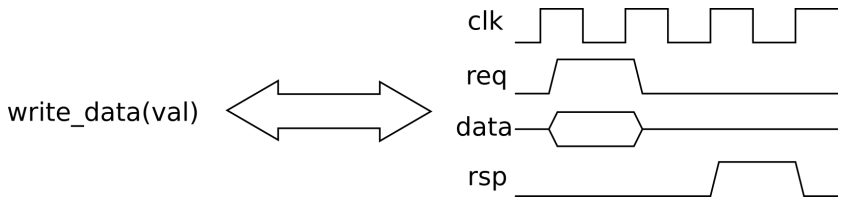
2.2 TLM 2.0 методологија

Трансакција је абстракција за интеракцију или комуникацију између два или више конкурентних процеса. Модели на нивоу трансакција комуницирају позивима функција, за разлику од модела на RTL нивоу абстракције који комуницирају постављањем вредности на појединачним сигналима (слика 2.5). Једном позиву функије може одговарати већи број постављања сигнала и стога су TLM симулације значајно брже од RTL симулација. Модели су једноставнији за писање и брже се развијају па се повећава продуктивност. Ипак су довољно детаљни да се на TLM моделима може развијати софтвер ниског нивоа.



Слика 2.5: Поређење TLM и RTL комуникације

Модели на TLM нивоу абстракције су бржи за симулацију од традиционалних и лакши су за развој. Један од случајева коришћења ових модела је као рана платформа за развој софтвера. Ови модели могу да се користе и за истраживање могућих архитектура на системском нивоу као и за симулацију целих блокова. Развијени модели могу даље да се користе и као алати. Пример за ово би био Instruction Set Simulator (ISS) приликом партиционисања хардвера и софтвера. Често се праве и системске функционалне виртуелне платформе.

Ове моделе могу развијати и користити системски инжењери. Помоћи њих могу развијати систем на вишем нивоу абстракције и потврдити тачан начин рада система. Додатно, ови модели могу служити као спрега или врста уговора са инжењерима који раде саму имплементацију система. Потом, TLM моделе могу користити и софтверски инжењери за рани приступ платформи на којој може да се развија софтвер. Сами модели су добри и за развој у каснијим фазама зато што су значајно бржи од HDL модела. Затим, модели могу служити и као златни, референтни модели за хардверске инжењере. TLM модели могу бити интерпретирани као функционална спецификација за инжењере који раде имплементацију. На крају, TLM модели могу бити део пакета који се испоручује као резултат развоја.

У току развоја идеја и техника у TLM моделима уочени су проблеми везани са недостатком стандардног начина рада. Сваки тим је имао свој

скуп правила развоја модела, који су уз то често били везани за саме алате на којима су развијани. Често су перформансе модела биле лоше или су модели писани тако да није могуће њихово поновно коришћење. Због наведених разлога није било могуће размењивати TLM IP моделе. Због наведених проблема развијен је стандард TLM 2.0 са циљем да се они реше.

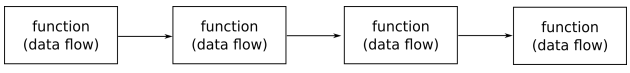
Фокус стандарда TLM је на комуникацији између процеса, посебно на моделовање меморијски мапираних магистрала у оквиру дигиралних система. Развијен је нови стандардни скуп фундаменталних интерфејса као и компатибилних компоненти: података, протокола, сокета и моделовања времена.

2.2.1 Абстрактни нивои у SystemC језику

У оквиру SystemC језика обично се прича о следећим нивоима абстракције:

- Функционални ниво
- Loosely Timed (LT) - Програмски поглед - слободно време
- Approximately Timed (AT) - Програмски поглед са временом - апроксимативно време
- Cycle Accurate Level (CA) - Ниво прецизан у циклус
- RTL ниво

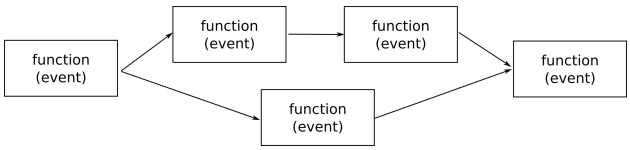
Функционални ниво абстракције служи за развој алгоритама (слика 2.6). Моделује се без архитектурних детаља. Обично се пише у стандардним програмским језицима, као што је C/C++. Функционални модел се извршава најчешће на само једној нити. Моделује се само понашање система као црне кутије.



Слика 2.6: Функционални ниво

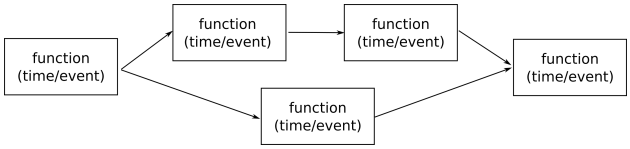
Наредна три нивоа абстракције се подразумевају под TLM нивоом: LT, AT и CA. LT ниво се најчешће користи у оквиру TLM методологије. Некада се овај ниво назива и програмски поглед - Programmer's View (PV). Фокус приликом моделовања је на меморијском мапирању.

Идеја је да се моделују магистрале и регистри система, тако да се добијени архитектурни опис може користити за развој софтвера. Добро моделован LT систем, омогућава паралелан развој хардвера и софтвера (слика 2.7). Време се моделује слободно, по потреби. На пример, интерапти у систему се моделују тако да време протиче, а за остале делове система се време не моделује. Комуникација се остварује преко сокета. Ови модели могу служити и за функционалну верификацију и деле се у оквиру целог тима људи. TLM стандард покрива LT ниво моделовања.



Слика 2.7: LT ниво

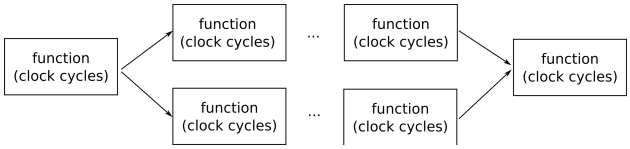
Приликом AT моделовања фокус је на временским карактеристикама протокола за комуникацију (слика 2.8). Модели за комуникацију су профињенији у односу на LT моделе и служе за валидацију архитектуре, процену перформанси и анализу система. Мањи број људи у тиму користи AT моделе у односу на LT. TLM стандард покрива AT ниво моделовања.



