Технология разработки моделей микроконтроллеров

с использованием декларативно-алгоритмических языков описания ядра и периферии

**Михаил Долинский,
Игорь Ермолаев,
Алексей Федорцов,
Вячеслав Литвинов,
Алексей Толкачев,
Игорь Гончаренко,
Игорь Коршунов**

Введение

Основными достоинствами комплекса средств [1-7] совместной разработки программного и аппаратного обеспечения встроенных мультипроцессорных систем являются возможность настройки на конкретное семейство микрокронтроллеров, а также эффективные средства создания моделей.

Как уже отмечалось, модели микроконтроллеров могут достаточно легко и быстро создаваться с использованием языков программирования высокого уровня [5]. Однако следствием такого подхода может оказаться недостаточная для конкретного приложения производительность созданных моделей. Поэтому в СНИЛ «Новые информационные технологии» [7] разработана альтернативная технология разработки модели микроконтроллера, основывающаяся на генерации исходных ассемблерных текстов моделей микроконтроллеров, за счет чего можно достигнуть увеличения производительности симуляции в 10 и более раз. Необходимо отметить, что несмотря на наличие специально разработанных декларативно-алгоритмических языков описания ядра (PCDL — Processor Core Decription Language) и периферии (PPDL — Processor Peripherals Decription Language), для создания такой модели разработчик должен иметь определенные навыки разработки и отладки ассемблерных программ.

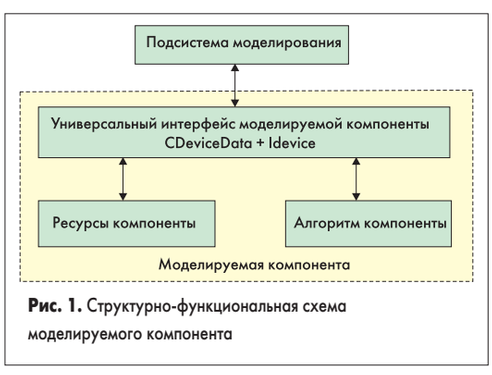

Рис. 1. Структурно-функциональная схема моделируемого компонента

1. Обобщенная структурно-функциональная схема моделируемого компонента

Моделируемый компонент состоит из двух основных блоков:

- ресурсы компонента;
- алгоритм компонента.

С этими блоками взаимодействует подсистема моделирования через универсальный интерфейс моделируемого компонента. Схематически взаимодействие представлено на рис. 1. Для ускорения моделирования взаимодействие между блоками происходит напрямую, без участия универсального интерфейса.

Для доступа к ресурсам используются свойства объекта CDeviceData. Структурная схема ресурсов компонента приведена на рис. 2. Полужирным шрифтом отмечены названия свойств объекта CDeviceData. Двунаправленными соединительными линиями указаны наложения одних ресурсов на другие. Пунктирной линией обведены ресурсы, которые присущи только моделям процессоров (это кодовая память и указатель инструкций).

Как видно из рисунка, почти все ресурсы компонента накладываются на ту или иную память. Это сделано для упрощения прямого доступа к ресурсам из блока с алгоритмом компонента — для доступа необходимо знать, в какой памяти лежит ресурс и по какому смещению. Если необходимая память не присутствует в реальном устройстве, то она помечается как скрытая и не отображается пользователю.

Контакты не наложены на память в связи с тем, что они имеют более сложную структуру, чем совокупность ячеек памяти. Каждый разряд контакта кодируется четырехбитным числом. Это связано с тем, что:

- контакт может быть четырех типов: обычный, открытый коллектор, открытый эммитер и контакт с Z-состоянием;
- разряд контакта может принимать четыре значения: ноль, единица, Z-состояние и неопределенность;

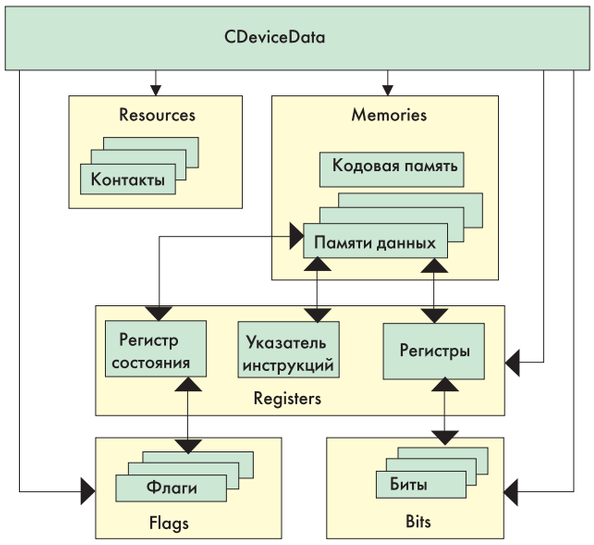

Рис. 2. Структурная схема ресурсов компонента

- необходимо учитывать соединение нескольких разнотипных контактов.

