Универзитет у Нишу
Електронски факултет
Катедра за рачунарство

Архитектура и организација рачунара
Вежбе, VHDL

Termin 5

Дизајн сложенијих кола

До сада упознати механизми VHDL-а су коришћени за пројектовање елементарних кола.
Прави смисао се достиже пројектовањем већих функционалних јединица, које извршавају неку функцију смислену за корисника.
У наставку ће бити описано једно сложеније коло, састављено од неких од описа приказаних раније у материјалу. Даћемо захтеве за колом и показати метод његовог пројектовања.

Z ПРИМЕР Систем са меморијом и аритметиком

Уведено: пројектовање сложенијег кола; коначни аутомат, принцип пројектовања

Потребно је реализовати коло које има улаз за такт, 8-битни улаз и 9-битни излаз података, и улаз за контролу рада кола WE. Када је WE=1, на сваки такт се очитава улаз података и смешта у интерну меморију капацитета 256 података. Када WE постане 0, на сваки такт се на излазу података поставља збир два суседна запамћена податка, редом како су памћени. Колико год да је збирова до тада израчунато, уколико поново WE постане 1, памћење података започиње из почетка (пребришу се први претходно запамћени подаци).

Анализа

Определићемо се за структуралан приступ пројектовању овог кола: Коло ће бити подељено на функционалне целине (компоненте) и установићемо сигнале којима ће компоненте бити повезане.

Установимо прво потребне компоненте.

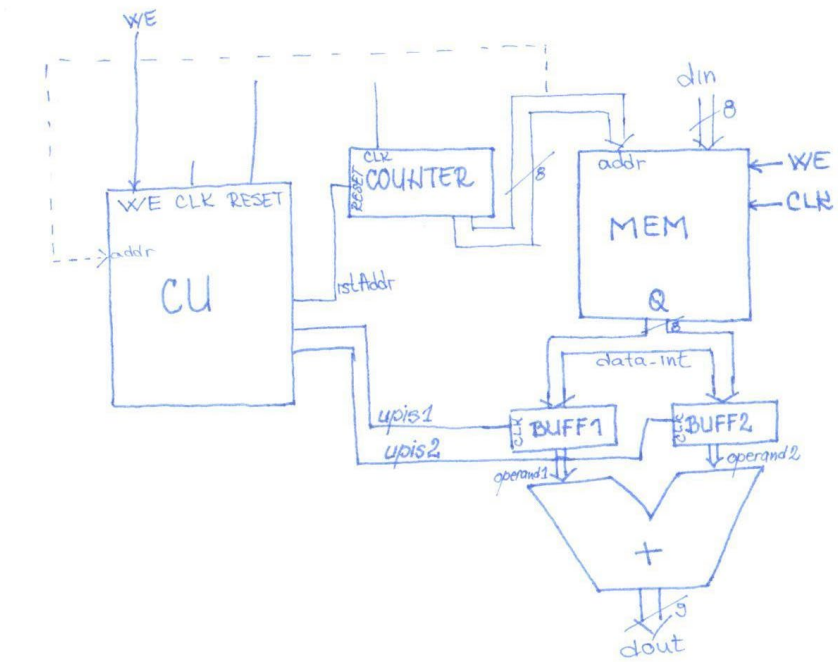
1. Пошто коло треба да памти податке, неопходна је меморија (слика 8).
2. Подаци се међусобно сабирају, стога нам је потребан сабирач.
3. Пошто се подаци у меморију уписују (и из ње читају) редом, за адресирање меморије може да се употреби бројач (на слици: COUNTER).
4. Како на излазном порту меморије у једном тренутку може да се налази један податак, а коло треба да сабира два податка, треба на улазима сабирача некако држати и податке које не издаје меморија у том тренутку. Из тог разлога, увешћемо и два регистра Меморија на свом излазу може да држи један податак у једном тренутку а неопходно је налазити збир два податка из меморије. Због тога треба да постоје регистри (на слици: BUFF1, BUFF2) који ће чувати податке добијене из меморије и након што престану да буду присутни на излазном порту.
5. На крају, потребно је управљати претходно побројаним компонентама. Локализујмо сву логику за управљање у једну компоненту и назовимо је управљачка јединица (на слици CU).