Слика 2.8: AT ниво

CA модели служе за добру процену перформанси, потрошње и временских карактеристика система. Фокус приликом моделовања је на временским циклусима (слика 2.9). CA модели су знатно сложенији за развој од LT модела и приближавају се RTL моделима. Перформасе симулације се до 10 пута боље од RTL симулације и бар исто толико пута лошије од LT модела. CA модели су погодни за HLS методологију и у односу на RTL моделе немају детаље о приступима система. TLM стандард не покрива CA ниво моделовања па је потребан већи напор за поновно коришћење ових модела.

RTL модели у оквиру TLM методологије се посматрају као коначна имплементација система. Као што је нетлиста на нивоу капија



Слика 2.9: CA ниво

коначан резултат пројектовања RTL методологије, RTL модел је коначна имплементација за TLM моделовање и нижи нивои абстракције се не разматрају.

2.2.2 Генеричка трансакција

Да би се омогућила што лакше спајање TLM модела, уведен је стандардни тип трансакције који се шаље комуникационим магистралама. Класа `tlm_generic_payload` имплементира информације које се размењују. Ова класа садржи већи број поља којима се може приступити искучиво методама за приступ. Најважнија од ових поља су:

- sc_dt::uint64 address;
- tlm_command command;
- unsigned char* data;
- unsigned int length;
- tlm_response_status response_status;

Поље address одређује адресу којој се приступа. Поље command је набројиви тип података и одређује да ли се чита или пише на одговарајућу адресу. Вредности овог набројаног типа могу бити:

- TLM_READ_COMMAND
- TLM_WRITE_COMMAND
- TLM_IGNORE_COMMAND

Поље `data` је показивач на податке који се уписују уколико је команда `TLM_WRITE_COMMAND`, или је то показивач на место где се смештају подаци уколико је команда `TLM_READ_COMMAND`. Команда `TLM_IGNORE_COMMAND` означава да ништа није потребно да се предузме поводом текуће

трансакције. Поље `length` означава број података који се уписују или читају приликом трансакције. Поље `response_status` је набројиви тип података који означава стање тренутне трансакције. Вредности овог поља могу бити:

- TLM_OK_RESPONSE
- TLM_INCOMPLETE_RESPONSE
- TLM_GENERIC_ERROR_RESPONSE
- TLM_ADDRESS_ERROR_RESPONSE
- TLM_COMMAND_ERROR_RESPONSE
- TLM_BURST_ERROR_RESPONSE
- TLM_BYTE_ENABLE_ERROR_RESPONSE

Вредност `TLM_OK_RESPONSE` означава да је све добро са тренутном трансакцијом. Вредност `TLM_INCOMPLETE_RESPONSE` је подразумевана вредност за ово поље и означава да је трансакција још увек у току. Остале вредности означавају разне врсте грешака.

Стандардни тип трансакције садржи још додатних поља које ће бити објашњена у зависности од конкретног примера. Постоје по две методе за приступ пољима. Метода за читање вредности има име `get_` па име поља, на пример `get_address`. Метода за постављање вредности има име `set_` па име поља са одговарајућим аргументом, на пример `set_address( 0x3456 )`.

2.2.3 Фундаментални TLM транспортни интерфејси

Фундаментални TLM транспортни интерфејси абстракују комплексне двосмерне меморијски мапиране магистрале. На овим магистралама постоје две компоненте, једна која иницира трансакцију, иницијаторкса компонента, и друга која одговара на трансакцију, циљана компонента. Постоје два интерфејса која је потребно имплементирати да би се остварила TLM комуникација. Први интерфејс је `tlm_fw_transport_if` и абстракује комуникацију од иницијаторске компоненте ка циљаној компоненти. Ова комуникација зваће се иницијаторска комуникација. Други интерфејс је `tlm_bw_transport_if` и абстракује комуникацију од циљане компоненте до иницијаторске компоненте. Ова комуникација зваће се циљана комуникација.

Интерфејс `sc_fw_transport_if` захтева да се имплементирају 4 методе:

```
typedef tlm::tlm_base_protocol_types::tlm_payload_type pl_t;
typedef tlm::tlm_base_protocol_types::tlm_phase_type phase_t;

void b_transport(pl_t&, sc_core::sc_time&);
tlm::tlm_sync_enum nb_transport_fw(pl_t&, phase_t&,
        sc_core::sc_time&);
bool get_direct_mem_ptr(pl_t&, tlm::tlm_dmi&);
unsigned int transport_dbg(pl_t&);
```

Интерфејс `sc_fw_transport_if` је шаблон класа која очекује два параметра: тип трансакције који се шаље (`pl_t`) и фазу у којој се комуникација налази (`phase_t`). Подразумевани тип трансакције који се шаље је `tlm_generic_payload` и овај тип се готово никада не треба мењати. Подразумевана фаза у којој се трансакција налази је `tlm_phase_type` и овај тип се исто не треба мењати. Фазе трансакције биће објашњене у каснијим поглављима (поглавља 2.3 и 2.4).

Метода `b_transport` моделује иницијаторску комуникацију функцијом која блокира даље извршавање процеса. Као параметре прима трансакцију која се шаље и време које трансакција треба да траје. Цела комуникације се моделује само једним позивом функције. Параметри су прослеђени као референце, што значи да се параметри могу мењати унутар функције. Иницијаторкса компонента позива ову методу, док је циљана компонента имплементира. Подразумева се да иницијаторкса компонента управља параметрима које прослеђује. Ова метода погодна је за моделовање система на LT нивоу абстракције (поглавље 2.3).

Метода `nb_transport_fw` моделује иницијаторску комуникацију функцијом која не блокира даље извршавање процеса. Као параметри за ову функцију прослеђује се трансакција, фаза у којој се комуникација налази као и време које треба да се утроши за текућу фазу у комуникацији. Метода се користи у пару са сличном методом у циљаној комуникацији. Да би се комуникација имплементирала коришћењем неблокирајућих метода, потребно је неколико позива ових функција. Параметри су прослеђени као референце. Ова метода служи да се имеплементира систем на AT нивоу абстракције (поглавље 2.4).

