Object-Oriented Programming VL

Laborprotokoll

Beispiel 3

Günther Neuwirth, Matr. Nr.0626638
e0626638@student.tuwien.ac.at
Manuel Mausz, Matr. Nr.0728348
manuel-tu@mausz.at

Wien, am 29. Mai 2009

Inhaltsverzeichnis

1 Aufgabenstellung - Beispiel 3

Beispielangaben zu
OOP (Objekt-Orientierte Programmierung)
SS 2009

Beispiel 3

Relevante Themen:

- Untertyp-Beziehungen
- Zusicherungen
- Wiederverwendung von Code

Design und Implementierung:

Schreiben Sie in C++ einen CPU-Simulator myCPU. Das Programm liest als ersten Kommandozeilenparameter den Dateinamen des auszuführenden Programmes ein, welches die Programminstruktionen als Assemblercode enthält. Der zweite Kommandozeilenparameter, welcher optional ist, erlaubt die Angabe einer Defaultbelegung der ersten Elemente des Programmspeichers. Wird kein Speicherbelegungsfile angegeben, so haben die Elemente des Programmspeichers einen undefinierten Wert.

Die Synopsis des Programms ist folgende: myCPU -c ⟨programfile⟩ [-m ⟨memfile⟩]

Der zu implementierende Prozessor hat 256 generell verwendbare Register, welche als R0 bis R255 ansprechbar sind. Das Register R0 stellt zugleich den Programmzähler dar, kann aber auch direkt modifiziert werden. Sollte der Programmzähler (R0) auf einen ungültigen Wert gesetzt werden, so soll mit einer Fehlermeldung abgebrochen werden. Der aktuelle Wert des Programmzählers entspricht der Zeilennummer (beginnend mit Eins) im Programmfile.
Weiters besitzt der Prozessor ein ZERO-Flag und ein SIGN-Flag, welche durch einen speziellen Test-Befehl gesetzt werden. Die Adressierung der Instruktionen ist nur über Register und Adresslabel möglich (siehe Beschreibung von ADDR). Konstanten können daher nur über die Initialisierung des Programmspeichers benutzt werden.

Format der **Programmdatei**:
Die Programmdatei enthält pro Zeile maximal eine Assemblerinstruktion (kann auch Leerzeilen enthalten, welche zu ignorieren sind). Whitespaces (Leerzeichen, Tabulatoren) sind zu ignorieren. Zeilen, welche mit „#“ beginnen, sind zu ignorieren (Kommentarzeilen).

Für die Beschreibung der Assemblerinstruktionen gelten folgende Konventionen:

- REGx steht für ein beliebiges Register. Am Beginn der Programmausführung haben alle Register den Wert 0 (Null).

- ADDR ist ein Labelname, welcher einer Sprungadresse im Programmcode entspricht. Sollte ein Labelname in einem Sprungbefehl benutzt werden, welcher nicht definiert wurde, so ist mit einer Fehlermeldung abzubrechen.
- DEV entspricht einer vordefinierten symbolischen Adresse für ein Gerät (Device). So werden beispielsweise WDEZ für Dezimalausgabe und WHEX für Hexadezimalausgabe verwendet.

Folgende **Assemblerinstruktionen** sind zu unterstützen:

inc REG1
: Erhöht den Wert des Registers um Eins: REG1 = REG1 + 1
Beispiel: inc R2

dec REG1
: Erniedrigt den Wert des Registers um Eins: REG1 = REG1 - 1

add REG1, REG2, REG3
: Führt folgende Addition durch: REG1 = REG2 + REG3
Beispiel: add R4, R4, R7

sub REG1, REG2, REG3
: Führt folgende Subtraktion durch: REG1 = REG2 - REG3

mul REG1, REG2, REG3
: Führt folgende Multiplikation durch: REG1 = REG2 * REG3

div REG1, REG2, REG3
: Führt folgende Addition durch: REG1 = REG2 / REG3. Sollte REG3 den Wert 0 (Null) haben ist mit einer Fehlermeldung abzubrechen.

load REG1, REG2
: Ladet den Inhalt des Hauptspeichers mit Index REG2 in das Register REG1.

store REG1, REG2
: Speicher den Inhalt des Registers REG1 in den Hauptspeichers auf Index REG2.

test REG1
: Dieser Befehl testet den Inhalt von REG1 und setzt das ZERO-Flag genau dann, wenn REG1 = 0 hat. Das SIGN-Flag wird genau dann gesetzt, wenn der Wert des Registers negativ ist. Andere Instruktionen haben keinen Einfluss auf den Wert des ZERO- und das SIGN-Flags.

label Name :
: Definiert ein Adresslabel „Name“ auf das direkt mit Sprungbefehlen verzweigt werden kann.
Beispiel: label LabelX

jumpa ADDR
: Der Programmzähler wird ADDR gesetzt.
Beispiel: jumpa LabelX

jumpz ADDR
: Der Programmzähler wird auf ADDR gesetzt genau dann wenn das ZERO-Flag gesetzt ist.

jumps ADDR
: Der Programmzähler wird auf ADDR gesetzt genau dann wenn das SIGN-Flag gesetzt ist.

write DEV, REG1

Schreibt den Inhalt von Register REG1 auf das Ausgabegerät DEV, wobei DEV für die Gerätenamen WDEZ oder WHEX steht. Dabei wird die Zahl im Dezimal- bzw. Hexformat auf die Standardausgabe geschrieben, gefolgt von einem Zeilenumbruch.
Beispiel: write WDEZ, R7

Der **Programmspeicher** ist als ein dynamisch wachsendes Array zu implementieren, wobei auf die Speicherelemente mit den Instruktionen load und store zugegriffen wird. Das erste Element im Programmspeicher hat den Index 0 (Null).

Das **Speicherbelegungsfile** dient zur Initialisierung der ersten Elemente im Programmspeicher. Das Speicherbelegungsfile enthält pro Zeile maximal eine Zahl, wobei der Inhalt der N-ten Zeile (startend mit Eins) in den Programmspeicher mit Index (N-1) geschrieben wird. Leerzeilen bedeuten, dass das entsprechende Speicherelement nicht initialisiert wird.
Da die Assemblerinstruktionen keine Verwendung von Konstanten als Parameter unterstützen, müssen entsprechende Konstanten im Speicherbelegungsfile definiert werden. Steht beispielsweise im Speicherbelegungsfile in der zweiten Zeile der Wert 20, so kann dieser mit folgendem Programmstück in das Register R3 zugewiesen werden (unter der Annahme, dass Register R1 noch den Default-Wert 0 hat):

```
#   copy the value zero into R2:
add R2,R1,R1
inc R2
#   read memory[R2]:
load R3, R2
```

Erstellen Sie dazu eine Klasse CCPU für die CPU, eine abstrakte Klasse CInstruction für die Assemblerbefehle, eine abstrakte Displayklasse CDisplay für die Ausgabe, eine Klasse CMem für den Programmspeicher, eine Klasse CProgram für den Programmspeicher, sowie eine Klasse CDat für den Datentyp der Assemblerinstruktionen.

Von CInstruction sind die einzelnen Assemblerbefehle abzuleiten. Von CDisplay sind die jeweiligen Displays abzuleiten.
Die Klasse CDat implementiert alle Rechenoperationen, welche für die jeweiligen Assemblerinstruktionen benötigt werden und kann eine aus dem Speicherbelegungsfile geladene Zeile parsen um den Wert zuzuweisen. Für dieses Beispiel ist als Datentyp von CDat der C++-Datentyp int zu verwenden (keine speziellen Zahlenformate bzw. Rechenregeln). Jedoch sollte das Programm so entworfen werden, dass durch alleiniges Umschreiben der Klasse CDat mit einem beliebigen anderen Datentyp gearbeitet werden kann (natürlich vorausgesetzt, dass dann auch das Speicherbelegungsfile im kompatiblen Format ist).

2 Beispiel 3

2.1 Design

Abbildung 1 zeigt das Klassendiagramm der Aufgabe.

Als Datentyp für Register und Hauptspeicher der CPU wurde ein allgemeines Template implementiert, welche andere Datentypen oder Klasse umhüllen kann. Zudem wurden die gängigsten Operatoren definiert, die man sich von einem Datentyp erwarten kann. Gemäß der Aufgabenstellung wurde der Datentyp CDat als umhüllter Integer-Wert definiert.

