Универзитет у Нишу

Електронски факултет

Катедра за рачунарство

Архитектура и организација рачунара

Вежбе, VHDL

Termin 2

# Инстанцирање компонената

У VHDL-у могуће је у оквиру описа једног кола инстанцирати ентитете који су већ дефинисани у другим ентитетима. На овај начин, кола се описују својом структуром - компонентама које их чине и њиховим међусобним везама (тзв. структурални опис).

Клаузула којом се инстанцира примерак ентитета спада у конкурентне клаузуле, и треба да се наведе унутар тела архитектуре.

|  |
| --- |
| 1 <име инстанце>: **entity** <библиотека>.<име ентитета>(<име арихтектуре>)  2 [**generic map**(<predefinisanje vrednosti generic konstanti>)]  3 **port map**(<mapiranje portova komponente na signale iz okruženja>); |
| <име инстанце> - јединствени идентификатор у оквиру архитектуре  <библиотека> - име библиотеке у којој је дефинисан ентитет који се инстанцира (подразумевана је библиотека work, која представља библиотеку текућег пројекта).  <име ентитета>(<име арихтектуре>) - ентитет и архитектура компонeнте која се инстанцира.  Мапирање *generic* константи и портова се може обављати именованим и позиционим мапирањем, што ће бити показано [у наредном примеру](#Primer4). |

ⓘ Уколико је наведен тзв. прототип компоненте у **component** клаузули, компонента се може инстанцирати синтаксом: <име инстанце>: <име компоненте> [**generic map**(<predefinisanje vrednosti generic konstanti>)] **port map**(<mapiranje portova komponente na signale iz okruženja>); У том случају се пре превођења мора навести и **configuration** клаузула којом се компоненте мапирају на ентитете. Овај начин инстанцирања компонената, мада флексибилнији, је синтаксно сложенији и биће избегнут у овом курсу.

|  |
| --- |
| **Z** ПРИМЕР 4, Tробитни сабирач, коришћењем једнобитних сабирача из претходгог примера  Уводе се: инстанцирање компонената, именовано и позиционо мапирање портова, интерни сигнали, издвајање појединачних битова у вишебитном сигналу.  Потребно је креирати опис тробитног потпуног сабирача, коришћењем већ постојећег описа једнобитног потпуног сабирача. Портови и структура кола су дати на [слици 6](#kix.mjrcee5sz9lx).    *Слика 6. Тробитни потпуни сабирач састављен од једнобитних сабирача*  d |
| 01 **ENTITY** adder3b **IS**  02 **PORT** **(**op1**,** op2**:** **IN** bit\_vector**(**2 **DOWNTO** 0**);**  03 cin**:** **IN** bit**;**  04 sum**:** **OUT** bit\_vector**(**2 **DOWNTO** 0**);**  05 cout**:** **OUT** bit**);**  06 **END** **ENTITY** adder3b**;**  07  08 **ARCHITECTURE** struct **OF** adder3b **IS**  09 **SIGNAL** c01**,** c12**:** bit**;**  10 **BEGIN**  11 bit0**:** **ENTITY** work**.**full\_adder**(**truth\_table**)**  12 **PORT** **MAP(**a**=>**op1**(**0**),** b**=>**op2**(**0**),** c\_in**=>**cin**,** s**=>**sum**(**0**),**c\_out**=>**c01**);**  13 -- именовано мапирање портовa (<port\_komponente>=><signal na koji se vezuje port>, …)  14 --op1(0) - bit 0 signala op1  15 bit1**:** **ENTITY** work**.**full\_adder**(**truth\_table**)**  16 **PORT** **MAP(**op1**(**1**),** op2**(**1**),** c01**,** sum**(**1**),**c12**);**  17 -- poziciono мапирање, u redosledu iz entity/port deklaracije komponent,  18  19 bit2**:** **ENTITY** work**.**full\_adder**(**truth\_table**)**  20 **PORT** **MAP(**op1**(**2**),** op2**(**2**),** c12**,** sum**(**2**),**cout**);**  21 **END** **ARCHITECTURE** struct**;** |
| л. 09: интерни сигнали - нису портови (не излазе изван сабирача), већ представљају интерне физичке везе, у овом случају за повезивање портова инстанцираних компонената.  л. 12: синтакса за именовано мапирање: <порт компоненте>**=>**<сигнал на који се повезује порт>. Редослед портова није важан.  л. 16: позиционо мапирање, наводе се сигнали који се повезују на портове компоненте, у тачном редоследу портова по дефиницији компоненте.  ⓘ Не морају се декларисати интерни сигнали за све везе ка портовима инстанцираних компонената. Mогу се мапирати портови компоненте директно на портове ентитета, не морају се користити интерни сигнали. У овом примеру, креирани су интерни сигнали само за међусобно повезивање компонената, тј. за везе које не иду на портове ентитета adder3b. |