Метода `get_direct_mem_ptr` служи за директан приступ меморији циљаног објекта, уз помоћ показивача. Ова метода заобилази главни ток комуникације и служи да би се убрзала симулација. Један

пример коришћења ове методе би био иницијализација програмске меморије у систему. Уколико би се меморија иницијализовала регуларним комуникационим путем, требао би велики број циклуса за иницијализацију. Уместо тога, помоћу методе за директан приступ, директно би се преузео показивач на циљану меморију и подаци би се писали директно и брзо на циљано место.

Метода `transport_dbg` слична је блокирајућој транспортној методи. Ова метода служи за уклањање грешака приликом склапања система коришћењем TLM методологије и такође заобилази главни ток комуникације. Метода не користи симулационо време.

Интерфејс `tlm_bw_transport_if` захтева да се имплементирају 2 методе:

```
typedef tlm::tlm_base_protocol_types::tlm_payload_type pl_t;
typedef tlm::tlm_base_protocol_types::tlm_phase_type phase_t;

tlm::tlm_sync_enum nb_transport_bw(pl_t&, phase_t&,
        sc_core::sc_time&);
void invalidate_direct_mem_ptr(sc_dt::uint64, sc_dt::uint64);
```

Интерфејс `tlm_bw_transport_if` је шаблон класа која очекује исте параметре као и `tlm_fw_transport_if` и параметри имају идентично значење.

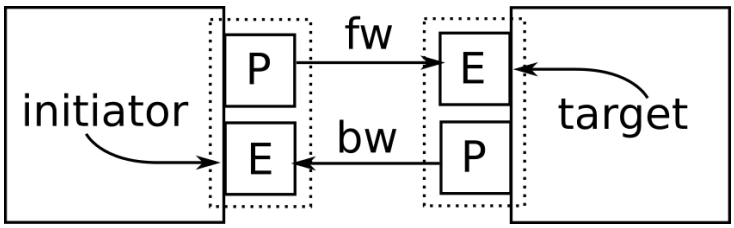
Метода `nb_transport_bw` моделује циљану комуникацију функцијом која не блокира даље извршавање процеса. Параметри су идентични онима које користи метода `nb_transport_fw`. Ове две методе у пару се користе за моделовање двосмерне комуникације на AT нивоу абстракције и у неким случајевима потребан је већи број позива ових метода да би се целокупна комуникација остварила.

Метода `invalidate_direct_mem_ptr` заобилази главни ток комуникације и служи да забрани приступ одређеним регионима меморије. Користи се у комбинацији са методом `get_direct_mem_ptr`.

Имплементацијом фундаменталних интерфејса може се остварити компатибилност са TLM 2.0 стандардном. Но, сама имплементација ових интерфејса на било који начин не значи да је испоштован стандард. Тачним правилиа комуникације по стандарду биће описана (поглавља 2.3 и 2.4). Фундаментални интерфејси се користе у комбинацији са прикључцима да би се моделовали Electronic System Level (ESL) систем.

2.2.4 Прикључци

Прикључак је новина уведена у TLM стандарду. Стандард је фокусиран око моделовања комплексних меморијски мапираних магистрала. Ове магистрале увек користе двосмерну комуникацију. Стога је у стандард уведен концепт прикључка који представља спајање класе `sc_port` и `sc_export` (слика 2.10). Постоје две главе врсте прикључака, иницијаторски прикључак и циљни прикључак.



Слика 2.10: Идеја сокета

Иницијаторски прикључак је SystemC приступ (`sc_port`) за приступање методама интерфејса на иницијаторском путу. Овај приступ садржи SystemC извозник (`sc_export`) за извоз интерфејс метода на циљаном путу. Имплементиран је класом `tlm_initiator_socket`. Интерфејс коме се приступа је `tlm_fw_transport_if`, а интерфејс који се извози је `tlm_bw_transport_if`. Класа која саджи иницијаторкси прикључак наслеђује `tlm_bw_transport_if`. Шаблон за овакву класу је приказан наредним кодним фрагментом.

```
class initiator :
public sc_core::sc_module,
public tlm::tlm_bw_transport_if<>
{
...
tlm::tlm_initiator_socket<> isoc;
...
};
```

Циљни прикључак је SystemC извозник (`sc_export`) за извоз метода интерфејса на иницијаторском путу. Овај извозник садржи SystemC приступ (`sc_port`) за приступ методама на циљаном путу. Имплементиран је класом `tlm_target_socket`. Интерфејс коме се приступа је `tlm_bw_transport_if`, а интерфејс који се извози је `tlm_fw_transport_if`. Класа која саджи циљни прикључак наслеђује

`tlm_fw_transport_if`. Шаблон за овакву класу је приказан наредним кодним фрагментом.

```
class target :
public sc_core::sc_module,
public tlm::tlm_fw_transport_if<>
{
public:
...
tlm::tlm_target_socket<> tsoc;
...
}
```

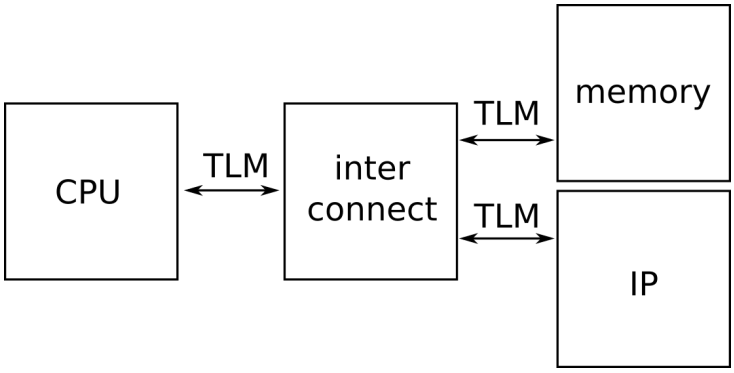
Прикључци су уведени у стандард да би се смањио број потрених повезивања приликом моделовања система. Коришћењем прикључака двоструко се смањује број потребних веза у систему у односу на систем у коме се корсите приступи и извозници. Прикључци се повезују слично као приступи. Увек се повезује иницијаторкси прикључак на циљни прикључак. Пример за повезивање претходна два прикључка би био:

```
target trg ("trg_u");
initiator int("int_u");
int.isoc(trg.tsoc);
```

