Tema 1 - Transferuri de date DMA intr-o arhitectura de tip Cell

Termen de trimitere a temei: Luni, 31 martie 2008, ora 23:55

1. Specificatii functionale

O arhitectura de tip Cell consta din urmatoarele componente, vizibile si in figura:

- N Synergistic Processing Units (SPU), numerotate 0, 1, ..., N-1
- N module pentru realizarea de transferuri de date DMA (Direct Memory Access) - cate unul pentru fiecare SPU (numerotate la fel ca SPU-ul corespunzator)
- K inele pentru transferul datelor in sensul 0 (sensul direct: 0, 1, 2, ...), numerotate 0, 1, ..., K-1
- K inele pentru transferul datelor in sensul 1 (sensul invers: 0, N-1, N-2, ...), numerotate 0, 1, ..., K-1
- 1 Arbitru de Date - gestioneaza inelele pentru transferul datelor
- 1 Magistrala de Comenzi - modulele DMA trimit comenzi de la unul la altul pe aceasta magistrala

Un exemplu concret de utilizare al acestui model poate fi observat in Element Interconnect Bus din Procesorul CellBE.

1.1. SPU-ul

Fiecare SPU este alcatuit dintr-un procesor si o memorie avand **mem_size** bytes. Adresele din memorie sunt numerotate de la **0** la **mem_size-1**. Programul executat pe un SPU consta dintr-o secventa de instructiuni. O instructiune poate fi una din urmatoarele:

- **read addr** -> intoarce valoare byte-ului de la adresa **addr** din memoria SPU-ului
- **write addr, v** -> seteaza byte-ul de la adresa **addr** din memoria SPU-ului la valoarea **v**
- **get myaddr, spuid, spuaddr, size** -> copiaza **size** bytes de la spu-ul numerotat cu **spuid**, incepand cu adresa **spuaddr**, in memoria proprie, incepand cu adresa **myaddr**
  - in pseudocod:
    - pentru **i** de la **0** la **size-1**:
      - copiaza byte-ul din memoria spu-ului **spuid**, de la adresa **spuaddr+i**, in memoria proprie la adresa **myaddr+i**
- **put myaddr, spuid, spuaddr, size** -> copiaza **size** bytes din memoria proprie, incepand cu adresa **myaddr**, in memoria spu-ului numerotat cu **spuid**, incepand de la adresa **spuaddr** a acestuia
  - in pseudocod:
    - pentru **i** de la **0** la **size-1:**
      - copiaza byte-ul din memoria proprie, de la adresa **myaddr+i**, in memoria spu-ului **spuid**, la adresa **spuaddr+i**

O instructiune de tip **read/write** realizeaza direct citirea/scrierea din/in memoria SPU-ului. Instructiunile de tip **get** si **put** sunt transmise modulului DMA asociat, sub forma de cereri. Modulul DMA intoarce un identificator al cererii (unic la nivelul SPU-ului respectiv) si, in continuare, modulul DMA se va ocupa de transferul datelor, iar programul SPU-ului isi va continua executia. O a cincea instructiune pe care o poate executa un program SPU este aceea de a interoga modulul DMA cu privire la status-ul unei cereri de transfer de date (**status request_id**). Modulul DMA intoarce unul din urmatoarele 4 status-uri posibile:

- identificator de cerere invalid

- transferul inca nu a inceput
- transferul este in desfasurare
- transferul s-a incheiat

Impreuna cu status-ul, modulul DMA va intoarce si sensul si inelul pe care are loc transferul respectiv (daca transferul nu a inceput sau s-a terminat, sensul si inelul pot fi valori oarecare).

1.2. Modulul DMA

Fiecare modul DMA este conectat la fiecare din cele **2*K** inele de date. La un inel cu sensul 0, modulele DMA sunt conectate in mod circular, in ordinea numerotarii acestora (0, 1, 2, ..., N, 0). La un inel cu sensul 1, modulele DMA sunt conectate in ordine inversa numerotarii (0, N-1, N-2, ..., 1, 0), dar tot circular. Pentru a realiza un transfer de date intre memoriile a 2 SPU-uri, **X** (cel care transmite datele) si **Y** (cel care receptioneaza datele), este necesar sa se rezerve portiunea unui inel din cele 2*K dintre modulul DMA corespunzator SPU-ului X si cel corespunzator SPU-ului Y, in sensul de parcurgere al inelului. Aceasta portiune consta din punctele unde sunt conectate modulele DMA X si Y, plus tot segmentul de inel dintre aceste 2 puncte (in sensul inelului). In cazul in care un SPU A initiaza un transfer de tip **get** de la un SPU B, atunci cel care transmite datele este B, iar cel care le receptioneaza este A; in cazul unui transfer de tip **put** initiat de SPU-ul A, cel care transmite datele este A, iar cel care le receptioneaza este B.