Алгоритм компонента состоит из процедуры инициализации при включении питания и набора процедур, зависящих от типа устройства. Если это периферийное устройство, то набор процедур выполняет реакцию на изменение состояния входных контактов и устанавливает значения на выходных. Если это процессор, то набор процедур выполняет симуляцию исполнения машинного кода и поведения внутренней периферии.

2. Настраиваемая модель процессора

2.1. Язык описания ядра процессора

Разработанный язык описания ядра процессора называется PCDL (Processor Core Description Language). Этот язык базируется на декларативном описании ресурсов процессора и на описании алгоритмов исполнения инструкций с помощью ассемблера для Intel 486.

Ресурсы процессора подразделяются на четыре класса: памяти, регистры, флаги и биты. Для описания этих ресурсов используются следующие синтаксические конструкции:

```
Memory(
<идентификатор памяти> : «<имя памяти>»,
<размер слова в битах>, <минимальное смещение в словах>,
<максимальное смещение в словах>, <размер памяти в словах>,
 «<имя регистра указателя (если есть)>»,
 [<список атрибутов через запятую>], <алгоритм после записи>) in
<идентификатор содержащей памяти>(<начальное смещение в словах>);
```

Имя памяти используется в среде симуляции, идентификатор памяти необходим для описания ресурсов, наложенных на эту память, и для использования в макросах чтения и записи.

Размер слова в битах определяется минимальным размером адресуемой ячейки памяти. Этот параметр используется средой симуляции для правильной работы с памятью.

Максимальное смещение в словах может быть нулевым, тогда оно рассчитывается путем добавления к минимальному смещению размера памяти и вычитанием единицы. Размер памяти в словах может быть нулевым, тогда он рассчитывается путем вычитания из максимального смещения минимального и добавлением единицы. Минимальное смещение, максимальное смещение и размер памяти в словах может быть задан в шестнадцатеричном виде, для этого перед значением необходимо добавить префикс «0x».

Имя регистра указателя — это имя того регистра, которым будет индексироваться память, например, для кодовой памяти это счетчик инструкций, для стековой — указатель на вершину стека. Если такого регистра нет, то имя пропускается (то есть между двумя запятыми ничего нет).

Атрибуты памяти могут быть следующие:

- mfCode — память, в которой расположен код программы (регистр-указатель в описании — это счетчик команд);
- mfStack — память, в которой расположен стек (регистр-указатель в описании — это указатель на вершину стека);
- mfData — память, в которой расположены данные;
- mfHide — память, недоступная пользователю среды, где используется модель — такой атрибут обычно устанавливается для логической памяти.

Идентификатор содержащей памяти — это идентификатор памяти, в которую вкладывается, начиная с указанного смещения, описываемая память. Данная информация может быть опущена.

Алгоритм после записи — это имя пользовательского алгоритма, который будет выполнен после изменения памяти в среде симуляции. Имя алгоритма может быть опущено.

Например, внутренняя, кодовая и внешняя памяти для i8051 могут быть описаны следующим образом:

```
Memory(Internal : «Internal Memory», 8, 0, 255, 0, «SP», [mfData, mfStack], InternalWrite);
Memory(Code : «Code Memory», 8, 0, 0xFFFF, 0, «PC», [mfCode]);
Memory(External : «External Memory», 8, 0, 0, 0x10000, , [mfData]);
```

Для описания регистров используется следующая синтаксическая конструкция:

```
Regs(
    <идентификатор памяти, в которой расположены регистры>,
    <адрес, с которого начинаются регистры в словах памяти>,
    <размер регистра в словах памяти>,
    (<список регистров через запятую>),
    [<список атрибутов через запятую>], <алгоритм после записи>);
```

Алгоритм после записи — это имя пользовательского алгоритма, который будет выполнен после изменения регистра в среде симуляции. Например, описание регистров A и PSW для i8051 выглядит так:

```
Regs(Internal, 0xE0, 1, (A));
Regs(Internal, 0xD0, 1, (PSW), [raDefault], WritePSW);
```

Для описания флагов используется следующая синтаксическая конструкция:

```
Flags(
<идентификатор ресурса, содержащего флаги>,
<начальное смещение в битах>, <конечное смещение в битах>,
(<список флагов через запятую>), [<список атрибутов через запятую>],
<алгоритм после записи>);
```

Казалось бы, достаточно одного начального смещения, но для контроля количества описанных битов и для задания направления описания (можно описывать с 0 по 7 бит, а можно с 7 по 0 бит) необходимо конечное смещение.