2.2.5 Пример ESL система

У комплексним ESL системима увек постоји бар један процесор, а највероватније и више њих. У систему постоји бар једна меморија као и бар један додатни хардверски IP блок. То значи да најједноставнији реалан систем садржи бар 4 компоненете: процесор, меморију, IP језгро и подсистем за повезивање (слика 2.11). Процесору су меморија и IP језгро меморијски мапирани. Компонента за повезивање на основу тражених адреса прослеђује трансакцију или меморији или IP језгру. Стандард TLM 2.0 је настао управо да реши проблеме компатибилности у системима са меморијским мапирањем, а то је већина ESL система.

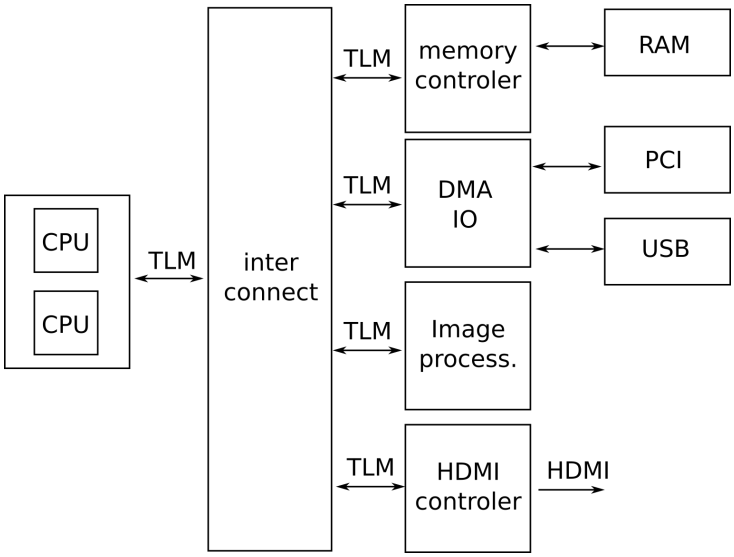
Приликом моделовања оваквих система, добро је да се корсите TLM стандарди за повезивање. На тај начин се омогућава лакше додатно коришћење. На пример, уколико се моделује компонента за повезивање процесора са остатком систма, једном, по овом стандарду, тада се та компоненета може користити у новим пројектима, поново. Могу се правити и библиотека стандардних компоненети које су добре за поновно



Слика 2.11: Једноставан ESL систем

коришћење. Највећи произвођачи EDA софтвера пружају библиотеке оваквих компоненети у својим алатима за ESL моделовање.

Као следећи пример узмимо систем за обраду и приказ слике (слика 2.12). Систем може да се састоји од процесора са два језгра, Random Access Memory (RAM) меморије, меморијског контролера, Direct Memory Access (DMA) за комуникацију са Peripheral Component Interconnect Extended (PCI-X) и Universal Serial Bus (USB) периферијама, хардверског блока за обраду слике и High-Definition Multimedia Interface (HDMI) контролером.



Слика 2.12: ESL систем за обраду слике

Систем би требао да ради на следећи начин. Преко PCI-X или USB периферија слика би се примила у систем. Слика би се сместила у RAM

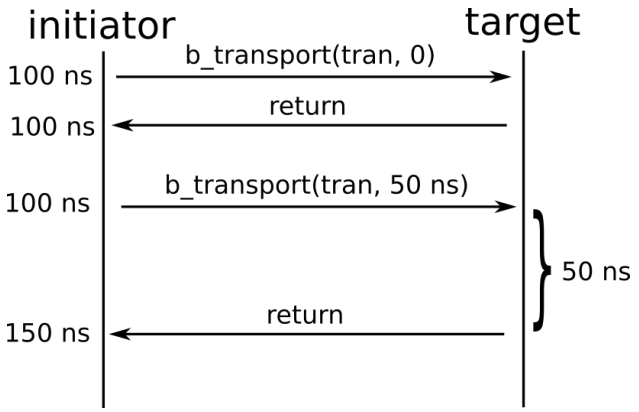
меморију без коришћења процесора, кроз DMA контролер и меоријски контролер. Слику би обрадио хардверски блок за обраду слике која би се на крају приказала на монитору преко HDMI контролера. Синхронизацију акција у систему би обављао процерор. Комуникација процесора и остатка система би се обављала преко подсистема за повезивање.

Овакав систем је веома комплексан. Идеја TLM методологије је да се цео систем реализује трансакцијама помоћу стандардних TLM интерфејса. Ово може да се уради релативно брзо. Коришћењем модела на висиком нивоу абстракције би се брзо уочили и уклонили потенцијални проблеми у систему. Тај модел би могао да послужи као Virtual Platform (VP) за развој софтвера. На тај начин би се развој целог система убрзвао паралелним развојем хардвера и софтвера. Коначна имплементација хардвера би се могла постићи постепеним профињавањем абстрактних модела.

2.3 Стандард TLM 2.0 - LT модели

Модели на LT нивоу абстракције користе блокирајуће методе фундаменталних интерфејса. Цела трансакција се обавља само једним позивом методе `b_transport` иницијаторског интерфејса. Трансакција се моделује стандардном генеричком трансакцијом (`tlm_generic_payload`). Комуникација започиње у тренутку позива методе и завршава се након времена прослеђеног као временски параметар методе `b_transport`.