Pe acelasi inel se pot afla in desfasurare **oricate** transferuri, cu conditia ca segmentele ocupate de acestea sa nu aiba nici un punct in comun (nici macar SPU-ul care transmite datele sau cel care le primeste).

1.3. Arbitrul de Date

Entitatea care gestioneaza starea tuturor inelelor este numita **Arbitru de Date**. Arbitrul de Date este cel care decide cand poate incepe un transfer de date si pe care inel. Modulele DMA comunica cu arbitrul pentru a obtine permisiunea de a incepe un transfer. O data ce transferul incepe, este doar responsabilitatea modulelor DMA sa transfere date (arbitrul nu se ocupa de transferuri efectiv, doar de gestiunea inelelor).

1.4. Functionarea la nivel de ciclu

Fiecare SPU, modul DMA si arbitrul functioneaza la nivel de ciclu (1 ciclu = 1 unitate de timp). Pe durata unui ciclu, acestea isi desfasoara activitatea normala:

- programul SPU-ului executa una sau mai multe instructiuni
- un modul DMA accepta cereri de transfer noi de la SPU-ul asociat si le trateaza
- un modul DMA se ocupa de transferul datelor pentru transferurile aflate in desfasurare
- arbitrul accepta cereri de transfer noi de la modulele DMA si le trateaza

Acestea sunt activitatile obligatorii. Pe langa acestea, fiecare entitate poate executa si alte activitati, in functie de implementarea dumneavoastra.

Viteza de transfer a datelor este de **1 byte/ciclu**. Asadar, o cerere pentru transferul a X bytes va ocupa un segment de inel pe durata a X cicli (transferul trebuie realizat fara intreruperi, dar nu trebuie sa inceapa imediat). Daca o cerere de transfer este transmisa modulului DMA pe durata unui ciclu C, transferul va putea incepe cel mai devreme la inceputul ciclului urmator (C+1). Daca un transfer incepe in ciclul A si dureaza B cicli, segmentul de inel corespunzator (ales de arbitru) va fi ocupat pe durata integrala a ciclilor A, A+1, ..., A+B-1, iar transferul se va incheia la sfarsitul ciclului A+B-1 (la inceputul ciclului A+B, segmentul de inel nu va mai fi ocupat de transferul respectiv).

In fiecare ciclu al duratei de desfasurare a unui transfer, modulul DMA al carui SPU a initiat transferul va transfera 1 byte. Transferul are loc in doua etape. Intai, modulul DMA trimite o comanda folosind Magistrala de Comenzi, catre modulul DMA catre care doreste sa transfere byte-ul respectiv (in cazul unui transfer de tip **put**) sau de la care doreste sa obtina byte-ul respectiv (in cazul unui transfer de tip **get**). Dupa transmiterea comenzii, modulul DMA realizeaza transferul propriu-zis. Magistrala de comenzi permite transmiterea unei singure comenzi la un moment dat (**acces exclusiv**), dar se pot transmite mai multe comenzi pe durata aceluiasi ciclu.

La sfarsitul fiecarui ciclu, SPU-urile, modulele DMA si arbitrul trebuie sa se sincronizeze (folosind o bariera reentranta).

2. Detalii de implementare

Tema va fi implementata folosind limbajul Python. Va trebui sa realizati un fisier numit **asc_t1.py**, care sa contina clasele **SPU_ASC_T1**, **DMA_Module_ASC_T1**, **Data_Arbiter_ASC_T1**, **Command_Bus_ASC_T1**. Aceste clase vor rebui sa extinda clasele **SPU**, **DMA_Module**, **Data_Arbiter** si **Command_Bus**, definite in fisierul **asc_t1_defs.py**. In acelasi fisier sunt definite multe alte clase si functii, folosite pentru testarea automata a claselor scrise de dumneavoastra (cititi mai multe detalii in seciunea **Testarea temei**). In continuare sunt descrise pe scurt principalele clase din fisierul **asc_t1_defs.py** pe care va trebui sa le intelegeti pentru a putea realiza tema (celelalte clase sunt folosite pentru testarea temei si nu este necesar sa intelegeti functionarea lor).