Der Hauptspeicher der CPU wurde als abgeleitetes Template CVectorMem des Templates std::vector<T, Allocator> implementiert, damit dieses um die Methode initialize und dump erweitert werden konnten. Erstere dient zur Initialisierung des Hauptspeichers bzw. Füllung des Vectors, zweiteres zur Debugausgabe. Der Datentyp CMem wurde via typedef als CVectorMem mit dem Templateargument CDat definiert. Somit enthält der Hauptspeicher nur Elemente des Typs CDat.

Die geforderten zwei Display-Klassen CDisplayWDEZ und CDisplayWHEX wurden von der abstrakten Klasse CDisplay abgeleitet, welche selbst als Template CDisplayT mit dem Templateargument CDat definiert ist.

Der Programmspeicher CProgram wurde vom Template std::vector<CInstruction *> abgeleitet. Somit ist diese Klasse, ähnlich wie CMem, selbst ein Vector. Zudem wurden weitere Methoden hinzugefügt. Die Methode compile dient zum Kompilieren bzw. Umwandeln des Syntax der Programmdatei in Instanzen der Klasse CInstruction, die später von der CPU ausgeführt werden. Zur Umwandlung werden bei der Instantiierung von CProgram sämtliche erlaubten Instruktionen (also Instanzen von CInstruction) in ein std::set eingefügt. Im Zuge der Methode compile wird durch Iterieren die jeweilige Instruktion gesucht und durch Anwendung des Designpattern *Factory method pattern* die Instruktion dupliziert und in CProgram gespeichert. Labels werden zwecks Effizienz in einer std::map festgehalten.

Die jeweiligen unterstützen Instruktionen wurden wie gefordert von der Klasse CInstruction abgeleitet. CInstruction fordert die Implementierung einer Methode compile, die während des parsens der Programmdatei von CProgram aufgerufen werden, als auch eine Methode execute, die im Zuge der Ausführung des Programms von CCPU aufgerufen werden.

Die eigentliche CPU wird durch die Klasse CCPU implementiert. Diese dient hauptsächlich als Container für die einzelnen, notwendigen Teile (Hauptspeicher, Programmspeicher, Displays, ...) und besitzt daher entsprechend viele get- und set-Funktionen. Die Methode run dient zur Ausführung des Programms und ist eine Schleife, die über die Instruktionen iteriert und die Methode execute aufruft.

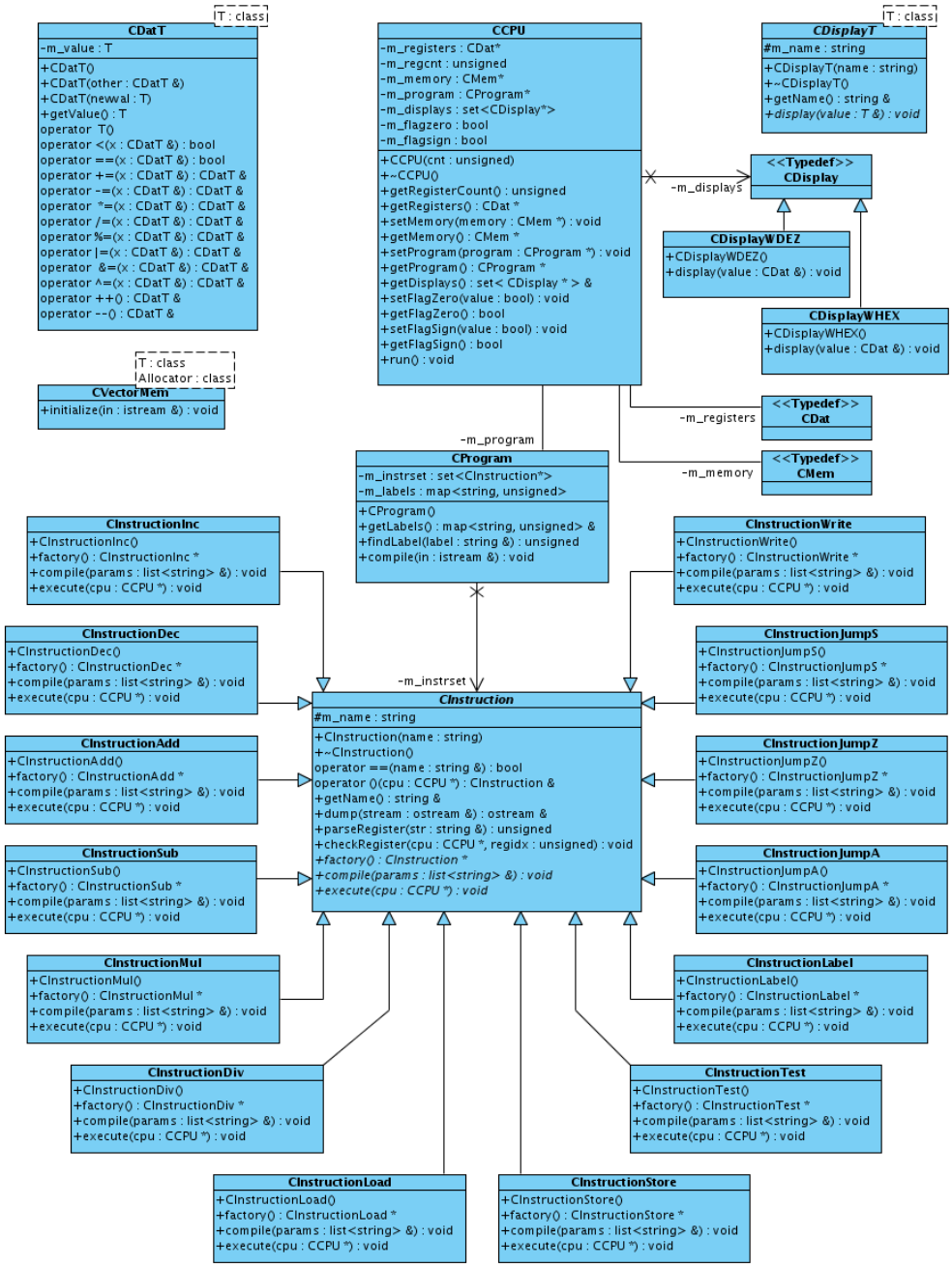

Abbildung 1: Klassendiagramm 1

2.2 Verwaltung der Ressourcen

Alle Objekte, die im Konstruktor alloziert werden, werden im Destruktor wieder freigegeben. Die Objekte, die über die *Factory method pattern* alloziert werden, werden im Zuge des Destruktor des Vectors bzw. von CProgram wieder freigegeben.

2.3 Fehlerbehandlung

Es wurden keine eigenen Exceptions eingeführt. Statt dessen werden Exceptions meist in Exceptions des Typs std::runtime_error umgewandelt. Da die Hierarchie nicht sehr tief ist, fängt lediglich CPropram::compile Exceptions des Typs std::runtime_error, um zusätzliche Information an den ursprünglichen Aufrufer der Methode weiterzugeben. Dies geschieht ebenfalls per Exception des Typs std::runtime_error.

2.4 Implementierung

Siehe Punkt 2.1 und Abbildung 1 sowie Punkt 4.
Es wurde viel mit Templates gearbeitet.

3 Projektverlauf

3.1 Probleme und Fallstricke

Abgesehen von den mittlerweile üblichen Schwierigkeiten die Angabe bzw. die Gedanken dessen Verfassers verstehen zu wollen, arbeitete das erste Design noch vermehrt mit Text. Unter anderem hatte die Methode CInstruction::execute einen Parameter des Typs std::string, was dazu führte, dass die Instruktion bei jeder Aufführung (zum Beispiel durch Jumps) erneut geparsed werden musste.