Установимо повезивање компонената и евентуалне додатне интерне сигнале:

- Излази сабирача (сума+излазни пренос) је оно што треба да буде на излазу кола, па су ови портови директно везани на излазни порт кола.
- Улаз података кола треба да се уписује у меморију, па овај порт може да се повеже директно на улазни порт меморије.

- Кроз меморију се ”иде” увек кроз сукцесивне локације, почев од нуле, па излаз бројача (уколико он броји унапред) можемо повезати на адресни улаз меморије.
- Адресе треба да се мењају на сваки такт, стога спољњи такт може да се доведе директно на clk порт бројача.
- Излаз из меморије се води на улазе регистара (сигнал data_int на слици 8), а тренуцима када податак треба да се упише у који регистар треба да се управља, одатле сигнали upis1 и upis2. Регистар иначе прима вредност на ивицу клока, тако да је контролне сигнале довољно довести на клок (није потребна друга посебна контрола за регистар).
- Сабирач је комбинационо коло; генерисаће нови резултат увек када му се промене улази. Променама на улазу сабирача диктира управљање регистрима BUF1 и BUF2, па сабирачу стога није потребна никаква друга посебна контрола (нпр. такт или сл.).
- Управљачка јединица треба да одређује када се уписује у који регистар (сигнали upis1 и upis2) и када се ресетују адресе (сигнал rstAddr). Треба управљати и режимом меморије, али се може уочити да меморија чита или издаје податке на сваки такт, па клок може да се доведе директно на меморију; а такође се у меморију уписује увек када је WE=1 а увек се чита када је WE=0, па се и WE може довести директно на WE порт меморије, нема потребе да CU управља режимом меморије.
- CU на сваки такт треба да мења регистар у који се уписује податак, зато клок треба довести на CU. На промену WE треба ресетовати адресе, тако да и WE треба довести на CU.
- Уведен је reset порт, који би било добро активирати на почетку рада кола. Овај порт није наведен у условима задатка. У овом случају, ресет функционалност би могла да се постигне и негативним импулсом на WE.

  ⓘ Када је (у задацима) интерфејс компонената прецизно наведен, не треба уводити нове, ненаведене портове (попут ресета у овом примеру), већ у тестбенчу треба демонстрирати како се (нпр. ресет) те функционалности могу постићи побудом.



Слика 8. Компоненте система

Синтеза

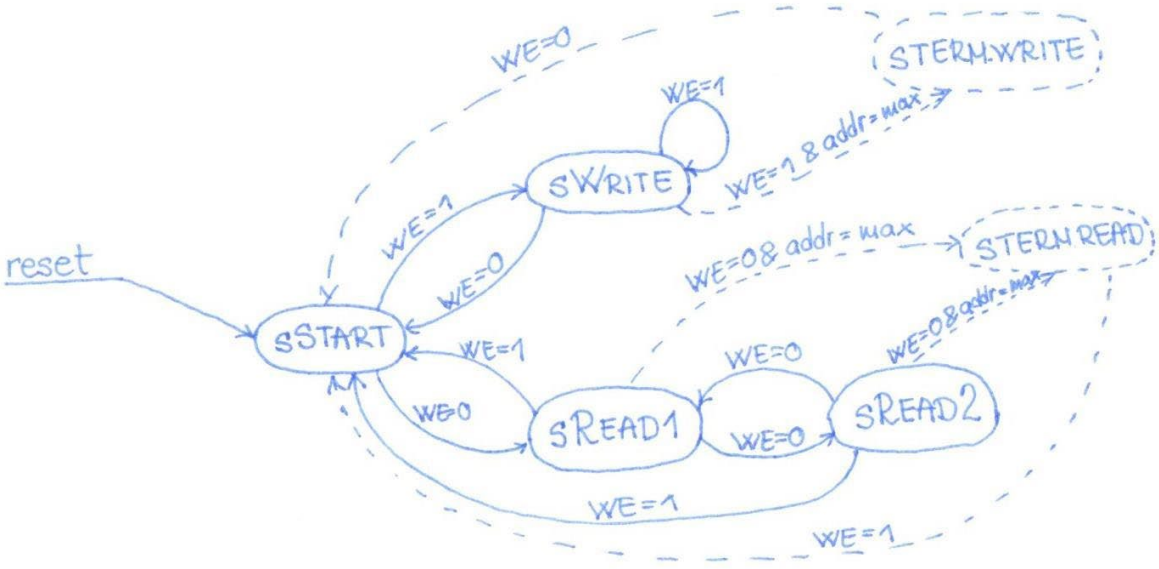
За појединачне компоненте кола можемо искористити описе раније дате у овом материјалу:

- ∉ сабирач: ентитет `carry_ripple_adder` из ранијих примера
- ∉ регистри: ентитет `register_tristate` из ранијих примера
- ∉ меморија: ентитет `Memorija` из ранијихпримера
- ∉ бројач: ентитет `counter8` из ранијих примера

Управљачка једница се може моделовати коначним аутоматом. Иако се функција управљачке јединице може пројектовати и на друге начине, моделовање коначним аутоматом омогућава једноставнију измену и проширивање функционалности; комплексност моделовања коначним аутоматом спорије расте са порастом сложености кола, него када се свака функција моделује директно.

Управљачка јединица

Дијаграм стања управљачке јединице је дат на слици 9, неиспрекидане линије. Ради прегледности, са слике су изостављени излази аутомата.
Сигнал reset уводи аутомат у почетно стање, у коме се ресетује бројач адреса. И из свих осталих стања reset преводи аутомат у почетно стање (није означено на слици).
Аутомат на сваки такт одређује своје наредно стање.
Уколико је WE=1, прелази (и остаје) у стању sWrite, у коме не генерише никакво управљање.
Када дође WE=0, аутомат прелази у почетно стање (где се ресетује бројач адреса), а затим (на следећи такт) у стања читања, где алтернативно мења стања sRead1 и sRead2, у сваком од њих алтернативно активира упис у по један од регистара BUF1 и BUF2.
WE=1 у сваком тренутку враћа аутомат у почетно стање из стања читања.