# Процеси

Процеси спадају у конкурентне клаузуле које се могу наћи у телу архитектуре. Процеси енкапсулирају секвенцијалне клаузуле, док је сам процес као клаузула конкурентан. Погледајмо изглед процеса:

|  |
| --- |
| 01 **process** **(**signal\_1**,** signal\_2**,** **...)** **is**  02 deklaracija\_1  03 deklaracija\_2  04 **...**  05 **begin**  06 klauzula\_1  07 klauzula\_2  08 **...**  09 **end** **process** |
| л. 01: Сигнали у загради (л. 01) представљају *sensitivity* листу (описано [касније](#SensitivityLista)).  л. 02-04: Локалне декларације, могу да садрже декларације константи, сигнала и променљивих. |

Са аспекта симулације, процес се може посматрати као бесконачна петља: клаузуле из тела процеса се извршавају редом како су написане (тело процеса се састоји од секвенцијалних клаузула), до краја тела процеса, а затим се поново прелази на почетак тела процеса. Овај процес се понавља непрекидно, уколико нису задате **wait** клаузуле или *sensitivity* листа, у ком случају се процес суспендује, док се не створе поново услови за његово поновно покретање. Уколико има више дефинисаних процеса, на почетку симулације сви процеси се покрећу истовремено, а након њиховог (евентуалног) суспендовања сваки се покреће када су испуњени његови услови за покретање.

**Wait** клаузула представља секвенцијалну клаузулу која се може наћи унутар тела процеса. При наиласку на **wait** клаузулу , процес се суспендује, а поново се покреће када су испуњени услови наведени у клаузули. Изглед wаit клаузуле:

**wait** **[on** lista\_signala**]** **[until** uslov**]** **[for** vreme**];**

Сви делови **wait** наредбе су опциони, тако да се може написати и само **wait;** (у ком случају се процес трајно суспендује; корисно нпр. да се напише процес који ће извршити неку иницијализацију на почетку а након тога неће радити ништа више).

Услови који покрећу поново процес у **wait** клаузули су као у примерима:

|  |
| --- |
| **WAIT** **ON** in1, in2**; - чека на промену на било ком од наведених сигнала**  **WAIT FOR** 100 ns**; - чека 100 ns**  **WAIT UNTIL** a=0; - **чека док СЛЕДЕЋИ ПУТ не постане испуњен услов** (чека на промену на било ком од наведених сигнала, која ће резултовати испуњење услова) |
| **wait** **for** није синтетизабилан, али налази примену у тестирању (у тзв. тестбенчевима)  **wait** **until** није увек синтетизабилан; најбезбедније је користити **wait** **on**. |

*SENSITIVITY* ЛИСТА (заграда са листом сигнала у заглављу процеса: **process** **(**signal\_1**,** signal\_2**,** **...)** има исти ефекат као једна **wait** **on** клаузула на крају тела процеса:

|  |  |
| --- | --- |
| 1 **process** **(**a**,**b**)** **begin**  2 y **<=** a **and** b**;**  3 **end** **process;** | 1 **process** **begin**  2 y **<=** a **and** b**;**  3 **wait** **on** a**,**b**;**  4 **end** **process** |

Ако процес има sensitivity листу, у телу процеса не сме да постоји **wait** клаузула.