3.2 Arbeitsaufwand

Entwicklungsschritt / Meilenstein	Arbeitsaufwand
Erstes Design	3 Stunden
Implementierung	3 Tage
Anpassung des Designs und der Implementierung	4 Stunden
Dokumentation (Doxygen) und Überprüfung aller Anforderungen gemäß der Programmierrichtlinien	4 Stunden
Erstellung des Protokolls	3 Stunden

4 Listings

4.1 mycpu.cpp

```
/**
 * @module mycpu
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  mycpu executes a programfile (in simple assembler) by parsing the
 *         programfile first. This creates a vector of instructions, which will
 *         be executed in linear order (except jumps) afterwards. In order to
 *         initialize the memory of the cpu before execution an optional
 *         memoryfile can be passed as commandline option.
 * @date   13.05.2009
 * @par Exercise
 *      4
 */

#include <boost/program_options.hpp>
#include <iostream>
#include <fstream>
#include <stdexcept>
#include <stdlib.h>
#include "ccpu.h"
#include "cmem.h"
#include "cprogram.h"

using namespace std;
namespace po = boost::program_options;

/**
 * @func   main
 * @brief  program entry point
 * @param  argc  standard parameter of main
 * @param  argv  standard parameter of main
 * @return 0 on success, not 0 otherwise
 * @globalvars none
 * @exception  none
 * @conditions none
 *
 * parse commandline options, create and initialize memory,
 * create cprogram instance, which parses the programfile and
 * execute CCPU::run()
 * On error print error message to stderr.
 * Unknown commandline options will print a usage message.
 */
int main(int argc, char* argv[])
{
  string me(argv[0]);

  /* define commandline options */
  po::options_description desc("Allowed options");
  desc.add_options()
    ("help,h",     "this help message")
    ("compile,c", po::value<string>(), "input programfile")
    ("memory,m",  po::value<string>(), "input memoryfile");

  /* parse commandline options */
  po::variables_map vm;
  try
  {
    po::store(po::parse_command_line(argc, argv, desc), vm);
    po::notify(vm);
```

```
  }
  catch(po::error& ex)
  {
    cerr << me << ": Error: " << ex.what() << endl;
  }

  /* print usage upon request or missing params */
  if (vm.count("help") || !vm.count("compile"))
  {
    cout << "Usage: " << me << " -c <programfile> [-m <memoryfile>]" << endl;
    cout << desc << endl;
    return 0;
  }

  /* create memory and optionally initialize memory from file */
  CMem memory;
  if (vm.count("memory"))
  {
    string memoryfile(vm["memory"].as<string>());
    ifstream file(memoryfile.c_str(), ios::in);
    if (!file.is_open())
    {
      cerr << me << ": Unable to open memoryfile '" << memoryfile << "' for reading
          ." << endl;
      return 1;
    }

    try
    {
      memory.initialize(file);
      file.close();
    }
    catch(runtime_error& ex)
    {
      file.close();
      cerr << me << ": Error while reading from memoryfile:" << endl
           << "  " << ex.what() << endl;
      return 1;
    }

#if DEBUG
    memory.dump(cerr);
#endif
  }

  /* create program instance */
  CProgram program;
  string programfile(vm["compile"].as<string>());
  ifstream file(programfile.c_str(), ios::in);
  if (!file.is_open())
  {
    cerr << me << ": Unable to open programfile '" << programfile << "' for reading
        ." << endl;
    return 1;
  }

  try
  {
    program.compile(file);
    file.close();
  }
  catch(runtime_error& ex)
  {
    file.close();
```

```
    cerr << me << ": Error while compiling programfile:" << endl
         << "  " << ex.what() << endl;
    return 1;
  }

#if DEBUG
  program.dump(cerr);
#endif


  /* create cpu and execute the program */
  try
  {
    CCPU cpu(256);
    cpu.setMemory(&memory);
    cpu.setProgram(&program);
    cpu.run();
#if DEBUG
    //cpu.dumpRegisters(cerr);
#endif
  }
  catch(runtime_error& ex)
  {
    cerr << me << ": Error while executing program:" << endl
         << "  " << ex.what() << endl;
#if DEBUG
    memory.dump(cerr);
#endif
    return 1;
  }

  return 0;
}

/* vim: set et sw=2 ts=2: */
```

4.2 cdat.h

```
/**
 * @module cdat
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Datatype template and datatype definition for CCPU and CMem
 * @date   10.05.2009
 */

#ifndef CDAT_H
#define CDAT_H 1

#include <boost/operators.hpp>
#include <iostream>

/**
 * @class CDatT
 *
 * Datatype template for CCPU and CMem.
 */
template <class T>
class CDatT
  : boost::operators<CDatT<T> >
{
  public:
    /**
     * @method CDatT
     * @brief  Default ctor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  bad_alloc
     * @conditions none
     */
    CDatT()
    {}

    /**
     * @method ~CDatT
     * @brief  Default dtor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    ~CDatT()
    {}

    /**
     * @method CDatT
     * @brief  Copy constructor for CDatT
     * @param  other  reference to CDatT which will be copied
     * @return -
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    CDatT(const CDatT& other)
      : m_value(other.m_value)
    {}

    /**
```

```
 * @method CDatT
 * @brief  Copy constructor for int
 * @param  newval  new value for CDatT
 * @return -
 * @globalvars none
 * @exception  none
 * @conditions none
 */
CDatT(T newval)
  : m_value(newval)
{}

/**
 * @method getValue
 * @brief  returns value of CDatT
 * @param  -
 * @return value of CDatT
 * @globalvars none
 * @exception  none
 * @conditions none
 */
T getValue() const
{
  return m_value;
}

/**
 * @method operator T
 * @brief  convert to T
 * @param  -
 * @return T
 * @globalvars none
 * @exception  none
 * @conditions none
 */
operator T()
{
  return m_value;
}

/**
 * @method operator<
 * @brief  implementation of operator <
 * @param  x  reference to CDatT
 * @return true if cdat is less than object x
 * @globalvars none
 * @exception  none
 * @conditions none
 */
bool operator<(const CDatT& x) const
{
  return m_value < x.m_value;
}

/**
 * @method operator==
 * @brief  implementation of operator ==
 * @param  x  reference to CDatT
 * @return true if cdat equals object x
 * @globalvars none
 * @exception  none
 * @conditions none
 */
bool operator==(const CDatT& x) const
```

```
{
  return m_value == x.m_value;
}

/**
 * @method operator+=
 * @brief implementation of operator +=
 * @param x reference to CDatT
 * @return refecence to CDatT
 * @globalvars none
 * @exception none
 * @conditions none
 */
CDatT& operator+=(const CDatT& x)
{
  m_value += x.m_value;
  return *this;
}

/**
 * @method operator-=
 * @brief implementation of operator -=
 * @param x reference to CDatT
 * @return refecence to CDatT
 * @globalvars none
 * @exception none
 * @conditions none
 */
CDatT& operator-=(const CDatT& x)
{
  m_value -= x.m_value;
  return *this;
}

/**
 * @method operator*=
 * @brief implementation of operator *=
 * @param x reference to CDatT
 * @return refecence to CDatT
 * @globalvars none
 * @exception none
 * @conditions none
 */
CDatT& operator*=(const CDatT& x)
{
  m_value *= x.m_value;
  return *this;
}

/**
 * @method operator/=
 * @brief implementation of operator /=
 * @param x reference to CDatT
 * @return refecence to CDatT
 * @globalvars none
 * @exception none
 * @conditions none
 */
CDatT& operator/=(const CDatT& x)
{
  m_value /= x.m_value;
  return *this;
}
```

```
/**
 * @method operator%=
 * @brief implementation of operator %=
 * @param x reference to CDatT
 * @return refecence to CDatT
 * @globalvars none
 * @exception none
 * @conditions none
 */
CDatT& operator%=(const CDatT& x)
{
  m_value %= x.m_value;
  return *this;
}

/**
 * @method operator|=
 * @brief implementation of operator |=
 * @param x reference to CDatT
 * @return refecence to CDatT
 * @globalvars none
 * @exception none
 * @conditions none
 */
CDatT& operator|=(const CDatT& x)
{
  m_value |= x.m_value;
  return *this;
}

/**
 * @method operator&=
 * @brief implementation of operator &=
 * @param x reference to CDatT
 * @return refecence to CDatT
 * @globalvars none
 * @exception none
 * @conditions none
 */
CDatT& operator&=(const CDatT& x)
{
  m_value &= x.m_value;
  return *this;
}

/**
 * @method operator^=
 * @brief implementation of operator ^=
 * @param x reference to CDatT
 * @return refecence to CDatT
 * @globalvars none
 * @exception none
 * @conditions none
 */
CDatT& operator^=(const CDatT& x)
{
  m_value ^= x.m_value;
  return *this;
}

/**
 * @method operator++
 * @brief implementation of operator ++
 * @param -
```

```
     * @return refecence to CDatT
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    CDatT& operator++()
    {
      m_value++;
      return *this;
    }

    /**
     * @method operator--
     * @brief  implementation of operator --
     * @param  -
     * @return refecence to CDatT
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    CDatT& operator--()
    {
      m_value--;
      return *this;
    }

    /**
     * @method operator<<
     * @brief  Shift/output operator for outputstream
     * @param  stream  reference to outputstream
     * @param  cdat    object which will be printed to stream
     * @return reference to outputstream
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    friend std::ostream& operator<<(std::ostream& stream, CDatT cdat)
    {
      stream << cdat.m_value;
      return stream;
    }

    /**
     * @method operator>>
     * @brief  Shift/read operator for inputstream
     * @param  stream  reference to inputstream
     * @param  cdat    reference to object which will be read from stream
     * @return reference to inputstream
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    friend std::istream& operator>>(std::istream & stream, CDatT& cdat)
    {
      stream >> cdat.m_value;
      return stream;
    }

  private:
    /* members */
    T m_value;
};

/**
```

```
 * @class CDat
 *
 * Datatype for CCPU and CMem
 */
typedef CDatT<int> CDat;

#endif

/* vim: set et sw=2 ts=2: */
```