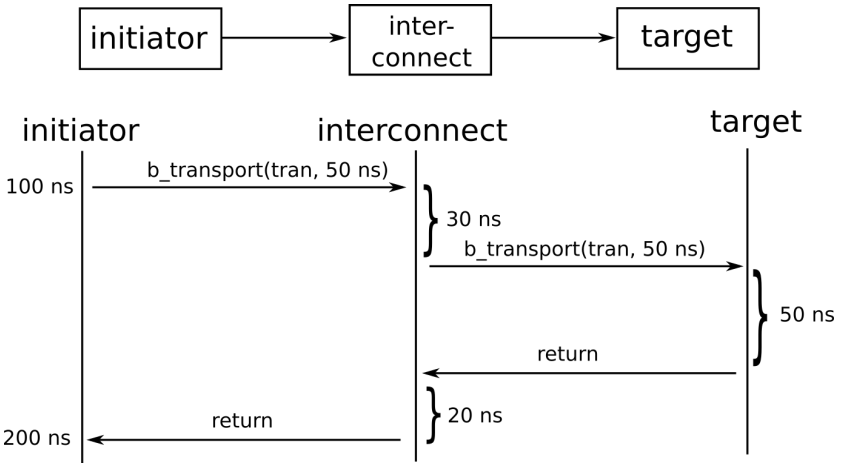
На пример, уколико комуникација почиње у тренутку 100 ns и као времески параметар се проследи 0 ns тада се цела трансакција завршава у тренутку 100 ns (слика 2.13). Уколико се у тренутку 100 ns опет започиње комуникација, али се као временски параметар проследи 50 ns, тада би цела трансакција требала да се заврши у тренутку 150 ns.



Слика 2.13: Проток времена блокирајуће методе

Временски параметар који се прослеђује блокирајућој методи означава време које треба да се дода на тренутно симулационо време. Другим речима, то је време које треба да протекне због тог позива методе. То није време након кога ће иницијатор комуникације добити контролу над извршавањем у сваком случају! Ако имамо комплекснији систем, иницијатор може да добије контролу и у каснијем временском тренутку (слика 2.14). Рецимо да је комуникација започела у тренутку 100 ns и да је иницијатор позвао методу са временом 50 ns. Компонента за повезивање, може да чека сачека 20 ns, па затим да позове блокирајућу методу на свом иницијаторком прикључку са неким временом, рецимо 50 ns. Циљана компонента одговориће након 50 ns и вратити контролу компоненти за повезивање. Компонента за позивање сечекаће још 30 ns, да би се поштовао параметар иницијаторског позива (20 + 30 = 50), и

тек онда вратитће контролу иницијаторској компоненти. Иницијаторкса компонента добиће контролу тек у тренутку 200 ns.



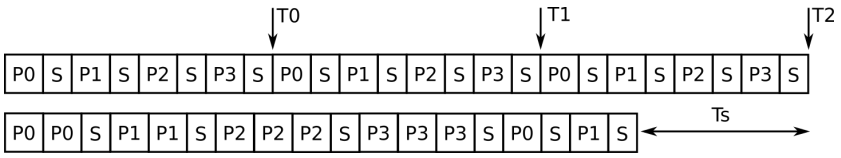
Слика 2.14: Проток времена при ланчаном позиву

2.3.1 Временско раздвајање

Време потрошено у току симулације може се поделити на време у коме се процеси извршавају и време у коме кернел симулатора мења контекс, односно преузима контролу од једног процеса и предаје контролу другом процесу. За корисно време симулације може се узети време проведено у извршавању процеса. Што је већи број процеса и што је краће време за које кернел треба да мења контекст, то ће кернел проводити више времена у мењању контекста и симулација ће се извршавати спорије. Но, што се чешће мења контекста, то ће симулација бити прецизнија.

Циљ временског раздвајања је да смањи број подребних мењања контекста а да се при томе добију прецизни резултати симулације. Идеја је да се процесу допусти да се извршава неко време испред тренутног времена симулације без мењања контекста. То време зваћемо квантно време. Максимално допуштено време, које стоји на раполагању процесу да се извршава, без мењања контекста, назива се максимално квантно време.

Узмимо за пример симулацију са 4 процеса (`P0`, `P1`, `P2`, `P3`, слика 2.15) за које кернел мења контекст сваку растућу ивицу синхронизационог сигнала. Рецимо да симулација траје 3 догађаја (`T0`, `T1`, `T2`). Да би се симулирао један догађај потребно је 4 мењања контекста (`S`). Укупно трајање симулације садржаће 12 мењања контекста.



Слика 2.15: Побољшање перформанси временским раздвајањем

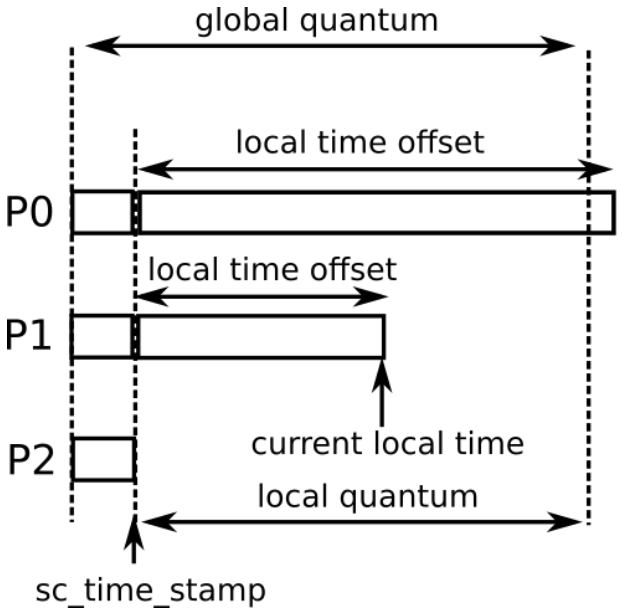
Уколико би се имплементирала симулација са временским раздвајањем, при чему би максимално квантно време износило 3 догађаја, оваква симулација могла би се убрзати. Рецимо да прва два процеса (`P0` и `P1`) могу да се извршавају два догађаја без губитка тачности, а друга два процеса (`P2` и `P3`) могу да се извршавају три догађаја. При симулацији прва два процеса уштедело би се по једно мењање контекста, а за друга два догађаја уштедело би се по два мењања контекста. Симулацја са временским раздвајањем извршила би се брже за време које износи 6 времена потребних за мењање контескта (`Ts`).