|  |
| --- |
| **Z** ПРИМЕР 5, генератор клока  Уводе се: процес, wait, задавање таласног облика помоћу after.  Потребно је креирати процес који ће генерисати сигнал који има облик тактног сигнала са полупериодом која је смештена у константи T\_pw, типа time. |
| **01 clock\_gen: PROCESS IS -- labela je opciona**  **02 ---------------------------------------------------**  **03 -- 1. način**  **04 BEGIN**  **05 clk1<='1';**  **06 wait for T\_pw;**  **07 clk1<='0';**  **08 wait for T\_pw;**  **09 END PROCESS clock\_gen;**  **10 ---------------------------------------------------**  **11 --2. način:**  **12 clock\_gen: PROCESS IS**  **13 BEGIN**  **14 clk2<='1' AFTER T\_pw, '0' AFTER 2\*T\_pw;**  **15 WAIT FOR 2\*T\_pw;**  **16 END PROCESS clock\_gen;**  **17 ---------------------------------------------------**  **18 -- 3. Nacin**  **19 clock\_gen: PROCESS IS**  **20 BEGIN**  **21 clk3<='1' AFTER T\_pw, '0' AFTER 2\*T\_pw;**  **22 WAIT UNTIL clk3='0'; -- WAIT UNTIL - čeka na promenu na signalu clk i da je uslov ispunjen**  **23 END PROCESS clock\_gen;**  **24 ---------------------------------------------------** |
| Употреба сигнала и променљивих типа time није синтентизабилна. Генератор клока се користи за симулацију и тестирање других кола.  1. начин : сигналу се додели ниво 1 (л. **05**), чека се једну полупериоду (л. **06**), па се додели ниво 0, и поново чека полупериоду, након чега процес креће од почетка.  2 . начин: сигналу се задаје таласни облик - вредност и време када ту вредност треба доделити након тренутка задавања.  3. начин : л.**21**: сигнал ће добити вредност 1 након једне полупериоде а 0 након једне периоде; **wait** у л.**22** чека да clk постане 0, што ће се десити због друге доделе у л. **21** након једне периоде, и тада ће процес наставити од почетка - сигнал добија 1 једну полупериоду након тога и тд.  Напомена : по описима на други и трећи начин, сигнал clk неће имати дефинисану вредност једну полупериоду на почетку.  Таласни облици за T\_pw = 10 (било којих јединица) за све три варијанте су следећи: |

THINK HARDWARE!

Процесима могу да се моделују комбинациона и секвенцијална кола. Уколико желимо чисто комбинационо коло, сви сигнали који се у процесу читају (нпр. налазе с десне стране оператора доделе), морају се навести у *sensitivity* листи - како би се изрази који их садрже израчунавали поново сваки пут кад се било који од тих сигнала промени. Ако процес моделује секвенцијално коло, само клок и асинхрони контролни улази (нпр ресет, уколико треба да је асинхрони) треба да се налазе у *sensitivity* листи.

# Модели кашњења

Секвенцијална додела вредности сигналу (она која се налази унутар тела процеса) се не понаша на очекиван начин - начин на који смо навикли у програмским језицима)! Говори се о *моделима кашњења*.

Транспортно кашњење - вредност се додељује са закашњењем у односу на тренутак када је задата додела. Примењује се коришћењем кључне речи **AFTER**. Пример је клок генератор из [примера 5](#Primer5), 2. i 3. начин.

ⓘ Иначе, коришћењем иницијализације при декларацији сигнала, клок генератор може и компактније да се напише, конкурентном клаузулом доделе вредности сигналу (без процеса, унутар архитектуре):

У декларативном делу архитектуре: **SIGNAL** clk **:** std\_logic **:=** ‘0’**;** -- напомена: иницијализација при декларацији није синтетизабилна!

U telu arhitekture: clk **<=** not clk **AFTER** T\_pw**;**

Постоји и инерцијално кашњење, којим се прецизније описује горња гранична фреквенца кола, али се овај модел кашњења неће обрађивати у овом курсу.

Делта кашњење - примењује се увек. **Вредности се додељују сигналима када је процес суспендован.** Док се процес елаборира (“извршава” - у домену симулације), планирају се доделе вредности (планирају се "трансакције"). Када се процес суспендује, све планиране трансакције се примењују истовремено. Уколико је за један сигнал било планирано више трансакција у једној секвенци клаузула (више сукцесивних додела једном сигналу), биће извршена само последња. Уколико постоји више процеса, трансакције се примењују када су суспендовани сви процеси.

