***Министерство образования Республики Беларусь***

***Учреждение образования***

***«Брестский государственный технический университет»***

***Кафедра ИИТ***

**Лабораторная работа №6**

**По дисциплине АОИС за III семестр**

**Тема: «Моделирование ассоциативной памяти»**

**Выполнил:**

Студент группы ИИ-15 (1)

2-го курса

Волк И. А.

**Проверил:**

Михно Е. В.

Брест 2018

**Цель работы:** Изучить основные принципы реализации ассоциативной памяти; построить и проверить программную модель, обеспечивающую выполнение операций поиска и выборки информации из ассоциативной памяти.

**Задание:** Написать на ЯВУ программу, которая моделирует один вид ассоциативной памяти согласно варианту в виде таблицы данных (таблицу можно брать из лабораторной работы №1). Отобразить на экране данные, хранимые в памяти, и содержимое регистров, реализовать операции сравнения согласно варианту над ключевым полем данных.

|  |  |  |  |
| --- | --- | --- | --- |
| **Вариант** | **Наличие регистра маски** | **Способы сравнения** | **Тип ассоциативной памяти** |
| 5 | + | = | с последовательной обработкой разрядов |

Код программы:

#include <iostream>

#include <array>

#include <algorithm>

template<int NUM\_OF\_BITS = 10, bool DEFAULT\_VALUE = false>

class Register

{

protected:

std::array<bool, NUM\_OF\_BITS> memory;

Register()

{

std::fill(memory.begin(), memory.end(), DEFAULT\_VALUE);

}

public:

void reboot()

{

std::fill(memory.begin(), memory.end(), DEFAULT\_VALUE);

}

bool& operator[](const int id)

{

return memory[id];

}

int size() const

{

return NUM\_OF\_BITS;

}

void fill(const std::array<bool, NUM\_OF\_BITS>& newMemory)

{

for (int i = 0; i < NUM\_OF\_BITS; i++)

memory[i] = newMemory[i];

}

virtual ~Register() {}

};

template<int N = 10>

class AIR : public Register<N, true> // address indicator register

{};

template<int N = 10>

class NSR : public Register<N, true> // nested search register

{};

template<int L = 8>

class AAR : public Register<L, false> // associative attribute register

{};

template<int L = 8>

class MR : public Register<L, true> // mask register

{};

template<int L = 8>

class MemoryWord : public Register<L, false>

{

bool isFilled;

public:

MemoryWord()

: Register< L, false>(), isFilled(false) {}

bool filled() const

{

return isFilled;

}

void clean()

{

isFilled = false;

}

void setFilled()

{

isFilled = true;

}

};

template<int N = 10, int L = 8>

class MemoryArray

{

std::array<MemoryWord<L>, N> cells;

public:

MemoryWord<L>& operator[](const int id)

{

return cells[id];

}

};

template<int N = 10>

class MRD // Multiple Response Device

{

AIR<N> \* air;

NSR<N> \* nsr;

public:

MRD(AIR<N> \*air, NSR<N> \*nsr)

: air(air), nsr(nsr) {}

void AIRtoNSR()

{

for (int i = 0; i < N; i++)

nsr->operator[](i) = air->operator[](i);

}

void NSRtoAIR()

{

for (int i = 0; i < N; i++)

air->operator[](i) = nsr->operator[](i);

}

std::array<bool, N> getAIR()

{

std::array<bool, N> result;

for (int i = 0; i < N; i++)

result[i] = air->operator[](i);

return result;

}

void rebootAIR()

{

air->reboot();

}

};

template<int N = 10>

class CompEl // comparison element

{

AIR<N> \* air;

bool elToCompare;

public:

CompEl(AIR<N> \* air)

: air(air) {}

void setEl(bool el)

{

elToCompare = el;

}

void compare(const std::array<bool, N>& cut)

{

for (int i = 0; i < N; i++)

{

air->operator[](i) &= elToCompare == cut[i];

}

}

};

template<int N = 10, int L = 8>

class CU // Control unit

{

int iteration;

MR<L>\* mr;

AAR<L> \* aar;

MemoryArray<N, L> \* memory;

MRD<N> \* mrd;

CompEl<N> \* compEl;

public:

CU(MR<L> \* mr, AAR<L> \* aar,

MemoryArray<N, L> \* memory,

MRD<N> \* mrd, CompEl<N> \* compEl)

: iteration(-1), mr(mr), aar(aar), memory(memory), mrd(mrd), compEl(compEl)

{}

bool useCompEl()

{

if (iteration == L)

{

iteration = -1;

return false;

}

if (iteration == -1)

