer(unsigned(addr)));
END PROCESS read_proc;

END rtl;
```

Obrázek č. 7 Použití asynchronní RAM



Poznámka : Zvláštní pozornost je třeba věnovat RAM v případě tvorbě constraints pro statickou časovou analýzu. RAM často znamená hranici statické časové analýzy. Pro zápis zde obvykle nedochází k problémům, neboť synchronní paměť RAM je ekvivalentní poli hranově řízených D klopných obvodů. Při čtení se ovšem RAM chová jako kombinační logika. Pokud RAM přerušuje cesty statické časové analýzy je možné, že nemusí proběhnout korenktně zvláště v případě, kdy adresa paměti je generována kombinační logikou. Proto je vhodné generovat adresu do paměti s použitím DFF. Jinak je nutné constraints specifikovat zvláště pro zpoždění do RAM a zpoždění z RAM.

4. Závěrem

Tento dokument představuje základní návod jak popisovat obvod v jazyce VHDL, tak aby tento popis byl snadno syntetizovatelný a navíc přenositelný na různé technologie. Autor doufá, že se mu podařilo začínajícího návrháře neznechutit seznamem omezení. Pokud text přesto tímto dojmem působí, autor zdůrazňuje, že drtivá většina omezení vychází často z těžce získaných negativních zkušeností z reálných projektů. Neboť všechno se stále vyvíjí mohou některá omezení být časem změněna. Autor proto uvítá jakékoli připomínky k tomuto textu.

5. Revize dokumentu

Číslo rev.	Autor	Datum	Popis
1.0	MBE	25.5.	Úvodní revize pro první kolo interních školení
2.0	MBE	23.11.	Kapitola o aritmetice přeformulována, zásada č. 7 změněna. Snížena velikost bloku v zásadě č. 5 (zkušenosti s ASIC syntézou) Kapitola o RAM doplňena o výklad a doporučení č. 18