Под идентификатором ресурса понимают имя регистра состояния процессора (если есть таковой) или идентификатор памяти.

Алгоритм после записи — это имя пользовательского алгоритма, который будет выполнен после изменения флага в среде симуляции.

Для i8051 описание флагов будет выглядеть следующим образом:

```
Flags(PSW, 7, 0, (CY, AC, F0, RS1, RS0, OV, , P), [raDefault], WritePSW);
```

Биты отличаются от флагов тем, что флаги наложены на регистр состояния процессора и влияют на ход выполнения программы, а биты нет. Для описания битов используется следующая синтаксическая конструкция:

```
Bits(
<идентификатор ресурса, содержащего биты>,
<начальное смещение в битах>, <конечное смещение в битах>,
(<список битов через запятую>), [<список атрибутов через запятую>],
<алгоритм после записи>);
```

Алгоритм после записи — это имя пользовательского алгоритма, который будет выполнен после изменения бита в среде симуляции. Для i8051 описание битов в регистре IE будет выглядеть следующим образом:

```
Bits(IE, 7, 0, (EA, , ET2, ES, ET1, EX1, ET0, EX0));
```

После описания ресурсов можно приступать к описанию инструкций. Описание инструкций осуществляется в два этапа: описание групп декодирования операндов и описание алгоритмов инструкций.

Для описания группы декодирования операндов используется следующая синтаксическая конструкция:

```
group <имя группы>(
      <битовое представление операндов>,
      {
        <алгоритм для симулятора>
      }
      {
        <алгоритм для дизассемблера>
      }
);
```

Битовое представление операндов берется из документации на процессор — биты, которые отвечают за алгоритм инструкции, помечаются символом «x», а биты, которые отвечают за операнды, помечаются подходящим по смыслу символом.

Алгоритм для симулятора и дизассемблера представляют собой текст на языке ассемблера Intel 80x86.

Например, для инструкции «ADD A, D8» (добавляет к регистру A 8-битное значение из внутренней памяти со смещением D8) в i8051 описание группы декодирования операндов будет выглядеть так:

```
group G_A_D8 (
      xxxx xxxx dddd dddd,
      {
            mov             DST_OFS, regA
            RES_Read8       memInternal, regA, DST32
            movzx           SRC32, CODE_H8
            RES_Read8       memInternal, SRC32, SRC32
      }
      {
            STR_Cpy         StrOpers, name_regA     ; «A»
            STR_Cat         StrOpers, StrComma      ; «A, «
            STR_Hex8        StrOper, CODE_H8        ; «03»
            STR_Cat         StrOpers, StrOper       ; «A, 03»
      }
);
```

Вместо «xxxx xxxx dddd dddd» можно было написать «x4 x4 d4 d4» или «x8 d8».

Описание алгоритмов инструкций осуществляется с помощью следующей синтаксической конструкции:

```
instr <имя инструкции>(
      <имя группы декодирования операндов>,
      <битовое представление кода инструкции>,
      <базовое количество циклов>) {
      <алгоритм исполнения инструкции>
      }, <тип инструкции>;
```

Если инструкция без операндов, то имя группы декодирования операндов может быть опущено.

Битовое представление операндов берется из документации на процессор — биты, которые отвечают за алгоритм инструкции, записываются как соответствующий набор нулей и единиц, а биты, которые отвечают за операнды, помечаются подходящим по смыслу символом.

Если при выполнении инструкции используются дополнительные циклы, количество которых зависит от значений операндов, то в алгоритм декодирования операндов следует добавить вызов макроса ALG_AddCycles, в который необходимо передать количество дополнительных циклов. Например, так:

```
ALG_AddCycles           2
```

Тип инструкции может быть следующим:

- itSimple — простая инструкция;
- itJump — переход по адресу;
- itCall — вызов подпрограммы;
- itRet — возврат из подпрограммы;
- itICall — вызов прерывания;
- itIRet — возврат из прерывания.