ⓘ Уколико су из више конкурентних клаузула (више процеса нпр) планиране трансакције над истим сигналом, биће покушано да се примене све, са потенцијалном опасношћу да се деси “конфликт” - нпр. да је на једном месту покушана да се постави јединица а на другом месту нула. Резултујућа вредност у реалном хардверу ће зависити од струје сваког од постављача (“драјвера”) тог сигнала сигнала и није лако утврдити је унапред. Тип std\_logic успешно моделује овакве ситуације вредношћу **X** на сигналу. Уколико сви драјвери покушавају да поставе исту вредност, конфликта неће бити.

|  |
| --- |
| **Z PRIMER 6. Делта кашњење** |
| **01 PROCESS (B,D) IS**  **02 BEGIN**  **03 A<=B;**  **04 C<=D;**  **05 -- сигнали А и C добијају вредности истовремено, након једног извршавања тела процеса.**  **06 -- Испада да није важан редослед,иако су ово секвенцијалне клаузуле!**  **07 -- Али! Ако би се између нашао wаit, то би суспендовало процес, па би редослед био важан.**  **08 END PROCESS;**  **09 ------------------------------------**  **10 --АЛИ:**  **11 PROCESS (A,B) IS**  **12 BEGIN**  **13 A<=B;**  **14 C<=A;**  **15 -- C добија претходну вредност А, не тренутну вредност B!!!**  **16 -- истовремено су доделе, кад се процес суспендује.**  **17 +** |
| **За размишљање:** како би се понашао процес који у телу процеса има секвенцу:  **a<=b;b<=a;**  ? |

Клок генератор се може написати и коришћењем ф-ја “rising\_edge” , односно “falling\_edge”, инфо на [.](https://stackoverflow.com/questions/15205202/clkevent-vs-rising-edge) https://stackoverflow.com/questions/15205202/clkevent-vs-rising-edge

## Променљиве (и константе)

Осим сигнала, постоје и **променљиве** и **константе**.

Декларација променљвиих и константи је обавезна, и синтакса је:

**constant** *ime*, *ime*, ... : *tip* [:= *izraz*;

**variable** *ime*, *ime*, ... : *tip* := *izraz*;

Декларације се могу наћи у декларационом делу архитектуре или процеса.

Рестрикција: променљива се декларише тако да је видљива само једном процесу. Значи, ако архитектура има више процеса, декларациони део архитектуре не сме да садржи декларације променљивих! Једноставније је придржавати се практичног правила: декларације променљивих само унутар процеса. Променљива је видљива где је дефинисана, и у свим елементима ниже по хијерархији (нпр. ако је дефинисана у архитектури, видљива је и у процесу унутар те архитектуре).

ⓘ Могући су случајеви да се имена променљивих преклапају, и онда се навођењем извора променљиве разрешавају конфликти. Нпр. ако је у архитектури glavna декларисана променљива N, и у тој архитектури процес у коме је такође декларисана променљива N, онда, да би се у телу процеса приступило променљивој N из архитектуре, треба да се напише glavna.N.

**Оператор доделе** вредности променљивој је различит од оператора доделе сигналу. Подсетимо се, оператор доделе сигналу је <**=**, док је оператор доделе променљивој **:=**.

Основна разлика између променљивих и сигнала је у моделу кашњења: када се променљивој додели вредност, она одмах узима ту вредност, за разлику од сигнала, којима се вредност додељује када су процеси суспендовани (видети Моделе кашњења).

Мада се не може тврдити да важи у свим случајевима, коришћење променљивих врло често узрокује креирање меморијских елемената.

|  |
| --- |
| **Z ПРИМЕР 7.** Променљиве и сигнали, додела вредности |
| 01 **SIGNAL** x**,** y**,** WR**:** bit**;**  02 -- ...  03 p1**:** **PROCESS** **(**WR**)**  04 **BEGIN**  05 x **<=** '1'**;**  06 y **<=** x**;**  07 -- у првом пролазу, y добија претходну вредност x, не 1!  08 -- тек на следећу промену сигнала WР ће се 1 из x преписати у y  09 **END** **PROCESS;**  10  11 p2**:** **PROCESS** **(**WR**)**  12 **VARIABLE** var **:** bit**;**  13 **BEGIN**  14 var **:=** '1'**;**  15 x **<=** var**;**  16 y **<=** var**;**  17 -- x и y постају 1 истовремено  18 **END** **PROCESS;** |