Слика 9. Дијаграм стања коначног аутомата који имплементира функцију CU

У наставку је пример употребе кола, са прегледом сигнала у сваком тактном интервалу:

```
---------------------------------
|00 | WE    = -      |           | -- Reset celog kola
|   | reset = 1      |           | --
|   | din   = -      |           | --
|   |                | dout= U   | --
---------------------------------
|01 | WE    = 1      |           | -- Na adresu 0(koja se postavlja iz
brojaca)
|   | reset = 0      |           | -- upisuje se podatak 00H u memoriju
|   | din   = 00H    |           | -- mem[0] = 00H
|   |                | dout= U   | --
---------------------------------
|02 | WE    = 1      |           | -- Brojac nastavlja da broji, sada je na 1
|   | reset = 0      |           | -- Na adresu 1 se upisuje 05H u memoriju
|   | din   = 05H    |           | -- mem[1] = 05H
|   |                | dout= U   | --
---------------------------------
|03 | WE    = 1      |           | -- Brojac nastavlja da broji, sada je na 2
|   | reset = 0      |           | -- Na adresu 2 se upisuje A0H u memoriju
|   | din   = 0A0H   |           | -- mem[2] = 0A0H
|   |                | dout= U   | --
---------------------------------
|04 | WE    = 1      |           | -- Brojac nastavlja da broji, sada je na 3
|   | reset = 0      |           | -- Na adresu 3 se upisuje A5H u memoriju
|   | din   = 0A5H   |           | -- mem[3] = 0A5H
|   |                | dout= U   | --
---------------------------------
|05 | WE    = 0      |           | -- Promena na WE vraća automat u sStart
|   | reset = 0      |           | -- Brojac ce ponovo brojati od 0.
|   | din   = -      |           | --
|   |                | dout= U   | --
---------------------------------
|06 | WE    = 0      |           | -- Brojač se resetuje, Automat = Read1
|   | reset = 0      |           | -- OP1 = mem[0]
|   | din   = -      |           | -- OP1 = 00H
|   |                | dout= U   | --
---------------------------------
|07 | WE    = 0      |           | -- Automat = Read2
|   | reset = 0      |           | -- OP2 = mem[1]
|   | din   = -      |           | -- OP2 = 05H
|   |                | dout= U   | -- Zapaziti da sada sl stanje = Read1
---------------------------------
|08 | WE    = 0      |           | -- Automat = Read1, OP1 = mem[2]
|   | reset = 0      |           | -- OP1 = A0H
|   | din   = -      |           | -- Izlaz je sada smislen
|   |                | dout= 05H | -- Moze se koristiti
---------------------------------
|09 | WE    = 0      |           | -- Automat = Read2
|   | reset = 0      |           | -- OP2 = mem[2]
|   | din   = -      |           | -- OP2 = A5H
|   |                | dout= 0A5H| -- Zapaziti izlaz!
---------------------------------
|10 | WE    = 1      |           | -- Stanje= sStart
|   | reset = 0      |           | -- zapaziti izlaz!
|   | din   = -      |           | --
```

```
|    |                | dout= 145H| --
---------------------------------
|11  | WE     = 1     |           | -- Mozemo zapoceti nov ciklus upisivanja
|    | reset  = 0     |           | -- mem[0]=05H
|    | din    = 05h   |           | -- Zato sto su 3statebaferi, izlaz postoji
|    |                | dout= 145H| --
---------------------------------
```

Пројектовање коначних аутомата

Коначни аутомати се на VHDL-у могу пројектовати на више начина. Овде ће бити приказан један од стандардизованих модела коначног аутомата помоћу два процеса. На овај начин, дијаграм стања се директно мапира на case структуру, и није сложеније имплементирати ни врло сложене дијаграме.

За кодирање стања користи се тип набрајања и два сигнала овог типа: у првом се памти тренутно стање аутомата, а у другом се израчунава стање у које треба прећи у следећем такту.

Дизајн коначног аутомата се састоји од два процеса.

Први процес је секвенцијална мрежа, осетљив је само на такт и ресет, и у њему се поставља иницијално стање и дешава транзиција наредог стања у садашње:

```
state_change: process (reset, clk)
    current_state <= next_state;
    --+ initial_state
    --+ синхрони излази, ако их има
```

Други процес је комбинациона мрежа, у њему се израчунава следеће стање и генеришу излази. Осетљив је на све улазе и сигнал који чува тренутно стање:

```
calculate_process(svi ulazi i current state)
    --računanje sledećeg stanja, generisanje асинхроних izlaza
```

Реализација управљачке јединице на описан начин дата је у опису иза, у линијама 005-080. Реализација целог кола приказана је у наставку.