{

std::array<bool, N> cut;

compEl->setEl(1);

for (int i = 0; i < N; i++)

cut[i] = memory->operator[](i).filled();

compEl->compare(cut);

iteration++;

return true;

}

if (mr->operator[](iteration) == false)

{

iteration++;

return useCompEl();

}

compEl->setEl(aar->operator[](iteration));

std::array<bool, N> cut;

for (int i = 0; i < N; i++)

cut[i] = memory->operator[](i).operator[](iteration);

compEl->compare(cut);

iteration++;

return true;

}

void fillAAR(std::array<bool, L>& word)

{

aar->fill(word);

}

void fillMR(std::array<bool, L>& mask)

{

mr->fill(mask);

}

void rebootAIR()

{

mrd->rebootAIR();

}

void moveAIRtoNSR()

{

mrd->AIRtoNSR();

}

void moveNSRtoAIR()

{

mrd->NSRtoAIR();

}

int findEmptyPlace()

{

int emptyPlace = -1;

for(int i = 0; i < N; i++)

if (!memory->operator[](i).filled())

{

emptyPlace = i;

break;

}

return emptyPlace;

}

void fillPlace(int place, const std::array<bool, L>& word)

{

memory->operator[](place).fill(word);

memory->operator[](place).setFilled();

}

int remove(std::array<bool, N> sRes)

{

int count = 0;

for(int i = 0; i < N; i++)

if (sRes[i] == true)

{

memory->operator[](i).clean();

count++;

}

return count;

}

std::array<bool, N> getResult()

{

return mrd->getAIR();

}

};

template<int N = 10, int L = 8>

class CAM // Content-addressable memory

{

NSR<N> nsr; // nested search register

MR<L> mr; // mask register

AAR<L> aar; // associative attribute register

MemoryArray<N, L> memory;

MRD<N> mrd; // Multiple Response Device

AIR<N> air; // address indicator register

CompEl<N> compEl; // comparison element

CU<N, L> cu; // Control unit

public:

CAM()

: nsr(), mr(), aar(), memory(), mrd(&air, &nsr), air(), compEl(&air),

cu(&mr, &aar, &memory, &mrd, &compEl) {}

bool add(std::array<bool, L> word)

{

int emptyPlace = cu.findEmptyPlace();

if (emptyPlace == -1)

return false;

cu.fillPlace(emptyPlace, word);

return true;

}

std::array<bool, N> search(std::array<bool, L> word, std::array<bool, L> mask)

{

cu.fillAAR(word);

cu.fillMR(mask);

while (cu.useCompEl());

cu.moveAIRtoNSR();

std::array<bool, N> result = cu.getResult();

cu.rebootAIR();

return result;

}

std::array<bool, N> nSearch(std::array<bool, L> word, std::array<bool, L> mask)

{

cu.fillAAR(word);

cu.fillMR(mask);

cu.moveNSRtoAIR();

while (cu.useCompEl());

cu.moveAIRtoNSR();

std::array<bool, N> result = cu.getResult();

cu.rebootAIR();

return result;

}

int remove(std::array<bool, L> word, std::array<bool, L> mask)

{

std::array<bool, N> sRes = search(word, mask);

return cu.remove(sRes);

}

};

#define out(a) for(auto x : a)\

{\

std::cout << x << " ";\

}\

std::cout << std::endl;

int main()

{

const int wn = 2;

const int wl = 4;

CAM<wn, wl> cam;

//test1

cam.add({ 1, 0, 0, 1 });

cam.add({ 1, 1, 0, 1 });

auto result = cam.search({ 1, 1, 0, 1 }, { 1, 0, 1, 1 });

std::cout << "Search test1: ";

out(result);

result = cam.nSearch({ 1, 1, 0, 1 }, { 1, 1, 1, 1 });

std::cout << "Nested search test1: ";

out(result);

std::cout << std::endl;

//test2

cam.remove({1, 1, 0, 1}, {1, 1, 1, 1});

result = cam.search({ 1, 1, 0, 1 }, { 1, 0, 1, 1 });

std::cout << "Search test2: ";

out(result);

result = cam.nSearch({ 1, 1, 0, 1 }, { 1, 1, 1, 1 });

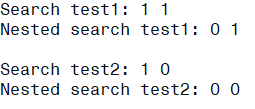
std::cout << "Nested search test2: ";

out(result);

std::cin.get();

}

Результат выполнения:



Вывод: Изучил основные принципы реализации ассоциативной памяти; построил и проверил программную модель, обеспечивающую выполнение операций поиска и выборки информации из ассоциативной памяти.