# Testbench

Тестбенчем се назива ентитет који се креира само за тестирање другог ентитета. Нема портове, садржи инстанцу компоненте која се тестира (тзв. ***U****nit* ***U****nder* ***T****est - UUT*), сигнале који се повезују на портове UUT-а , и процес којим се дефинишу таласни облици побуде (*stimulusi*).

ⓘ Многа развојна окружења садрже команде за (полу)аутоматско креирање тестбенча. У том случају дизајнеру преостаје само да дефинише таласне облике побуде.

|  |
| --- |
| **Z** ПРИМЕР 8, Тестбенч за тробитни сабирач из ранијег примера  Уводи се: тестбенч.  Треба дефинисати тестбенч за тестирање тробитног сабирача карактеристичним вредностима. |
| 01 **ENTITY** adder3b\_tb **IS**  02 **END** **ENTITY** adder3b\_tb**;**  03 **ARCHITECTURE** tb **OF** adder3b\_tb **IS**  04 **SIGNAL** sigA**,**sigB**,**sigC **:** bit\_vector**(**2 **DOWNTO** 0**);**  05 **SIGNAL** c\_in**,** c\_out **:** bit;  06 **BEGIN**  07 uut**:** **ENTITY**  **work.**adder3b**(**struct)  08 **PORT** **MAP(**  09 op1**=>**sigA**,**  10 op2**=>**sigB**,**  11 cin**=>**c\_in**,**  12 sum**=>**sigC**,**  13 cout**=>**c\_out  14 **);**  15 stimuli**:** **PROCESS**  16 **BEGIN**  17 sigA**<=**"001"**;**  18 sigB**<=**"010"**;**  19 c\_in**<=**'0'**;**  20 **WAIT** **FOR** 1 ns**;**  21 sigA**<=**"111"**;**  22 sigB**<=**"010"**;**  23 c\_in**<=**'0'**;**  24 **WAIT** **FOR** 1 ns**;**  25 sigA**<=**"111"**;**  26 sigB**<=**"010"**;**  27 c\_in**<=**'1'**;**  28 **WAIT** **FOR** 1 ns**;**  29 --...  30 **END** **PROCESS** stimuli**;**  31 **END** **ARCHITECTURE** tb**;** |
| л. 04,05: дефинишу се сигнали на које ће се повезати портови компоненте која се тестира.  л. 07: инстанцира се компонента која се тестира; портови инстанциране компоненте се повезују на интерне сигнале тестбенча  л. 15: креира се процес којим се задају таласни облици сигнала везаних на улазне портове компоненте.  Симулира се тестбенч као *top-level* компонента , и посматрају се таласни облици оних сигнала који су декларисани у тестбенчу.  ⓘ Тест бенч може да садржи и **assert-report-severity** клаузуле којима се скреће пажња дизајнеру на неочекиване случајеве и олакшава тестирање, али ово препуштамо индивидуалном истраживању. |

ⓘПонашање секвенцијалних делова описа се може брзо истестирати уметањем клаузуле **report** у код компоненте која се развија. У току симулације се приказују излази у прозору конзоле симулатора. Синтакса:

REPORT “komentar” & tip’IMAGE(neki\_signal); -- тип заменити с типом сигнала neki\_signal. Нпр. bit ili integer…

# Атрибути

Објекти у VHDL-у имају своје **атрибуте**. Атрибути представљају механизам добијања информација о објекту (најчешће ћемо испитивати вредности атрибута сигнала). Синтакса за писање атрибута изгледа:

ime\_objekta'**atribut**

Атрибути могу бити уграђени или кориснички дефинисани, али ћемо се у овом курсу задржати на уграђеним атрибутима, и то врло малом подскупу свих постојећих атрибута. На објекту се могу применити само атрибути примерени за врсту и тип објекта. Тако нпр. атрибут **range** се може применити над низом да би се добио опсег у коме се крећу индекси низа, док на сигналу типа bit не би имао смисла.