4.3 cmem.h

```
/**
 * @module cmem
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Memory template and memory definition for CCPU
 * @date   10.05.2009
 */

#ifndef CMEM_H
#define CMEM_H 1

#include <vector>
#include <istream>
#include <sstream>
#include <stdexcept>
#include <boost/lexical_cast.hpp>
#ifdef DEBUG
# include <iostream>
# include <iomanip>
#endif
#include "cdat.h"

/**
 * @class CVectorMem
 *
 * Extends std::vector template for use as memory for CCPU.
 */
template <class T, class Allocator=std::allocator<T> >
class CVectorMem
  : public std::vector<T, Allocator>
{
  public:
    using std::vector<T, Allocator>::size;
    using std::vector<T, Allocator>::at;

    /**
     * @method initialize
     * @brief  initialize the vector with the content of istream. istream is
     *         read per line. empty lines will add unitialized elements.
     * @param  in  inputstream to read from
     * @return void
     * @globalvars none
     * @exception  std::runtime_error
     * @conditions none
     */
    void initialize(std::istream& in)
    {
      if (!in.good())
        return;

      std::string line;
      unsigned i = 0;
      while (!in.eof() && in.good())
      {
        ++i;
        std::getline(in, line);

        /* skip last line if it's empty */
        if (line.empty() && in.eof())
          break;

        T value;
```

```
        try
        {
          if (!line.empty())
            value = boost::lexical_cast<T>(line);
        }
        catch(boost::bad_lexical_cast& ex)
        {
          std::stringstream sstr;
          sstr << "Unable to convert input (line " << i << "): " << ex.what();
          throw std::runtime_error(sstr.str());
        }

        push_back(value);
      }
    }

#if DEBUG
    /**
     * @method dump
     * @brief  dumps contents of vector to outputstream
     * @param  out  outputstream to write to
     * @return void
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    void dump(std::ostream& out)
    {
      out << "[MEMORY_DUMP]" << std::endl;
      for(unsigned i = 0; i < size(); ++i)
      {
        out << "[" << std::setw(4) << std::setfill('0') << i << "]  "
            << at(i) << std::endl;
      }
    }
#endif
};

/**
 * @class CMem
 *
 * Memory definition for CCPU
 */
typedef CVectorMem<CDat> CMem;

#endif

/* vim: set et sw=2 ts=2: */
```

4.4 cinstruction.h

```
/**
 * @module cinstruction
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Abstract class for displays
 * @date   13.05.2009
 */

#ifndef CINSTRUCTION_H
#define CINSTRUCTION_H 1

#include <iostream>
#include <list>

/* forward declare CCPU */
class CCPU;

/**
 * @class CInstruction
 *
 * Abstract class for displays
 */
class CInstruction
{
  public:
    /**
     * @method CInstruction
     * @brief  Default ctor
     * @param  name  name of instruction
     * @return -
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    CInstruction(std::string name)
      : m_name(name)
    {}

    /**
     * @method ~CInstruction
     * @brief  Default dtor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    virtual ~CInstruction()
    {}

    /**
     * @method operator==
     * @brief  implementation of operator ==
     * @param  name reference to std::string
     * @return true if instructionname is name
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    virtual bool operator==(std::string& name)
    {
      return name == m_name;
```

```
}

/**
 * @method operator()
 * @brief implementation of operator (CCPU)
 * @param cpu pointer to cpu
 * @return -
 * @globalvars none
 * @exception std::runtime_error
 * @conditions none
 */
virtual CInstruction& operator()(CCPU *cpu)
{
  execute(cpu);
  return *this;
}

/**
 * @method getName
 * @brief returns instruction name
 * @param -
 * @return name of instruction
 * @globalvars none
 * @exception none
 * @conditions none
 */
virtual const std::string& getName()
{
  return m_name;
}

/**
 * @method dump
 * @brief dumps information about instruction to outputstream
 * @param stream outputstream
 * @return reference to outputstream
 * @globalvars none
 * @exception none
 * @conditions none
 */
virtual std::ostream& dump(std::ostream& stream)
{
  stream << m_name;
  return stream;
}

/**
 * @method operator<<
 * @brief Shift/output operator for outputstream
 * @param stream reference to outputstream
 * @param instr object which will be printed to stream
 * @return reference to outputstream
 * @globalvars none
 * @exception none
 * @conditions none
 */
friend std::ostream& operator<<(std::ostream& stream, CInstruction& instr)
{
  return instr.dump(stream);
}

/**
 * @method parseRegister
 * @brief parses register syntax Rx (e.g. "R1")
```

```
     * @param  str  register in assembler syntax
     * @return registernumber
     * @globalvars none
     * @exception  std::runtime_error
     * @conditions none
     */
    virtual const unsigned parseRegister(const std::string& str);

    /**
     * @method checkRegister
     * @brief  performs a register boundary check
     *         does the register exist in cpu?
     * @param  cpu     pointer to cpu
     * @param  regidx  registernumber
     * @return -
     * @globalvars none
     * @exception  std::runtime_error
     * @conditions none
     */
    virtual void checkRegister(CCPU *cpu, const unsigned regidx);

    /**
     * @method factory
     * @brief  creates a new instance of this instruction
     * @param  -
     * @return new instruction instance
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    virtual CInstruction *factory() = 0;

    /**
     * @method compile
     * @brief  parses instruction parameters and prepares the
     *         instruction for executing
     * @param  params  list of parameters of this instruction
     * @return -
     * @globalvars none
     * @exception  std::runtime_error
     * @conditions none
     */
    virtual void compile(std::list<std::string>& params) = 0;

    /**
     * @method execute
     * @brief  executes the instruction
     * @param  cpu  pointer to cpu
     * @return -
     * @globalvars none
     * @exception  std::runtime_error
     * @conditions none
     */
    virtual void execute(CCPU *cpu) = 0;

  protected:
    /* members */
    /** name of instruction */
    std::string m_name;
};

#endif

/* vim: set et sw=2 ts=2: */
```

4.5 cinstruction.cpp

```
/**
 * @module cinstruction
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Abstract class for displays
 * @date   13.05.2009
 */

#include <sstream>
#include <stdexcept>
#include <boost/lexical_cast.hpp>
#include <assert.h>
#include "cinstruction.h"
#include "ccpu.h"

using namespace std;

const unsigned CInstruction::parseRegister(const std::string& str)
{
  unsigned reg;
  if (str.length() < 2 || str[0] != 'r')
    throw runtime_error("Invalid syntax of register");

  try
  {
    reg = boost::lexical_cast<unsigned>(str.substr(1));
  }
  catch(boost::bad_lexical_cast& ex)
  {
    throw runtime_error("Invalid syntax of register");
  }

  return reg;
}

/*-----------------------------------------------------------------------------*/

inline void CInstruction::checkRegister(CCPU *cpu, const unsigned regidx)
{
  assert(cpu != NULL);
  if (regidx >= cpu->getRegisterCount())
  {
    stringstream sstr;
    sstr << "Register R" << regidx << " doesn't exist (out of bound)";
    throw runtime_error(sstr.str());
  }
}

/* vim: set et sw=2 ts=2: */
```