Z ПРИМЕР Реализација система са меморијом и аритметиком.
Уведено: пројектовање сложенијег кола; пројектовање коначног аутомата моделом два процеса.
(Захтеви и опис пројектовања дати су раније у тексту.)

```
000 --Upravljacka logika
001 LIBRARY IEEE;
002 USE IEEE.STD_LOGIC_1164.ALL;
003 USE ieee.std_logic_arith.ALL;
004
005 ENTITY Kontrola IS
006     PORT ( clk, WE, reset : IN STD_LOGIC;
007         addr : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
008         rstAddr: OUT std_logic;
009         upis1 : OUT STD_LOGIC;
010         upis2 : OUT STD_LOGIC);
011 END Kontrola;
012
013 ARCHITECTURE stateMachine OF Kontrola IS
014     TYPE states IS (sWrite, sRead1, sRead2, sStart);
```

```
015     -- i, za dodatak: sTermWrite, sTermRead
016     SIGNAL current_state, next_state: states;
017     SIGNAL upis1_async, upis2_async: STD_LOGIC;
018 BEGIN
019     --
020     -- Promena stanja
021     --
022     stateChange: PROCESS (reset, clk) -- samo clk i asinhroni ulazi
023     BEGIN
024         IF (reset='1') THEN
025             current_state <= sStart;
026         ELSIF (clk'EVENT AND clk='1') THEN
027             upis1<=upis1_async;
028             upis2<=upis2_async;
029             current_state <= next_state;
030         END IF;
031     END PROCESS;
032 
033     -- Izračunavanje narednog stanja i izlaza
034     -- current_state i svi ostali ulazi.
035     -- Ovde addr nije neophodan,
036     -- ali ako bi se generisanje adrese
037     -- prebacilo iz brojača u kontrolu,
038     -- onda sa addr ovde automat biva
039     -- svestan da se menja adresa i
040     -- da treba da je promeni ponovo.
041     --
042     OutputAndStateCalculation: PROCESS (WE, addr, current_state)
043     BEGIN
044         CASE current_state IS
045             WHEN sStart =>
046                 rstAddr<='1';--resetuje brojac adresa
047                 upis1_async<='0';
048                 upis2_async<='0';
049                 IF (WE='1') THEN
050                     next_state <= sWrite;
051                 ELSIF (WE='0') THEN
052                     next_state <= sRead1;
053                 END IF;
054             WHEN sWrite =>
055                 rstAddr<='0';
056                 IF (WE='1') THEN
057                     next_state <= sWrite;
058                 ELSIF (WE='0') THEN
059                     next_state <= sStart;
060                 END IF;
061             WHEN sRead1 =>
062                 rstAddr<='0';
063                 upis1_async <= '1';
064                 upis2_async <='0';
065                 IF (WE='0') THEN
066                     next_state <= sRead2;
067                 ELSIF (WE='1') THEN
068                     next_state <= sStart;
069                 END IF;
070             WHEN sRead2 =>
```

```
071                 upis1_async <= '0';
072                 upis2_async <='1';
073                 IF (WE='0') THEN
074                     next_state <= sRead1;
075                 ELSIF (WE='1') THEN
076                     next_state <= sStart;
077                 END IF;
078         END CASE;
079     END PROCESS;
080 END ARCHITECTURE stateMachine;
081 
082 ------------------------------------------------
083 ------------------------------------------------
084 ------------------------------------------------
085 ------------------------------------------------
086 ------------------------------------------------
087 
088 -- top level
089 LIBRARY IEEE;
090 USE IEEE.STD_LOGIC_1164.ALL;
091 USE ieee.numeric_std.ALL; --zbog to_unsigned
092 
093 ENTITY sistem IS
094     PORT( clk, WE, reset : IN std_logic;
095         din : IN std_logic_vector(7 DOWNTO 0);
096         dout : OUT STD_LOGIC_VECTOR(8 DOWNTO 0)
097         -- 9b, da hvata i izlazni prenos (jer smo tako u mogucnosti)
098     );
099 END ENTITY sistem;
100 
101 ARCHITECTURE structural OF sistem IS
102     SIGNAL addr, data_int, operand1, operand2 : STD_LOGIC_VECTOR (7
DOWNTO 0);
103     SIGNAL addrInt: integer RANGE 0 TO 255; -- izlaz iz brojača
104     SIGNAL rstAddr, upis1, upis2 : STD_LOGIC;
105 
106 BEGIN
107     --
108     -- Kontrola
109     --
110     cu: ENTITY work.Kontrola(stateMachine)
111         PORT MAP(
112             clk=>clk,
113             WE=>WE,
114             reset=>reset,
115             addr => addr,
116             rstAddr=>rstAddr,
117             upis1=> upis1,
118             upis2 => upis2
119         );
120 
121     --
122     -- brojac iz ranijih primera
123     --
124     cnt: ENTITY work.counter8(counter8_arch)
125         PORT MAP(
```

```
126             clk=>clk,
127             reset=>rstAddr,
128             ce=>'1',
129             -- ne koristimo ce, uvek je aktivno, uvek broji navise
130             load=>'0',
131             dir=>'1',
132             din=> 0,
133             -- nije od uticaja, ne koristi se jer je load uvek neaktivno
134             count=>addrInt);
135 
136     --
137     -- izlaz brojaca je integer, konvertujemo ga u unsigned
138     -- pa u _vector, jer je tog tipa adresni ulaz memorije
139     --
140     addr <= std_logic_vector(to_unsigned(addrInt, 8));
141 
142     --
143     -- memorija iz ranijih primera
144     --
145     mem: ENTITY work.Memorija(Behavioral)
146         PORT MAP(
147             WE =>WE,
148             clk => clk,
149             addr => addr,
150             data => din,
151             Q =>data_int
152         );
153 
154     --
155     -- registar iz ranijih primera
156     --
157     buff1: ENTITY work.register_tristate(cell_level)
158         GENERIC MAP(width=>8)
159         PORT MAP(
160             clock=>upis1,
161             out_enable=>'1',
162             --ne koristimo ovaj feature, zato uvek aktivno
163             data_in=>data_int,
164             data_out=>operand1
165         );
166 
167     --
168     -- registar iz ranijih primera
169     --
170     buff2: ENTITY work.register_tristate(cell_level)
171         GENERIC MAP(width=>8)
172         PORT MAP(
173             clock=>upis2,
174             out_enable=>'1',
175             data_in=>data_int,
176             data_out=>operand2
177         );
178 
179     --
180     -- sabirac iz ranijih primera
181     --
```

```
182     sabirac: ENTITY work.carry_ripple_adder(w_generate)
183         GENERIC MAP(n=>8)
184         PORT MAP(
185             a=>operand1,
186             b=>operand2,
187             cin=>'0',
188             s=> dout(7 DOWNTO 0),
189             cout=> dout(8)
190         );
191 END ARCHITECTURE structural;
```