Временско раздвајање је техника које је погодна за LT моделовање. Овом техником симулација се синхронизује у временским интервалима који се називају глобално квантно време (слика 2.16). Глобано квантно време је потребно одабрати тако да буде довољно велико да убрзава симулацију и довољно мало да симулација буде довољно презизна. Квантни циклус је један пролазак свих процеса кроз глобално квантно време.

Време од последње синхронизације до наредног глобалног квантног тренутка назива се локално квантно време. У једном квантном циклусу на случајан начин се бирају процесу и додељује им се право на извршавање. Сваки процес ажурира протекло време након последњег тренутка синхронизације. То време се назива локални вермески офсет. Када локални временски офсет постане већи од локалног квантног времна, процес зауставља своје извршавање и враћа контролу кернелу симулатора. Тренутно локално време је збир времена последњег тренутка синхронизације и локалног временског офсета.

Процеси чији је локални временски офсет већи од локалног квантног времена су завршили са извршавањем у тренутном квантном циклусу (P0 слика 2.16). Процеси који се извршавају у тренутном квантном циклусу имају локални временски офсет мањи од локалног квантног времена (`P1`). Процеси који још увек нису покренути у тренутном квантном циклусу имају локални квантни офсет који је једнак 0 ns (`P2`).

SystemC језик садржи класе које олакшавају имплементацију временског раздвајања. Класа `tlm_quantumkeeper` омогућава да се



Слика 2.16: Симулација са временским зардвајањем

одржава локални времески офсет и тренутно локално време. Најважније методе ове класе су:

```
void reset();
void inc( const sc_core::sc_time& );
void set( const sc_core::sc_time& );
sc_core::sc_time get_current_time() const;
sc_core::sc_time get_local_time();
bool need_sync() const;
void sync();
void set_and_sync(const sc_core::sc_time&);
```

Метода reset поставља све временске тренутке на нулу и треба да се позове пре почетка коришћења осталих метода. Метода set поставља тренутно локално офсет време. Метода inc надодаје на локално офсет време. Тренутно локлано офсет време се може добити позивом `get_local_time` методе док се позивом `get_current_time` добија тренутно локално време. Метода `need_sync` одређује да ли је потребна синхронизација, док метода `sync` синхронизује. Најпоузданије је користити методу `set_and_sync` која поставља тренутни локални офсет, проверава да ли је потребна синхронизација и уколико јесте, синхронизује се са глобалним квантим временом.

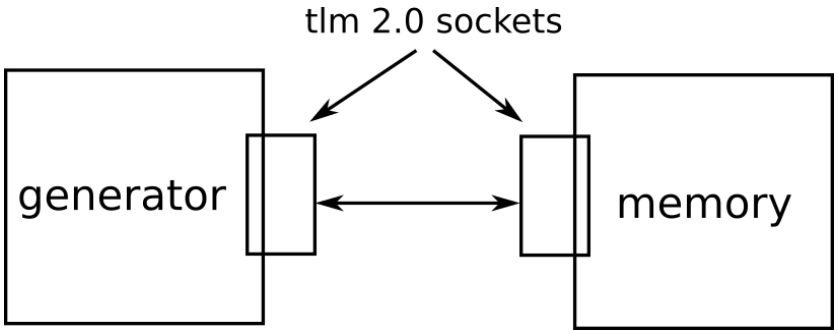
Да би се поставило глобално квантно време користи се класа `tlm_global_quantum`. Ова класа је синглтон. Најважније методе ове класе су:

```
static tlm_global_quantum& instance();
void set( const sc_core::sc_time& );
```

Метода instance враћа јединствени објекат класе `tlm_global_quantum`. Метода set поставља глобално квантно време.

2.3.2 Пример једноставног LT система

Најједноставнији TLM систем састоји се од иницијаторске компоненете и циљне компоненте (слика 2.17). Иницијаторкса компонента генерисаће уписе и читања за циљну компоненту. Циљна компонента представљаће модел меморије. Трансакције састојаће се од захтева за писање или читање од 1 до 4 речи из меморије.



Слика 2.17: Једноставан TLM систем

У примеру биће имплементирани две врсте модела: модели који не троше време и модели који користе временско раздвајање. Разлика између модела је веома мала и стога коришћена је предпроцесорска дитектива да би се уметнуле разлике између ове две врсте модела. На основу тога да ли је дефинисана константа QUANTUM уметаће се додатни делови кода.

Компоненте се инстанцирају и повезују у главном програму (листинг 2.3). Компоненте су инстанциране и повезане помоћу пркључака. У главном програму је постављено глобално квантно време, позивом методе set синглтон објекта класе `tlm_global_quantum`, уколико је дефинисана константа `QUANTUM`.

Листинг 2.3: Главни програм једноставног LT система

```
#include <systemc>
#include "generator.hpp"
#include "memory.hpp"

using namespace sc_core;
using namespace tlm;

int sc_main(int argc, char* argv[])
{
        memory mem ("memory_u");
        generator gen("generator_u");
        gen.isoc(mem.tsoc);
        #ifdef QUANTUM
        tlm_global_quantum::instance().set(sc_time(10, SC_NS));
        #endif

        sc_start(200, SC_NS);
    return 0;
}
```

Иницијатор комуникације у овом систему имплементиран је класом generator. Ова класа налеђује иницијатоски интерфејс (листинг 2.4). Осим неопходног иницијаторског прикључка (`isoc`), као и метода иницијаторског интерфејса, ту је и један процес(`gen`), који генерише трансакције. Дефинисане су и две променљиве (`dmi_valid` и `dmi_mem`) помоћу којих ће се демонстрирати метода за директан приступ меморији циљног објекта.