4.6 instructions.h

```
/**
 * @module instructions
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Implementations of CInstruction
 * @date   10.05.2009
 */

#ifndef INSTRUCTIONS_H
#define INSTRUCTIONS_H 1

#include "cinstruction.h"
#include "ccpu.h"

/**
 * @class CInstructionInc
 *
 * Implementation of assembler command "inc"
 * Syntax: inc R1
 *   (R1++)
 */
class CInstructionInc
 : public CInstruction
{
  public:
    CInstructionInc()
      : CInstruction("inc")
    {}

    CInstructionInc *factory()
    {
      return new CInstructionInc;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
};

/*============================================================================*/

/**
 * @class CInstructionDec
 *
 * Implementation of assembler command "dec"
 * Syntax: dec R1
 *   (R1--)
 */
class CInstructionDec
 : public CInstruction
{
  public:
    CInstructionDec()
      : CInstruction("dec")
    {}

    CInstructionDec *factory()
    {
      return new CInstructionDec;
```

```
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
};

/*============================================================================*/

/**
 * @class CInstructionAdd
 *
 * Implementation of assembler command "add"
 * Syntax: add R1, R2, R3
 *   (R1 = R2 + R3)
 */
class CInstructionAdd
 : public CInstruction
{
  public:
    CInstructionAdd()
      : CInstruction("add")
    {}

    CInstructionAdd *factory()
    {
      return new CInstructionAdd;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
    /** register number */
    unsigned m_regidx2;
    /** register number */
    unsigned m_regidx3;
};

/*============================================================================*/

/**
 * @class CInstructionSub
 *
 * Implementation of assembler command "sub"
 * Syntax: sub R1, R2, R3
 *   (R1 = R2 - R3)
 */
class CInstructionSub
 : public CInstruction
{
  public:
    CInstructionSub()
      : CInstruction("sub")
    {}

    CInstructionSub *factory()
    {
      return new CInstructionSub;
```

```
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
    /** register number */
    unsigned m_regidx2;
    /** register number */
    unsigned m_regidx3;
};

/*============================================================================*/

/**
 * @class CInstructionMul
 *
 * Implementation of assembler command "mul"
 * Syntax: mul R1, R2, R3
 *   (R1 = R2 * R3)
 */
class CInstructionMul
 : public CInstruction
{
  public:
    CInstructionMul()
      : CInstruction("mul")
    {}

    CInstructionMul *factory()
    {
      return new CInstructionMul;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
    /** register number */
    unsigned m_regidx2;
    /** register number */
    unsigned m_regidx3;
};

/*============================================================================*/

/**
 * @class CInstructionDiv
 *
 * Implementation of assembler command "div"
 * Syntax: div R1, R2, R3
 *   (R1 = R2 / R3)
 */
class CInstructionDiv
 : public CInstruction
{
  public:
    CInstructionDiv()
      : CInstruction("div")
    {}
```

```
    CInstructionDiv *factory()
    {
      return new CInstructionDiv;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
    /** register number */
    unsigned m_regidx2;
    /** register number */
    unsigned m_regidx3;
};

/*=============================================================================*/

/**
 * @class CInstructionLoad
 *
 * Implementation of assembler command "load"
 * Syntax: load R1, R2
 *   (R1 = memory[R2])
 */
class CInstructionLoad
 : public CInstruction
{
  public:
    CInstructionLoad()
      : CInstruction("load")
    {}

    CInstructionLoad *factory()
    {
      return new CInstructionLoad;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
    /** register number */
    unsigned m_regidx2;
};

/*=============================================================================*/

/**
 * @class CInstructionStore
 *
 * Implementation of assembler command "store"
 * Syntax: store R1, R2
 *   (memory[R2] = R1)
 */
class CInstructionStore
 : public CInstruction
{
  public:
    CInstructionStore()
```

```
      : CInstruction("store")
    {}

    CInstructionStore *factory()
    {
      return new CInstructionStore;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
    /** register number */
    unsigned m_regidx2;
};

/*============================================================================*/

/**
 * @class CInstructionTest
 *
 * Implementation of assembler command "test"
 * Syntax: test R1
 *   (R1 == 0: zeroflag: true, R1 < 0: signflag: true)
 */
class CInstructionTest
 : public CInstruction
{
  public:
    CInstructionTest()
      : CInstruction("test")
    {}

    CInstructionTest *factory()
    {
      return new CInstructionTest;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
};

/*============================================================================*/

/**
 * @class CInstructionLabel
 *
 * Implementation of assembler command "label"
 * Syntax: label name:
 */
class CInstructionLabel
 : public CInstruction
{
  public:
    CInstructionLabel()
      : CInstruction("label")
    {}
```

```
    CInstructionLabel *factory()
    {
      return new CInstructionLabel;
    }

    void compile(std::list<std::string>& params)
    {}

    void execute(CCPU *cpu)
    {}
};

/*============================================================================*/

/**
 * @class CInstructionJumpA
 *
 * Implementation of assembler command "jumpa"
 * Syntax: jumpa labelname
 *   (jump to labelname)
 */
class CInstructionJumpA
 : public CInstruction
{
  public:
    CInstructionJumpA()
      : CInstruction("jumpa"), m_addr("")
    {}

    CInstructionJumpA *factory()
    {
      return new CInstructionJumpA;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU *cpu);

  protected:
    /** labelname */
    std::string m_addr;
};

/*============================================================================*/

/**
 * @class CInstructionJumpZ
 *
 * Implementation of assembler command "jumpz"
 * Syntax: jumpz labelname
 *   (jump to labelname if zeroflag)
 */
class CInstructionJumpZ
 : public CInstruction
{
  public:
    CInstructionJumpZ()
      : CInstruction("jumpz"), m_addr("")
    {}

    CInstructionJumpZ *factory()
    {
      return new CInstructionJumpZ;
    }
```

```
    void compile(std::list<std::string>& params);
    void execute(CCPU *cpu);

  protected:
    /** labelname */
    std::string m_addr;
};

/*============================================================================*/

/**
 * @class CInstructionJumpS
 *
 * Implementation of assembler command "jumps"
 * Syntax: jumps labelname
 *   (jump to labelname if signflag)
 */
class CInstructionJumpS
 : public CInstruction
{
  public:
    CInstructionJumpS()
      : CInstruction("jumps"), m_addr("")
    {}

    CInstructionJumpS *factory()
    {
      return new CInstructionJumpS;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU *cpu);

  protected:
    /** labelname */
    std::string m_addr;
};

/*============================================================================*/

/**
 * @class CInstructionWrite
 *
 * Implementation of assembler command "write"
 * Syntax: write DEV, R1
 *   (write R1 to DEV, which is a name of a display)
 */
class CInstructionWrite
 : public CInstruction
{
  public:
    CInstructionWrite()
      : CInstruction("write"), m_dev("")
    {}

    CInstructionWrite *factory()
    {
      return new CInstructionWrite;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU *cpu);

  protected:
```

```
    /** register number */
    unsigned m_regidx1;
    /** device name */
    std::string m_dev;
};

#endif

/* vim: set et sw=2 ts=2: */
```