Условима задатка није прецизирано:

- Капацитет сваке меморије је ограничен. Шта треба да се деси када је WE предуго на истом нивоу: шта када упис траје толико да се напуни меморија, а шта када читање траје толико да се саберу сви подаци из меморије? (Како се описано коло понаша у тим случајевима?)
- Шта треба да се дешава са претходно памћеним подацима након новог уписивања, да ли се бришу или треба да остану у меморији. (Како је у овој реализацији?)

Дискусија

- Шта би требало променити да се не дозволи упис у пуну меморију ни даље читање када се прочита цела меморија? У смеру решења могу да послуже додатна стања, приказана на слици 9 испрекиданим линијама, у којима би се искључивала дозвола бројања у терминалним случајевима, и адреса би морала да се доведе у аутомат (приказано испрекиданом линијом на слици 8).
- Како би се генерисање адресе укључило у CU?
- Како би било могуће једним управљачким сигналом управљати са оба регистра?
- Како би се пројектовала меморија која би издавала податке на обе ивице такта?

За размишљање:

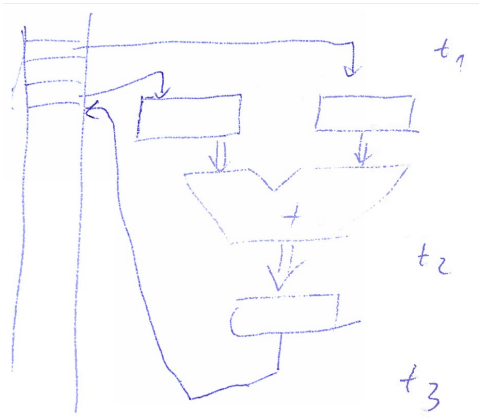
- Да ли видите проточност у овом примеру?
- Да ли видите да би овај хардвер брже сабирао бројеве од Фон Нојманове архитектуре која је заступљена у рачунарима?
- Употреба специфичног хардвера постоји у свим система у којима је брзина неопходна (нпр. ГПУ, претпроцесирање података, обрада велике количине података...)
- Описано коло приказује збирове података са суседних адреса ([0]+[1], [1]+[2], [2]+[3]...). Шта би требало да се промени да се приказују збирови суседних података, али без понављања ([0]+[1], [2]+[3], [4]+[5])? Шта би додатно требало урадити да се ови збирови појављују на сваки такт?