У овом тренутку нам је од интереса атрибут сигнала **EVENT**. Враћа логичку вредност true уколико је сигнал променио вредност у прошлом периоду суспензије процеса (термин је: уколико се десио догађај на сигналу). Уколико сигнал није мењао вредност, атрибут враћа false. Напомена: важно је разликовати трансакцију од догађаја. Трансакција значи да се планира додела вредности неком сигналу када процеси буду суспендовани. Догађај значи да је трансакција изазвала промену вредности сигналу (да је вредност пре и после трансакције различита). Нпр ако се планира трансакција да сигнал постане 0, а сигнал је већ био 0, та трансакција неће изазвати догађај.

|  |
| --- |
| **Z** ПРИМЕР 9, DFF са синхроним, и DFF sa asinhronim resetom  Уводи се: атрибут сигнала, предња ивица клока, if-elsif секвенцијална клаузула.  Треба дефинисати D-флип -флоп са синхроним ресетом и са асинхроним ресетом. Моделовати га као реални флип-флоп са пропагационим кашњењем 2 ns. |
| 01 **ENTITY** edge\_triggered\_Dff **IS**  02 **PORT** **(** D**:** **IN** bit**;** clk**:** **IN** bit**;** clr**:** **IN** bit**;**  03 Q**:** **OUT** bit**);**  04 **END** **ENTITY** edge\_triggered\_Dff**;**  05 -------------------  06 -------------------  07 -------------------  08 **ARCHITECTURE** asyncCLR **OF** edge\_triggered\_Dff **IS**  09 **BEGIN**  10 state\_change**:** **PROCESS** **(**clk**,** clr**)** **IS**  11 **BEGIN**  12 **IF** clr**=**'1' **THEN**  13 Q**<=**'0' **AFTER** 2 ns**;**  14 **ELSIF** clk'**EVENT** **and** clk**=**'1' **THEN**  15 -- <- атрибут сигнала EVENT је True  16 -- ако је сигнал променио вредност.  17 -- Ова конструкција значи предњу ивицу клока  18 Q**<=**D **AFTER** 2 ns**;**  19 **end** **if;**  20 **END** **PROCESS** state\_change**;**  21 **END** **ARCHITECTURE** asyncCLR**;**  22 -------------------  23 -------------------  24 -------------------  25 **ARCHITECTURE** syncCLR **OF** edge\_triggered\_Dff **IS**  26 **BEGIN**  27 state\_change**:** **PROCESS** **(**clk**)** **IS**  28 **BEGIN**  29 **IF** clk'**event** **and** clk**=**'1' **THEN**  30 **IF** clr**=**'1' **THEN**  31 Q**<=**'0' **AFTER** 2 ns**;**  32 **ELSE**  33 Q**<=**D **AFTER** 2 ns**;**  34 **end** **if;**  35 **end** **if;**  36 **END** **PROCESS** state\_change**;**  37 **END ARCHITECTURE** syncCLR**;** |
| l. 14, l. 29: услов clk'**EVENT** **and** clk**=**'1' је стандардни (и синтетизабилни) начин да се моделује детекција предње (узлазне) ивице сигнала.  **IF** клаузула која је искоришћена у овом примеру би требало да буде разумљива сама по себи. Будући да је секвенцијална клаузула, понаша се на исти начин као структура гранања у програмским језицима.  “Синхроност ” неког сигнала значи да тај сигнал треба да има утицаја само у комбинацији са тактним сигналом (на ивицу такта, уколико је ивично окидање; или у току активне полупериоде такта, уколико се ради о окидању нивоом).  Различит третман ресет сигнала у две приказане архитектуре постигнут је редоследом услова: у првој архитектури, asyncCLR, прво се тестира clr (л. 12), и уколико је на њему активна вредност, излаз се ресетује, без обзира на clk. У другој архитектури је обрнуто: ниво clr се тестира само уколико се процес пробудио због догађаја на сигналу clk (л. 27) и ако је условом је установљено да је догађај у ствари предња ивица (л. 29). На овај начин, када год у току периоде клока да се активирао ресет, ефекат ће се десити тек на следећу предњу ивицу клока. |
| За размишљање:  Да ли се (и шта се) мења када се и сигнал clr код DFF-а са синхроним ресетом стави у *sensitivity* лист? Упоредите резултате симулације у једном и другом случају. |

# Управљачке струткуре

Унутар тела процеса могу се користити клаузуле за управљачке стрктуре. У VHDL-у постоје 3 управљачке структуре: **if, for** и **case**. Њихова значења су иста као и у било ком програмском језику.