Листинг 2.4: Заглавље класе generator LT модел

```
#ifndef _GENERATOR_HPP_
#define _GENERATOR_HPP_

#include <systemc>
#include <tlm>

class generator :
        public sc_core::sc_module,
        public tlm::tlm_bw_transport_if<>
{
public:
        generator(sc_core::sc_module_name);

        tlm::tlm_initiator_socket<> isoc;

        typedef tlm::tlm_base_protocol_types::tlm_payload_type pl_t;
        typedef tlm::tlm_base_protocol_types::tlm_phase_type phase_t;

        tlm::tlm_sync_enum nb_transport_bw(pl_t&, phase_t&, sc_core::sc_time&);
        void invalidate_direct_mem_ptr(sc_dt::uint64, sc_dt::uint64);

protected:
        void gen();
        bool dmi_valid;
        unsigned char* dmi_mem;
};

#endif
```

У конструктору класе региструје се процес и извози се сама класа у извозни део прикључка (листинг 2.5 isoc). Главни процес (`gen`) у главној и јединој for петљи, прави 20 трансакција за писање по меморији на сличајан начин а потом 20 трансакција читања. У извршавања процеса исписују се информативне поруке помоћу макроа `SC_REPORT_INFO`. Овај макро испиује поруке озбиљности `INFO`. Постоје сличне поруке и

за остале нивое озбиљности: `SC_REPORT_WARNING`, `SC_REPORT_ERROR` и `SC_REPORT_FATAL`.

Листинг 2.5: Имплементација класе generator LT модел

```
#include "generator.hpp"
#include <tlm_utils/tlm_quantumkeeper.h>

using namespace sc_core;
using namespace sc_dt;
using namespace tlm;

SC_HAS_PROCESS(generator);

generator::generator(sc_module_name name) :
        sc_module(name),
        isoc("isoc"),
        dmi_valid(false)
{
        SC_THREAD(gen);
        isoc(*this);
}


void generator::gen()
{
        tlm_generic_payload pl;
        sc_time offset = SC_ZERO_TIME;
        unsigned char buf[4];

        // This is used only for debugging
        unsigned char dbg_buf[10000];
        tlm_generic_payload dbg_pl;
        dbg_pl.set_data_ptr(dbg_buf);
        dbg_pl.set_command(TLM_READ_COMMAND);

        // Init first 20 locations to FF with DMI access
        tlm_dmi dmi;
        dmi_valid = isoc->get_direct_mem_ptr(pl, dmi);
        if (dmi_valid)
        {
                dmi_mem = dmi.get_dmi_ptr();
                for (int i = 0; i != 20; ++i)
                        dmi_mem[i] = 0xFF;
        }

        #ifdef QUANTUM
        tlm_utils::tlm_quantumkeeper qk;
        qk.reset();
        #endif

        // Normal TLM transport interface.
        for (unsigned int i = 0; i != 40; ++i)
        {
                unsigned int data_length = 1 + rand() % 4;
                unsigned int addr = rand() % 200;
                tlm_command cmd = i < 20 ? TLM_WRITE_COMMAND : TLM_READ_COMMAND;
                std::string msg = cmd == TLM_WRITE_COMMAND ? "Write " : "Read  ";
                if (cmd == TLM_WRITE_COMMAND)
                        for (unsigned i = 0; i != data_length; ++i)
                        {
                                buf[i] = rand() % 100;
                                msg += std::to_string(buf[i]);
                                msg += " ";
                        }
                msg += "- address ";
                msg += std::to_string(addr);
                msg += " ";

                pl.set_command          ( cmd                       );
                pl.set_address          ( addr                      );
                pl.set_data_ptr         ( buf                       );
                pl.set_data_length      ( data_length               );
                pl.set_response_status ( TLM_INCOMPLETE_RESPONSE );

                #ifdef QUANTUM
                qk.inc(sc_time(4, SC_NS));
                offset = qk.get_local_time();
                #else
                offset += sc_time(4, SC_NS);
                #endif
```

```
                isoc->b_transport(pl, offset);
                if (cmd == TLM_READ_COMMAND)
                        for (unsigned i = 0; i != data_length; ++i)
                        {
                                msg += std::to_string(buf[i]);
                                msg += " ";
                        }
                SC_REPORT_INFO("generator", msg.c_str());

                #ifdef QUANTUM
                qk.set_and_sync(offset);
                #endif

                /*
                  Print debugging information using method call for debugging
                  purpose.
                 */
                isoc->transport_dbg(dbg_pl);
                msg = " RAM at time " + sc_time_stamp().to_string();
                msg += "\n";
                for (int i = 0; i != 200; ++i)
                {
                        msg += std::to_string(dbg_pl.get_data_ptr()[i]);
                        msg += " ";
                }
                SC_REPORT_INFO("generator", msg.c_str());
        }
}

tlm_sync_enum generator::nb_transport_bw(pl_t& pl, phase_t& phase, sc_time& offset)
{
        return TLM_ACCEPTED;
}

void generator::invalidate_direct_mem_ptr(uint64 start, uint64 end)
{
        dmi_valid = false;
}
```

Помоћу прикључка позива се блокирајућа метода за слање циљног интерфејса `b_transport`. Овим позивом се обавља комуникација. Уколико је дефинисано `QUANTUM` позивају се и методе класе `quantum_keeper` одговарајуће променљиве (`kq`). Методом `inc` ажурира се локални времески офсет, док се методом `set_and_sync` поставља тренутно локално време и обавља синхронизација, уколико је то потребно.

Након синхронизације, илустрована је употреба методе чији је циљ лов на грешке, `transport_dbg`. Копишћењем ове методе, преузет је садржај меморије циљане компоненте и приказан је у процесу иницијаторске компоненете. Ова метода знатно успорава симулацију, пошто се у њеној имплементацији копира велики број вредности. У великим пројектима ова метода се користи док се не уклоне грешке у систему, а потом, уколико су перформансе симулације од значаја, уклања се из модела.

Пре главне и једине for петље користи се метода циљног прикључка за директан приступ меморији `get_direct_mem_ptr`. Помоћу ове методе, иницијаторска компонента може добити директан приступ, помоћу показивача, меморијском региону циљне компоненте. Ова метода служи да се у току нормалног рада убрза симулација. Уколико циљна компонента не подржава директан приступ меморији, имплементација

ове методе треба да врати false вредност. У примеру се помоћу добијеног показивача првих 20 локација добијеног региона поставља на вредност 0xFF.

Неблокирајућа метода за комуникацију иницијаторског интерфејса `nb_transport_bw` је тривијална, пошто се ова метода не користи за LT моделовање. Метода `invalidate_direct_mem_ptr` може да поништи регион меморије додељене директним приступом. У примеру се поништава целокупан приступ додељеној меморији.