Принципи дигиталног дизајна

На крају овог дела курса, важно је истаћи да просто познавање синтаксе VHDL-а, чак ни познавање синтетизабилног подскупа језика, не гарантује успех при пројектовању.
Постоје бројни проблеми који проитичу из природе хардвера, о којима се мора водити рачуна при пројектовању.
Синтаксно исправне и синтетизабилне конструкције, при имплементацији у хардверу могу да дају непредвиђене или нестабилне ефекте. Да би се то избегло, при пројектовању се мора водити рачуна о принципима дигиталног дизајна.
Упознавање са свим принципима дигиталног дизајна би захтевало цео засебан курс; на овом месту ће дати само два илустративна примера.

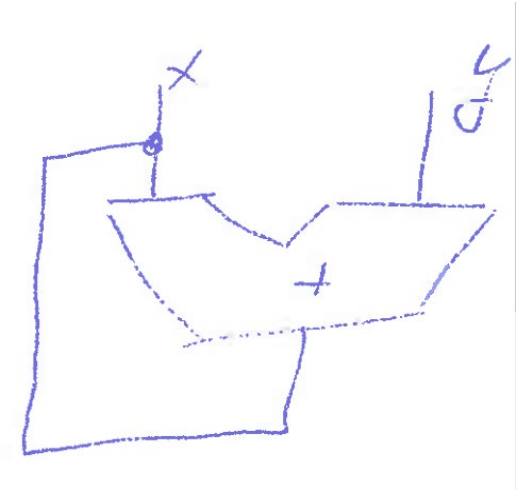
Combinatorial feedback

У програмском језику, нпр C++, савршено је у реду написати:
x=x+y;
јер ова наредба производи да се на улазне бафере сабирача у ALU процесора доведу x и y, па да се резултат из излазног бафера сабирача одведе у меморију на адресу x (слика 10).



Слика 10. Ефекат наредбе x=x+y у неком програмском језику

Конструкција X<=X+Y; у VHDL-у за ефекат има да се краткоспаја излаз сабирача на један од његових улаза (слика 11). Како је сабирач комбинациона мрежа са реалним пропагационим кашњењем, ова клаузула ће произвести нестабилно стање. Да ствар буде компликованија, X и Y су могли због неких других клаузула да буду имплементирани имплицитним меморијским елементима, у ком случају неће бити проблема, али се то из ове предметне клаузуле никако не може видети.
Из тог разлога, сигурније је експлицитно уградити регистре са тактом и контролом уписа. Употреба променљивих уместо сигнала у овом случају такође може да помогне, али о случајевима када се за променљиву генерише меморијски елемент није било детаљно речи у овом материјалу.



Слика 11. Ефекат VHDL клаузуле X<=X+Y;

Активне ивице такта

Кола се могу пројектовати тако да све синхроне компоненте реагују на узлазну **ИЛИ** на силазну ивицу такта. Другим речима, није препоручљиво један сигнал мењати на обе ивице такта (експлицитно, или на сваку промену тактног сигнала - што се своди на обе ивице)!
Иако је могуће пројектовати таква кола - нпр. многи модерни процесори имају артметичко-логичке јединице велике пропусности које израчунавају по један резултат у свакој полуперіоди такта, алати за синтезу VHDL описа на већини хардверских платформи не могу да креирају такав хардвер. Алати за синтезу који за циљну платформу имају програмабилне хардверске платформе углавном и не покушавају синтезу кола која се окидају на обе ивице.

Из овог разлога, меморија која би издавала податке на обе ивице такта из последње тачке дискусије напред, не би могла да се синтетизује на већини платформи.