2.1. Clasa SPU

Clasa **SPU** extinde clasa **Thread** (din modulul **threading**). Constructorul clasei primeste urmatorii parametri:

- **id** -> identificatorul SPU-ului (un numar intre 0 si numarul de SPU-uri - 1)
- **memory_module** -> o instanta a clasei **Memory_Module** (definita in acelasi fisier), reprezentand modulul de memorie
- **program** -> o functie ce contine programul SPU-ului ; aceasta functie este stabilita de programul de test, iar SPU-ul trebuie neaparat sa o execute
- **controller** -> o instanta a clasei **Cycle_Controller** ; la sfarsitul fiecarui ciclu, SPU-ul trebuie sa apeleze metoda **spu_finished_cycle** a controller-ului

Functiile pe care va trebui sa le redefiniti voi in clasa **SPU_ASC_T1** care va extinde clasa **SPU** sunt urmatoarele:

- **write_to_memory(mem_addr, v)** -> scrie la adresa mem_addr din memorie valoarea v (va trebui sa folositi functiile modulului de memorie primit in constructor pentru a realiza scrierea)
- **read_from_memory(mem_addr)** -> intoarce valoarea de la adresa mem_addr din memorie (va trebui sa folositi functiile modulului de memorie primit in constructor pentru a realiza scrierea)
- **program_finished_cycle()** -> aceasta functie este apelata de catre programul executat de SPU, pentru a notifica SPU-ul ca s-a incheiat ciclul de executie (intrucat programul SPU-ului este un parametru de test la care nu aveti acces, fara aceasta functie un SPU nu ar putea sti cand se termina un ciclu)

2.2. Clasa DMA_Module

Clasa **DMA_Module** extinde clasa **Thread** (din modulul **threading**). Constructorul clasei primeste urmatorii parametri:

- **num_cycles** -> numarul de cicli cat dureaza executia
- **controller** -> o instanta a clasei **Cycle_Controller** ; la sfarsitul fiecarui ciclu, modulul DMA trebuie sa apeleze metoda **dma_module_finished_cycle** a controller-ului

Functiile pe care va trebui sa le redefiniti sunt urmatoarele: **get**, **put**, **status** (semnificatia lor a fost mentionata anterior) si **run**. Functia status trebuie sa intoarca un tuplu cu 3 elemente: (status, sens, inel). **Status**-ul este unul din cele 4 valori definite in **asc_t1_defs.py** (**TRANSFER_STATUS_INVALID_ID, TRANSFER_STATUS_NOT_STARTED, TRANSFER_STATUS_RUNNING, TRANSFER_STATUS_FINISHED**). In cazul in care status-ul este **TRANSFER_STATUS_RUNNING**, **sens** si **inel** vor reprezenta sensul si inelul pe care se desfasoara transferul (daca **status**-ul are una din celelalte 3 valori, **sens** si **inel** pot lua orice valoare).

2.3. Clasa Data_Arbiter

Clasa **Data_Arbiter** extinde clasa **Thread** (din modulul **threading**). Constructorul clasei primeste urmatorii parametri:

- **dma_module_list** -> lista cu modulele DMA, in ordinea numerotarii (0, 1, 2, ..., N-1)
- **K** -> numarul de inele pe sens
- **num_cycles** -> numarul de cicli cat dureaza executia
- **controller** -> o instanta a clasei **Cycle_Controller** ; la sfarsitul fiecarui ciclu, arbitrul de date trebuie sa apeleze metoda **data_arbiter_finished_cycle** a controller-ului

Va trebui sa redefiniti functia **run**.

2.4. Clasa Command_Bus

Clasa **Command_Bus** corespunde **Magistralei de Comenzi**. Constructorul acestei clase primeste urmatorii parametri:

- **dma_module_list** -> lista cu modulele DMA, in ordinea numerotarii (0, 1, 2, ..., N-1)

Va trebui sa redefiniti functia **send_command(src, dest, cmd)**, unde:

- **src** = modulul DMA sursa
- **dest** = modulul DMA destinatie
- **cmd** = comanda (**put** sau **get**)

Functia **send_command** redefinita de voi in clasa **Command_Bus_ASC_T1** va trebui sa apeleze functia **send_command** din clasa **Command_Bus** (aici se mentine un contor cu numarul de apeluri si se testeaza sincronizarea accesului la magistrala de comenzi).