Циљна компонента комуникације имплементирана је класом memory. Ова класа налеђује циљни интерфејс (листинг 2.6). Осим циљног прикључка (`tsoc`), као и метода циљног интерфејса, ова класа садржи још само промнљиву која моделује меморијске локације (`ram`). Ова класа је у потпуности пасивна и не садржи процесе. Додатно, у овом примеру, ова класа не исписује поруке, већ је за то коришћена иницијатоска компонента и метода `transport_dbg`.

Листинг 2.6: Заглавље класе memory LT модел

```
#ifndef _MEMORY_HPP_
#define _MEMORY_HPP_

#include <systemc>
#include <tlm>

class memory :
        public sc_core::sc_module,
        public tlm::tlm_fw_transport_if<>
{
public:
        memory(sc_core::sc_module_name);

        tlm::tlm_target_socket<> tsoc;

        typedef tlm::tlm_base_protocol_types::tlm_payload_type pl_t;
        typedef tlm::tlm_base_protocol_types::tlm_phase_type phase_t;

        void b_transport(pl_t&, sc_core::sc_time&);
        tlm::tlm_sync_enum nb_transport_fw(pl_t&, phase_t&, sc_core::sc_time&);
        bool get_direct_mem_ptr(pl_t&, tlm::tlm_dmi&);
        unsigned int transport_dbg(pl_t&);

protected:
        static const int RAM_SIZE = 1024;
        unsigned char ram[RAM_SIZE];
};

#endif
```

У конструктору класе извози се сама класа у извозни део прикључка (листинг 2.7 `tsoc`) па се потом иницијализује меморија.

Имплементација методе за блокирајуће слање, `b_transport` је директна. Поља улазне трансакције (`pl`) читају се и тумаче. Уколико се читају подаци, пишу се информације из променљиве која моделује меморију (`ram`) у бафер трансакције. Уколико се пишу подаци, читају се подаци из бафера трансакције у променљиву. У оба случаја се означава да је трансакција добро обрађена (`fTLM_OK_RESPONSE`). Уколико се добије нека трећа команда, имплементација ове меморије означава

да је дошло до грешке приликом обраде (TLM_COMMAND_ERROR_RESPONSE). Улазни временски офсет се повећава. Циљна компонента не траба да се бави синхронизацијом, па се класа quantum_keeper не користи.

Листинг 2.7: Имплементација класе memory LT модел

```
#include "memory.hpp"

using namespace sc_core;
using namespace tlm;
using namespace sc_dt;

memory::memory(sc_module_name name) :
        sc_module(name),
        tsoc("tsoc")
{
        tsoc(*this);
        for (int i = 0; i != RAM_SIZE; ++i)
                ram[i] = 0;
}

void memory::b_transport(pl_t& pl, sc_time& offset)
{
        tlm_command cmd    = pl.get_command();
        uint64 adr         = pl.get_address();
        unsigned char *buf = pl.get_data_ptr();
        unsigned int len   = pl.get_data_length();

        switch(cmd)
        {
        case TLM_WRITE_COMMAND:
                for (unsigned int i = 0; i != len; ++i)
                        ram[adr++] = buf[i];
                pl.set_response_status( TLM_OK_RESPONSE );
                break;
        case TLM_READ_COMMAND:
                for (unsigned int i = 0; i != len; ++i)
                        buf[i] = ram[adr++];
                pl.set_response_status( TLM_OK_RESPONSE );
                break;
        default:
                pl.set_response_status( TLM_COMMAND_ERROR_RESPONSE );
        }

        offset += sc_time(3, SC_NS);
}

tlm_sync_enum memory::nb_transport_fw(pl_t& pl, phase_t& phase, sc_time& offset)
{
        return TLM_ACCEPTED;
}

bool memory::get_direct_mem_ptr(pl_t& pl, tlm_dmi& dmi)
{
        dmi.allow_read_write();

        dmi.set_dmi_ptr        ( ram );
        dmi.set_start_address ( 0   );
        dmi.set_end_address   ( 199 );

        return true;
}

unsigned int memory::transport_dbg(pl_t& pl)
{
        tlm_command cmd = pl.get_command();
        unsigned char* ptr = pl.get_data_ptr();

        if ( cmd == TLM_READ_COMMAND )
                memcpy(ptr, ram, RAM_SIZE);
        else if ( cmd == TLM_WRITE_COMMAND )
                memcpy(ram, ptr, RAM_SIZE);

        return RAM_SIZE;
}
```

Неблокирајућа метода за комуникацију циљног интерфејса `nb_transport_fw` је тривијална, пошто се ова метода не користи за LT моделовање.

Имплементација методе за директан приступ меморији, `get_direct_mem_ptr`, омогућава приступу интерној променљивој која моделује меморију (`ram`). Помоћу метода `tlm_dmi` класе, прослеђује се показивач за приступ меморији и даје се приступ до првих 200 локација. Повратна вредност означава да је могуће користити директан приступ меморији ове циљне компоненте.

Имплементација методе `transport_dbg`, интепретира само неке параметре улазне трансакције (`pl`). Уколико се ради читање из низа, за моделовање меморије (`ram`), уписују се подаци у излауни бафер (`ptr`). Уколико је команда трансакције писање, подаци из улазног бафера се копирају у низ. Ова метода не моделује време, не користи се у главном току комуникације и служи за лов на грешке.

2.3.3 Вежбе

1. Моделовати неки IIR филтар 5 реда, TLM LT стилом. Модел треба да садржи регистар у који се уписује вредност улазног одбирка, као и регисар који садржи вредност излазног одбирка. Направити тестбенч за овај модел. Тестбенч треба да садржи референтни модел са којим се пореди TLM - LT модел.

2. Моделовати генератор Фибоначијевих бројава TLM LT стилом. Модел треба да садржи улазни регистар који одређује колико је дугачка генерисана секвенца. Постоји и регисар који садржи вредност генерисаног броја. Направити тестбенч за овај модел. Тестбенч треба да садржи референтни модел са којим се пореди TLM - LT модел.