4.7 instructions.cpp

```
/**
 * @module instructions
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Implementations of CInstruction
 * @date   10.05.2009
 */

#include <map>
#include <assert.h>
#include "instructions.h"

using namespace std;

void CInstructionInc::compile(std::list<std::string>& params)
{
  if (params.size() != 1)
    throw runtime_error("Invalid paramater count - must be 1");
  m_regidx1 = parseRegister(params.front());
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

void CInstructionInc::execute(CCPU *cpu)
{
  assert(cpu != NULL);
  assert(cpu->getRegisters() != NULL);
  checkRegister(cpu, m_regidx1);
  cpu->getRegisters()[ m_regidx1 ]++;
}

/*============================================================================*/

void CInstructionDec::compile(std::list<std::string>& params)
{
  if (params.size() != 1)
    throw runtime_error("Invalid paramater count - must be 1");
  m_regidx1 = parseRegister(params.front());
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

void CInstructionDec::execute(CCPU *cpu)
{
  assert(cpu != NULL);
  assert(cpu->getRegisters() != NULL);
  checkRegister(cpu, m_regidx1);
  cpu->getRegisters()[ m_regidx1 ]--;
}

/*============================================================================*/

void CInstructionAdd::compile(std::list<std::string>& params)
{
  if (params.size() != 3)
    throw runtime_error("Invalid paramater count - must be 3");
  m_regidx1 = parseRegister(params.front());
  params.pop_front();
  m_regidx2 = parseRegister(params.front());
  params.pop_front();
```

```
  m_regidx3 = parseRegister(params.front());
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

void CInstructionAdd::execute(CCPU *cpu)
{
  assert(cpu != NULL);
  assert(cpu->getRegisters() != NULL);
  checkRegister(cpu, m_regidx1);
  checkRegister(cpu, m_regidx2);
  checkRegister(cpu, m_regidx3);
  cpu->getRegisters()[ m_regidx1 ] = cpu->getRegisters()[ m_regidx2 ]
    + cpu->getRegisters()[ m_regidx3 ];
}

/*============================================================================*/

void CInstructionSub::compile(std::list<std::string>& params)
{
  if (params.size() != 3)
    throw runtime_error("Invalid paramater count - must be 3");
  m_regidx1 = parseRegister(params.front());
  params.pop_front();
  m_regidx2 = parseRegister(params.front());
  params.pop_front();
  m_regidx3 = parseRegister(params.front());
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

void CInstructionSub::execute(CCPU *cpu)
{
  assert(cpu != NULL);
  assert(cpu->getRegisters() != NULL);
  checkRegister(cpu, m_regidx1);
  checkRegister(cpu, m_regidx2);
  checkRegister(cpu, m_regidx3);
  cpu->getRegisters()[ m_regidx1 ] = cpu->getRegisters()[ m_regidx2 ]
    - cpu->getRegisters()[ m_regidx3 ];
}

/*============================================================================*/

void CInstructionMul::compile(std::list<std::string>& params)
{
  if (params.size() != 3)
    throw runtime_error("Invalid paramater count - must be 3");
  m_regidx1 = parseRegister(params.front());
  params.pop_front();
  m_regidx2 = parseRegister(params.front());
  params.pop_front();
  m_regidx3 = parseRegister(params.front());
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

void CInstructionMul::execute(CCPU *cpu)
{
  checkRegister(cpu, m_regidx1);
  checkRegister(cpu, m_regidx2);
```

```
  checkRegister(cpu, m_regidx3);
  cpu->getRegisters()[ m_regidx1 ] = cpu->getRegisters()[ m_regidx2 ]
    * cpu->getRegisters()[ m_regidx3 ];
}

/*============================================================================*/

void CInstructionDiv::compile(std::list<std::string>& params)
{
  if (params.size() != 3)
    throw runtime_error("Invalid paramater count - must be 3");
  m_regidx1 = parseRegister(params.front());
  params.pop_front();
  m_regidx2 = parseRegister(params.front());
  params.pop_front();
  m_regidx3 = parseRegister(params.front());
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

void CInstructionDiv::execute(CCPU *cpu)
{
  assert(cpu != NULL);
  assert(cpu->getRegisters() != NULL);
  checkRegister(cpu, m_regidx1);
  checkRegister(cpu, m_regidx2);
  checkRegister(cpu, m_regidx3);
  cpu->getRegisters()[ m_regidx1 ] = cpu->getRegisters()[ m_regidx2 ]
    / cpu->getRegisters()[ m_regidx3 ];
}

/*============================================================================*/

void CInstructionLoad::compile(std::list<std::string>& params)
{
  if (params.size() != 2)
    throw runtime_error("Invalid paramater count - must be 2");
  m_regidx1 = parseRegister(params.front());
  params.pop_front();
  m_regidx2 = parseRegister(params.front());
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

void CInstructionLoad::execute(CCPU *cpu)
{
  assert(cpu != NULL);
  assert(cpu->getRegisters() != NULL);
  assert(cpu->getMemory() != NULL);
  checkRegister(cpu, m_regidx1);
  checkRegister(cpu, m_regidx2);
  CDat val(cpu->getRegisters()[ m_regidx2 ]);
  cpu->getRegisters()[ m_regidx1 ] = (*cpu->getMemory())[ val ];
}

/*============================================================================*/

void CInstructionStore::compile(std::list<std::string>& params)
{
  if (params.size() != 2)
    throw runtime_error("Invalid paramater count - must be 2");
  m_regidx1 = parseRegister(params.front());
```

```
  params.pop_front();
  m_regidx2 = parseRegister(params.front());
  params.pop_front();
}

/*-------------------------------------------------------------------------*/

void CInstructionStore::execute(CCPU *cpu)
{
  assert(cpu != NULL);
  assert(cpu->getRegisters() != NULL);
  assert(cpu->getMemory() != NULL);
  checkRegister(cpu, m_regidx1);
  checkRegister(cpu, m_regidx2);
  CDat val(cpu->getRegisters()[ m_regidx2 ]);
  (*cpu->getMemory())[ val ] = cpu->getRegisters()[ m_regidx1 ];
}

/*=========================================================================*/

void CInstructionTest::compile(std::list<std::string>& params)
{
  if (params.size() != 1)
    throw runtime_error("Invalid paramater count - must be 1");
  m_regidx1 = parseRegister(params.front());
  params.pop_front();
}

/*-------------------------------------------------------------------------*/

void CInstructionTest::execute(CCPU *cpu)
{
  assert(cpu != NULL);
  assert(cpu->getRegisters() != NULL);
  checkRegister(cpu, m_regidx1);
  if (cpu->getRegisters()[ m_regidx1 ] == CDat(0))
    cpu->setFlagZero(true);
  if (cpu->getRegisters()[ m_regidx1 ] < CDat(0))
    cpu->setFlagSign(true);
}

/*=========================================================================*/

void CInstructionJumpA::compile(std::list<std::string>& params)
{
  if (params.size() != 1)
    throw runtime_error("Invalid paramater count - must be 1");
  m_addr = params.front();
  params.pop_front();
}

/*-------------------------------------------------------------------------*/

void CInstructionJumpA::execute(CCPU *cpu)
{
  assert(cpu != NULL);
  assert(cpu->getRegisters() != NULL);
  assert(cpu->getProgram() != NULL);
  if (m_addr.empty())
    throw runtime_error("Empty address");
  cpu->getRegisters()[ 0 ] = cpu->getProgram()->findLabel(m_addr);
}

/*=========================================================================*/
```

```
void CInstructionJumpZ::compile(std::list<std::string>& params)
{
  if (params.size() != 1)
    throw runtime_error("Invalid paramater count - must be 1");
  m_addr = params.front();
  params.pop_front();
}

/*-------------------------------------------------------------------------*/

void CInstructionJumpZ::execute(CCPU *cpu)
{
  assert(cpu != NULL);
  assert(cpu->getRegisters() != NULL);
  assert(cpu->getProgram() != NULL);
  if (!cpu->getFlagZero())
    return;
  if (m_addr.empty())
    throw runtime_error("Empty address");
  cpu->getRegisters()[ 0 ] = cpu->getProgram()->findLabel(m_addr);
}

/*=========================================================================*/

void CInstructionJumpS::compile(std::list<std::string>& params)
{
  if (params.size() != 1)
    throw runtime_error("Invalid paramater count - must be 1");
  m_addr = params.front();
  params.pop_front();
}

/*-------------------------------------------------------------------------*/

void CInstructionJumpS::execute(CCPU *cpu)
{
  assert(cpu != NULL);
  assert(cpu->getRegisters() != NULL);
  assert(cpu->getProgram() != NULL);
  if (!cpu->getFlagSign())
    return;
  if (m_addr.empty())
    throw runtime_error("Empty address");
  cpu->getRegisters()[ 0 ] = cpu->getProgram()->findLabel(m_addr);
}

/*=========================================================================*/

void CInstructionWrite::compile(std::list<std::string>& params)
{
  if (params.size() != 2)
    throw runtime_error("Invalid paramater count - must be 2");
  m_dev = params.front();
  params.pop_front();
  m_regidx1 = parseRegister(params.front());
  params.pop_front();
}

/*-------------------------------------------------------------------------*/

void CInstructionWrite::execute(CCPU *cpu)
{
  assert(cpu != NULL);
```

```
  assert(cpu->getRegisters() != NULL);
  checkRegister(cpu, m_regidx1);
  if (m_dev.empty())
    throw runtime_error("Empty device");

  CDisplay *display = NULL;
  std::set<CDisplay *> displays = cpu->getDisplays();
  std::set<CDisplay *>::iterator it;
  for(it = displays.begin(); it != displays.end(); ++it)
  {
    if ((*it)->getName() == m_dev)
    {
      display = *it;
      break;
    }
  }
  if (display == NULL)
    throw runtime_error("Unknown display");

  display->display(cpu->getRegisters()[ m_regidx1 ]);
}


/* vim: set et sw=2 ts=2: */
```