2.5. Clasa Memory_Module

Clasa M**emory_Module** ofera functiile **read(addr)** pentru a citi continutul memoriei de la adresa **addr** si **write(addr, v)**, pentru a scrie valoarea **v** la **adresa addr**.

Un schelet pentru realizarea temei il gasiti in fisierul **asc_t1_skeleton.py**. Rolul functiei **init_all** este descris in sectiunea urmatoare.

3. Testarea temei

Tema va fi testata automat. Programul care lanseaza in executie testele se gaseste in fisierul **asc_t1_runtest.py**. Pentru a functiona, trebuie sa puneti in acelasi director programul de test, impreuna cu fisierele **asc_t1_defs.py**

(mentionat mai sus) si **asc_t1.py** (care contine implementarile voastre ale celor 4 clase). In fisierul **asc_t1.py** va trebui sa definiti si o functie **init_all**. Ea va fi apelata de functia de test inainte de a incepe testul si va permite sa initializati eventualele variabile globale de care aveti nevoie. Parametrul functiei este **N** (numarul de SPU-uri).

Programul de test verifica urmatoarele:

- Executia corecta a instructiunului programelor executate de SPU-uri: se intorc id-uri de cereri distincte si status-uri valide
- Sincronizarea corecta la sfarsitul fiecarui ciclu (clasa **Cycle_Controller** realizeaza aceasta verificare)
- Sincronizarea corecta a accesului la obiectele partajate (modulele de memorie, magistrala de comenzi) - se folosesc clasele **Shared_Object** si **Sync_Scheduler** ; clasa **Sync_Scheduler** intarzie in mod intentionat operatiile de read/write la obiecte de tip Shared_Object, pentru a pune in evidenta evetualele erori de sincronizare => acest lucru mareste timpul de executie
- Fiecare transfer a inceput dupa ce a fost initiat (in ciclul urmator sau mai tarziu), a ocupat un interval continuu de timp (avand durata corespunzatoare) o portiune a aceluiasi inel, de pe acelasi sens
- Nu s-au suprapus transferuri pe acelasi inel, sens si in acelasi moment de timp
- Continutul memoriei la sfarsitul fiecarui ciclu poate fi obtinut pe baza continutului memoriei de la ciclul anterior si operatiile de citire/scriere din cadrul ciclului respectiv

Programul de test nu poate detecta toate erorile de sincronizare. El incearca sa detecteze erorile de sincronizare la modulele de memorie si magistrala de comenzi, insa nu este sigur ca va reusi. De asemenea, programul de test nu poate detecta erorile de sincronizare la variabilele partajate din interiorul claselor voastre. Din acest motiv, detectarea erorilor de sincronizare se va realiza si prin **inspectie vizuala a codului**.

Tema contine si un element de performanta: **timpul total de asteptare**. Timpul de asteptare pentru un transfer este egal cu diferenta dintre ciclul in care acesta incepe si ciclul urmator celui in care acesta a fost initiat. De exemplu, daca un transfer este initiat in ciclul 3, iar executia acestuia incepe in ciclul 6, timpul deasteptare este 6-4=2 (deoarece transferul ar fi putut incepe cel mai devreme la ciclul 4). Timpul total de asteptare este egal cu suma timpilor de asteptare pentru toate transferurile. Va exista o limita superioara pentru timpul total de asteptare pentru un test (stabilita in urma executiei unui program "etalon", care nu optimizeaza nimic, dar nici nu iroseste cicli).

4. Notarea temei

Exista 4 tipuri de erori posibile in aceasta tema, ordonate in functie de gravitatea lor:

- exceptii la executia programului
- erori fatale -> nu se respecta cerintele functionale ale temei (aceste erori sunt detectate automat)
- erori de sincronizare -> accesul la variabile partajate si sincronizarea la sfarsit de ciclu (aceste erori vor fi detectate atat automat, cat si prin inspectia vizuala a codului sursa)
- probleme de performanta -> timpul total de asteptare este prea mare (se detecteaza automat)

Depunctarile pentru exceptii si erorile fatale vor fi foarte mari, in conditiile in care aveti la dispozitie programul de test care le poate pune in evidenta inca din etapa de realizare a temei. Temele realizate partial (nu se implementeaza toate clasele) vor fi punctate foarte putin, deoarece nu respecta specificatiile functionale (sunt echivalente cu erorile fatale).