**IF** клаузула је већ демонстрирана у претхдном примеру, а комплетна синтакса **if** клаузуле је:

|  |
| --- |
| 1 **[**labela ifa**:]** **IF** logicki izraz **THEN**  2 sekvencijalni izraz  3 **ELSIF** logicki izraz **THEN**  4 sekvencijalni izraz  5 **ELSE**  6 sekvencijalni izraz  7 **END** **IF** **[**labela ifa**];** |

|  |
| --- |
| **Z** ПРИМЕР 10, Мултиплексер са ексклузивном селекцијом  Уведено: others - низовни литерали  Треба реализовати мултиплексер који има два улаза и двобитни селекциони улаз са “хотбит” декодирањем - 1 на биту мање тежине активира један улаз, 1 на биту више тежине активира други улаз. 00 поставља на излазу константу 0, а 11 поставља стање високе импедансе (“HiZ”). |
| 01 -------------------------------------------------  02 **LIBRARY** ieee**;**  03 **USE** ieee**.**std\_logic\_1164**.ALL;**  04 --------------------------------------------------  05 **ENTITY** muxEx **IS**  06 **PORT** **(**  07 a**,** b**:** **IN** STD\_LOGIC\_VECTOR**(**7 **DOWNTO** 0**);**  08 sel**:** **IN** STD\_LOGIC\_VECTOR**(**1 **DOWNTO** 0**);**  09 c**:** **OUT** STD\_LOGIC\_VECTOR**(**7 **DOWNTO** 0**));**  10 **END** muxEx**;**  11 ------------------------------------------- -------  12 **ARCHITECTURE** example **OF** muxEx **IS**  13 **BEGIN**  14 **PROCESS** **(**a**,** b**,** sel**)**  15 **BEGIN**  16 **IF** **(**sel**=**"00"**)** **THEN**  17 c **<=** "00000000"**;**  18 **ELSIF** **(**sel**=**"01"**)** **THEN**  19 c **<=** a**;**  20 **ELSIF** **(**sel**=**"10"**)** **THEN**  21 c **<=** b**;**  22 **ELSE**  23 c **<=** **(OTHERS** **=>** 'Z'**);**  24 --или , пошто c има тачно 8 битова: c<="ZZZZZZZZ";  25 **END** **IF;**  26 **END** **PROCESS;**  27 **END** **ARCHITECTURE** example**;**  28 ---------------------------------- |
| л. 23: елемент агрегата **OTHERS**, специфицира да “све остале” чланове агрегата, ненаведене испред у агрегату. У конкрентом случају, обзиром да је ширина порта 8 битова, могла се навести и константа као у л. 24. Да ширина порта зависи од *generic* константе , употреба агрегата као у л. 23 би била неопходна. |
|  |

Нови елемент језика приказан у [пример 10](#Primer10) јесте употреба **агрегата OTHERS**. Агрегат OTHERS се користи када желимо да остатак елемената агрегата, који у агрегату нису наведени испред, поставимо на одређену вредност.

Нпр (0=>’0’, 1|2=>’1’, **others**=>’0’) израз каже да ће се елемент 0 сетовати на ‘0’, елементи 1 и 2 сетовати на ‘1’, а сви остали елементи ће бити сетовани на ‘0’. Овде треба запазити и симбол | (*pipe*), којим се написани индекси раздвајају. Ово је именовани запис агрегације, елементи се именују. Код позиционог записа, наводе се само вредности елемената, у редоследу из декларације. Иста агрегација, под условом да је декларација била **(**0 **TO** X**)** (а не **(**X **DOWNTO** 0**)**), би се могла записати: (‘0’,’1’,’1’, others=>0). Ако постоји **others** у агрегацији, онда мора да се пише на крају.