Тип инструкции может быть опущен, в качестве значения по умолчанию принимается itSimple.

Например, инструкция «ADD A, D8» для i8051 может быть описана следующим образом:

```
instr  ADD(G_A_D8, 0010 0101 dddd dddd, 1) {
       ALG_ReadFlags
       add                DST8, SRC8
       ALG_WriteFlags
       RES_Write8         memInternal, DST_OFS, DST8
};
```

Вместо «0010 0101 dddd dddd» можно было написать «0x25 d8».

Кроме алгоритмов исполнения инструкций существуют еще дополнительные алгоритмы, которые используются при описании. Разделяют два вида дополнительных алгоритмов:

- стандартные (STR_Cat, STR_Cpy, STR_Hex8, STR_Hex16, STR_Bit, ALG_ReadFlags, ALG_WriteFlags, RES_Read8, RES_Write8 и т. д.);
- определяемые пользователем (например: UALG_BankRegRead8, UALG_InternalWrite8).

Для описания алгоритмов, определенных пользователем, используют следующие синтаксические конструкции:

```
execute <имя алгоритма>(<список параметров через запятую>) {
      <алгоритм, использующий параметры>
};

disassemble <имя алгоритма>(<список параметров через запятую>) {
      <алгоритм, использующий параметры>
};

common <имя алгоритма>(<список параметров через запятую>) {
      <алгоритм, использующий параметры>
};
```

Алгоритм представляет собой текст на языке ассемблер для i486. Для вызова пользовательского алгоритма следует написать:

```
UALG_<имя алгоритма> <список параметров, если есть>
```

Ключевые слова execute, disassembler, common означают, что алгоритм доступен при описании инструкций, алгоритм доступен при описании дизассемблера, алгоритм доступен при описании инструкций и дизассемблера.

При написании алгоритмов на языке ассемблер для i486 можно использовать переменные и массивы. Объявление переменных и массивов осуществляется с помощью следующих синтаксических конструкций:

```
[static] <тип переменной> <имя переменной> = <значение переменной>;

[static] <тип элемента> <имя массива>[] = (
      <значение 1-го элемента>,
      …
      <значение n-го элемента>
);
```

Ключевое слово static указывает на то, что переменная одна на все процессоры этого типа. Иначе у каждого процессора будет своя переменная с указанным именем, которую они могут менять, не мешая другим. Обычно static указывается для констант. Например:

```
static dword UMaskBit0[] = (
      0xFFFFFFFE,
      0xFFFFFFFD,
      0xFFFFFFFB,
      0xFFFFFFF7,
      0xFFFFFFEF,
      0xFFFFFFDF,
      0xFFFFFFBF,
      0xFFFFFF7F
);

dword CurBankPtr = 0;
```

При описании дизассемблера часто необходимы строковые константы, для объявления таких констант используется следующая синтаксическая конструкция:

```
strings (
      <имя 1-ой строки> = «<текст>»;
      …
      <имя n-ой строки> = «<текст>»;
);
```

Пример:

```
strings (
      UStrCommaAt       = «, @»
      UStrCommaR        = «, R»
      UStrCommaAtR      = «, @R»
      UStrCommaSharp    = «, #»
);
```

В описании можно использовать два типа комментариев: строчные и блочные. Строчные начинаются с «//» и заканчиваются концом строки. Блочные начинаются с «/*» и заканчиваются «*/». Например:

```
/*
            Файл описания групп декодирования параметров.
            Последнее обновление: 07.08.2001
*/

group G_D8 { // Первый операнд регистр A, второй в памяти по смещению.
```

При описании можно использовать директивы C-препроцессора для объявления макропеременных и условной компиляции. Каждая директива должна начинаться с символа '#', находящегося в начале строки.

2.2. Язык описания периферии

Разработанный язык описания периферийных устройств называется PPDL (Processor Peripherals Description Language). Этот язык базируется на декларативном описании основных составляющих внутренней периферии процессора и на описании алгоритмов их взаимодействия с помощью C-подобного языка.

Описание внутренней периферии состоит из следующих секций:

- описание контактов (пинов);
- описание событий;
- описание счетчиков;
- описание прерываний;
- описание тактов;
- описание циклов;
- описание инициализации.