4.8 cdisplay.h

```
/**
 * @module cdisplay
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Abstract template class for displays
 * @date   10.05.2009
 */

#ifndef CDISPLAY_H
#define CDISPLAY_H 1

/**
 * @class CDisplayT
 *
 * Abstract template class for displays
 */
template <class T>
class CDisplayT
{
  public:
    /**
     * @method CDisplayT
     * @brief  Default ctor
     * @param  name  name of display
     * @return -
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    CDisplayT(std::string name)
      : m_name(name)
    {}

    /**
     * @method ~CDisplayT
     * @brief  Default dtor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    virtual ~CDisplayT()
    {}

    /**
     * @method getName
     * @brief  returns name of display
     * @param  -
     * @return name of display
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    virtual const std::string& getName()
    {
      return m_name;
    }

    /**
     * @method display
     * @brief  prints value to display
```

```
     * @param  value  value to display
     * @return -
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    virtual void display(const T &value) = 0;

  protected:
    /* members */
    /** name of display */
    std::string m_name;
};

/**
 * @class CDisplay
 *
 * Memory definition for CCPU
 */
typedef CDisplayT<CDat> CDisplay;

#endif

/* vim: set et sw=2 ts=2: */
```

4.9 displays.h

```
/**
 * @module displays
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Implementations of CDisplay
 * @date   10.05.2009
 */

#ifndef DISPLAYS_H
#define DISPLAYS_H 1

#include <iomanip>
#include "cdisplay.h"

/**
 * @class CDisplayWDEZ
 *
 * Implementation of CDisplay
 * Prints CDat to stdout as decimal
 */
class CDisplayWDEZ
  : public CDisplay
{
  public:
    CDisplayWDEZ()
      : CDisplay("wdez")
    {}

    /**
     * @method display
     * @brief  prints value to display
     * @param  value  value to display
     * @return -
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    void display(const CDat &value)
    {
      std::cout << std::dec << value << std::endl;
    }
};

/*============================================================================*/

/**
 * @class CDisplayWHEX
 *
 * Implementation of CDisplay
 * Prints CDat to stdout as decimal
 */
class CDisplayWHEX
  : public CDisplay
{
  public:
    CDisplayWHEX()
      : CDisplay("whex")
    {}

    /**
     * @method display
     * @brief  prints value to display
```

```
     * @param   value   value to display
     * @return -
     * @globalvars none
     * @exception   none
     * @conditions none
     */
    void display(const CDat &value)
    {
      std::cout << std::hex << value << std::endl;
    }
};

#endif

/* vim: set et sw=2 ts=2: */
```

4.10 cprogram.h

```
/**
 * @module cprogram
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  CProgram extends std::vector and adds a method for parsing programfile
 * @date   10.05.2009
 */

#ifndef CPROGRAM_H
#define CPROGRAM_H 1

#include <vector>
#include <set>
#include <map>
#include "cinstruction.h"

/**
 * @class CProgram
 *
 * CProgram extends std::vector and adds a method for parsing
 * programfile. This adds instances of CInstruction to CProgram itself.
 */
class CProgram
  : public std::vector<CInstruction *>
{
  public:
    /**
     * @method CProgram
     * @brief  Default ctor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    CProgram();

    /**
     * @method ~CProgram
     * @brief  Default dtor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    ~CProgram();

    /**
     * @method getLabels
     * @brief  get reference to labels map
     * @param  -
     * @return reference to labels map
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    const std::map<std::string, unsigned>& getLabels() const
    {
      return m_labels;
    }
```

```
    /**
     * @method findLabel
     * @brief  search for label
     * @param  label  name of label to search for
     * @return index of found label in program
     * @globalvars none
     * @exception  std::runtime_error
     * @conditions none
     */
    unsigned findLabel(const std::string& label) const;

    /**
     * @method compile
     * @brief  create instructions from parsing stream
     * @param  in  inputstream to read from
     * @return void
     * @globalvars none
     * @exception  std::runtime_error
     * @conditions none
     */
    void compile(std::istream& in);

#if DEBUG
    /**
     * @method dump
     * @brief  dumps contents to outputstream
     * @param  out  outputstream to write to
     * @return void
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    void dump(std::ostream& out);
#endif

  private:
    /* members */
    /** set of known instructions */
    std::set<CInstruction *> m_instrset;
    std::map<std::string, unsigned> m_labels;
};

#endif

/* vim: set et sw=2 ts=2: */
```

4.11 cprogram.cpp

```
/**
 * @module cprogram
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  CProgram extends std::vector and adds a method for parsing programfile
 * @date   12.05.2009
 */

#include <boost/algorithm/string.hpp>
#include <boost/algorithm/string/split.hpp>
#ifdef DEBUG
# include <iostream>
# include <iomanip>
#endif
#include "cprogram.h"
#include "instructions.h"

using namespace std;

CProgram::CProgram()
{
  m_instrset.insert(new CInstructionInc);
  m_instrset.insert(new CInstructionDec);
  m_instrset.insert(new CInstructionAdd);
  m_instrset.insert(new CInstructionSub);
  m_instrset.insert(new CInstructionMul);
  m_instrset.insert(new CInstructionDiv);
  m_instrset.insert(new CInstructionLoad);
  m_instrset.insert(new CInstructionStore);
  m_instrset.insert(new CInstructionTest);
  m_instrset.insert(new CInstructionLabel);
  m_instrset.insert(new CInstructionJumpA);
  m_instrset.insert(new CInstructionJumpZ);
  m_instrset.insert(new CInstructionJumpS);
  m_instrset.insert(new CInstructionWrite);
}

/*----------------------------------------------------------------------------*/

CProgram::~CProgram()
{
  /* free instruction set */
  set<CInstruction *>::iterator it;
  for (it = m_instrset.begin(); it != m_instrset.end(); ++it)
    delete *it;

  /* free instruction */
  for (iterator it = begin(); it != end(); ++it)
    delete *it;
}

/*----------------------------------------------------------------------------*/

void CProgram::compile(std::istream& in)
{
  if (!in.good())
    return;

  string line;
  unsigned i = 0;
  while (!in.eof() && in.good())
  {
```

```
++i;

/* read stream per line */
getline(in, line);
if (line.empty())
  continue;

boost::trim(line);
boost::to_lower(line);

/* ignore comments */
if (line.find_first_of('#') == 0)
  continue;

/* get instruction name */
size_t pos = line.find_first_of('␣');
string instrname(line.substr(0, pos));

/* search and create instruction */
CInstruction *instrptr = NULL;
set<CInstruction *>::iterator it;
for (it = m_instrset.begin(); it != m_instrset.end(); ++it)
{
  if (*(*it) == instrname)
  {
    instrptr = *it;
    break;
  }
}
if (instrptr == NULL)
{
  stringstream sstr;
  sstr << "Unknown␣instruction␣'" << instrname << "'␣on␣line␣" << i << ".";
  throw runtime_error(sstr.str());
}

/* create instruction */
CInstruction *instr = instrptr->factory();

/* parse instruction parameters */
string params = (pos == string::npos) ? "" : line.substr(pos + 1);
boost::trim(params);
list<string> instrparams;
boost::split(instrparams, params, boost::is_any_of(",␣\t"), boost::
    token_compress_on);

/* let instruction parse the parameters. catch+throw exception */
try
{
  /* handle label instruction ourself, but still add a dummy instruction */
  if (instrname == "label")
  {
    if (instrparams.size() != 1)
      throw runtime_error("Invalid␣paramater␣count␣-␣must␣be␣1");
    string label(instrparams.front());
    if (label.length() < 2 || label[ label.length() - 1] != ':')
      throw runtime_error("Label␣has␣invalid␣syntax");
    m_labels[ label.substr(0, label.length() - 1) ] = size();
  }
  instr->compile(instrparams);
}
catch(runtime_error& ex)
{
  stringstream sstr;
```

```
      sstr << "Unable to compile instruction '" << instrname
           << "' (line " << i << "): " << ex.what();
      throw runtime_error(sstr.str());
    }

    push_back(instr);
  }
}

/*-----------------------------------------------------------------------------*/

unsigned CProgram::findLabel(const std::string& label) const
{
  map<string, unsigned>::const_iterator it;
  it = m_labels.find(label);
  if (it == m_labels.end())
    throw runtime_error("Unknown label '" + label + "'");
  return it->second;
}

/*-----------------------------------------------------------------------------*/

#if DEBUG
void CProgram::dump(std::ostream& out)
{
  out << "[PROGRAM DUMP]" << endl;
  unsigned i = 0;
  for(iterator it = begin(); it < end(); ++it)
  {
    out << "[" << std::setw(4) << std::setfill('0') << i << "]  "
        << *(*it) << endl;
    ++i;
  }
}
#endif

/* vim: set et sw=2 ts=2: */
```

4.12 ccpu.h

```
/**
 * @module ccpu
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  CPU implementation. Used as a container for memory and instructions.
 *         Implements a run method to execute the program (= the instructions).
 * @date   10.05.2009
 */

#ifndef CCPU_H
#define CCPU_H 1

#include <iostream>
#include <set>
#include "cdat.h"
#include "cmem.h"
#include "cprogram.h"
#include "cdisplay.h"

/**
 * @class CCPU
 *
 * CPU implementation. Used as a container for memory and instructions.
 * Implements a run method to execute the program (= the instructions).
 */
class CCPU
{
  public:
    /**
     * @method CCPU
     * @brief  Default ctor
     * @param  cnt  number of registers to allocate for this cpu
     * @return -
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    CCPU(const unsigned cnt);

    /**
     * @method ~CCPU
     * @brief  Default dtor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    ~CCPU();

    /**
     * @method getRegisterCount
     * @brief  get number of registers
     * @param  -
     * @return number of registers
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    const unsigned getRegisterCount() const
    {
      return m_regcnt;
```

```
}

/**
 * @method getRegisters
 * @brief  get pointer to registers array
 * @param  -
 * @return pointer to registers array
 * @globalvars none
 * @exception  none
 * @conditions none
 */
CDat *getRegisters() const
{
  return m_registers;
}

/**
 * @method setMemory
 * @brief  set memory of cpu
 * @param  memory  pointer to memory
 * @return -
 * @globalvars none
 * @exception  none
 * @conditions none
 */
void setMemory(CMem *memory)
{
  m_memory = memory;
}

/**
 * @method getMemory
 * @brief  get pointer to memory
 * @param  -
 * @return pointer to memory
 * @globalvars none
 * @exception  none
 * @conditions none
 */
CMem *getMemory() const
{
  return m_memory;
}

/**
 * @method setProgram
 * @brief  set program to execute
 * @param  program  pointer to program
 * @return -
 * @globalvars none
 * @exception  none
 * @conditions none
 */
void setProgram(const CProgram *program)
{
  m_program = program;
}

/**
 * @method getProgram
 * @brief  get pointer to program
 * @param  -
 * @return pointer to program
 * @globalvars none
```

```
 * @exception  none
 * @conditions none
 */
const CProgram *getProgram()
{
  return m_program;
}

/**
 * @method getDisplays
 * @brief  get set of pointers to displays
 * @param  -
 * @return reference to set of pointers to displays
 * @globalvars none
 * @exception  none
 * @conditions none
 */
const std::set<CDisplay *>& getDisplays()
{
  return m_displays;
}

/**
 * @method setFlagZero
 * @brief  set zero flag
 * @param  value  new value of zero flag
 * @return -
 * @globalvars none
 * @exception  none
 * @conditions none
 */
void setFlagZero(const bool value)
{
  m_flagzero = value;
}

/**
 * @method getFlagZero
 * @brief  get value of zero flag
 * @param  -
 * @return value of zero flag
 * @globalvars none
 * @exception  none
 * @conditions none
 */
const bool getFlagZero()
{
  return m_flagzero;
}

/**
 * @method setFlagSign
 * @brief  set sign flag
 * @param  value  new value of sign flag
 * @return -
 * @globalvars none
 * @exception  none
 * @conditions none
 */
void setFlagSign(const bool value)
{
  m_flagsign = value;
}
```

```
    /**
     * @method getFlagSign
     * @brief  get value of sign flag
     * @param  -
     * @return value of sign flag
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    const bool getFlagSign()
    {
      return m_flagsign;
    }

    /**
     * @method run
     * @brief  execute current program
     * @param  -
     * @return -
     * @globalvars none
     * @exception  std::runtime_error
     * @conditions none
     */
    void run();

#if DEBUG
    /**
     * @method dumpRegisters
     * @brief  dump content of registers to outputstream
     * @param  out  outputstream to write to
     * @return void
     * @globalvars none
     * @exception  none
     * @conditions none
     */
    void dumpRegisters(std::ostream& out);
#endif

  private:
    /* members */
    CDat *m_registers;
    unsigned m_regcnt;
    CMem *m_memory;
    const CProgram *m_program;
    std::set<CDisplay *> m_displays;
    bool m_flagzero;
    bool m_flagsign;
};

#endif

/* vim: set et sw=2 ts=2: */
```

4.13 ccpu.cpp

```
/**
 * @module ccpu
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  CPU implementation. Used as a container for memory and instructions.
 *         Implements an run method to execute the program (= the instructions).
 * @date   10.05.2009
 */

#ifdef DEBUG
# include <iostream>
# include <iomanip>
#endif
#include "ccpu.h"
#include "displays.h"

using namespace std;

CCPU::CCPU(const unsigned cnt)
  : m_regcnt(cnt), m_memory(NULL), m_program(NULL), m_flagzero(false), m_flagsign(
      false)
{
  /* create registers */
  m_registers = new CDat[cnt];
  for(unsigned i = 0; i < cnt; ++i)
    m_registers[i] = 0;

  /* create displays */
  m_displays.insert(new CDisplayWDEZ);
  m_displays.insert(new CDisplayWHEX);
}

/*----------------------------------------------------------------------------*/

CCPU::~CCPU()
{
  /* delete registers */
  delete[] m_registers;
  m_registers = NULL;

  /* delete displays */
  std::set<CDisplay *>::iterator it;
  for (it = m_displays.begin() ; it != m_displays.end(); ++it)
    delete *it;
}

/*----------------------------------------------------------------------------*/

void CCPU::run()
{
  if (m_memory == NULL)
    throw runtime_error("CPU␣has␣no␣memory");
  if (m_program == NULL)
    throw runtime_error("CPU␣has␣no␣program␣to␣execute");
  if (m_regcnt == 0)
    throw runtime_error("CPU␣has␣no␣registers");

  bool run = true;
  while(run)
  {
    unsigned pc = static_cast<unsigned>(m_registers[0]);
```

```
    /* end of the program reached */
    if (pc == m_program->size())
      break;

    /* pc is out of bound */
    if (pc > m_program->size())
      throw runtime_error("Programcounter is out of bound");

    /* execute instruction */
    (*m_program->at(pc))(this);
    ++m_registers[0];
  }
}

/*-----------------------------------------------------------------------------*/

#if DEBUG
void CCPU::dumpRegisters(std::ostream& out)
{
  out << "[REGISTER DUMP]" << endl;
  for(unsigned i = 0; i < getRegisterCount(); ++i)
  {
    out << "[" << std::setw(4) << std::setfill('0') << i << "]  "
        << m_registers[i] << endl;
  }
}
#endif

/* vim: set et sw=2 ts=2: */
```