Контакты используются процессором для взаимодействия с внешним окружением. При описании контактов необходимо указать имя контакта, его размерность, направление, тип линии, расположение на корпусе и тип контакта. Если этот контакт наложен на какой-нибудь регистр процессора, то это тоже следует указать. Если зависимость контакта и регистров процессора сложнее, чем простое наложение, то следует использовать секции Input и Output. Для описания контактов используется следующая синтаксическая конструкция:

```
pin(
    <имя контакта>, <размерность>, <направление>, <тип линии>,
    <расположение>, <тип контакта>, <ресурс процессора>)
{
    Input: {
        <алгоритм для чтения контакта>
    }
    Output: {
        <алгоритм для записи в контакт>
    }
};
```

Направление может быть следующим:

- in — значение контакта устанавливается внешним окружением;
- out — значение контакта устанавливается моделью процессора;
- bi — значение контакта устанавливается как внешним окружением, так и моделью процессора.

Тип линии может быть следующим:

- line — обычная линия;
- dot — инверсная линия (отрицание входного значения);
- clk — передний фронт (перепад с нуля на единицу);
- dotclk — задний фронт (перепад с единицы на ноль);
- bus — шина (несколько линий);
- dotbus — инверсная шина.

Расположение контакта на корпусе процессора может быть следующим:

- left — слева;
- right — справа;
- up — сверху;
- down — снизу.

Контакт может быть следующих типов:

- OD — обычный;
- OZ — принимает значения 0, 1 и Z;
- OC — открытый коллектор;
- OE — открытый эмиттер.

События — это реакции на изменения каких-либо ресурсов процессора. То есть обработка событий происходит только в том случае, если указанный ресурс (или выражение) изменило свое значение. При описании события указывается алгоритм и новое значение, при котором следует выполнить этот алгоритм. Для описания события используется следующая синтаксическая конструкция.

```
event(<ресурс или выражение>)
{
    <список значений>
}
```

Список значений представляет собой число и соответствующий ему алгоритм. Когда указанный ресурс процессора или выражение изменится и примет указанное значение, тогда исполнится указанный алгоритм.

Счетчики — это переменные, которые инкрементируются или декрементируются при внешних или внутренних воздействиях. Под внешними воздействиями понимаются изменения значений контактов. Под внутренними воздействиями — изменения ресурсов процессора, тактового генератора и независимого генератора, работающего с определенной частотой. В качестве внутреннего воздействия может также выступать переполнение другого счетчика. Значения счетчиков могут храниться как в независимой переменной, так и в ресурсах процессора. Можно задать алгоритм, который выполнится при достижении указанного лимита. Для описания счетчиков используется следующая синтаксическая конструкция:

```
counter(
<имя счетчика>, <источник>, <условие счета>, <предел>, <направление>, <ресурс процессора>)
{
    <алгоритм, выполняемый при переполнении>
}
```

Источником может быть другой счетчик, счетчик тактов (TacktsCounter), счетчик циклов (CyclesCounter) или мегагерцовый счетчик (количество МГц). Направление бывает двух типов: увеличение (inc) и уменьшение (dec). Условие счета, направление и ресурс процессора могут быть опущены. Счетчик при выполнении условия счета инкрементирует или декрементирует свое значение в зависимости от направления, при достижении указанного предельного значения выполняется алгоритм.

Прерывание — это выполнение определенного алгоритма при определенных условиях и продолжение исполнения программы с того места, где она была прервана. При описании прерываний необходимо указать:

- стратегию обработки приоритетов;
- общее условие разрешения прерываний;
- алгоритмы, которые будут исполняться при возникновении прерывания и при окончании обработки прерывания;
- условия возникновения каждого из прерываний и его приоритет;
- алгоритмы входа и выхода из каждого прерывания.

Приоритет определяет, будет ли прерываться обработка прерывания из-за другого прерывания и при каких условиях. Для указания стратегии обработки прерывания используется следующий синтаксис:

```
NewInterrupt = <стратегия>;
```

Стратегия может быть следующей:

- none — прерывания не прерываются;
- equal — прерывание прерывается прерыванием с большим или равным приоритетом;
- high — прерывания прерываются прерыванием с большим приоритетом.

Для указания общего условия разрешения прерываний и алгоритмов, которые будут выполняться при возникновении прерываний и при окончании обработки прерываний, используется следующая синтаксическая конструкция:

```
IntterruptProcessing(<общее условие разрешения прерываний>)
{
    Begin:  <общие действия по обработке прерываний>
    Enter:  <алгоритм, выполняемый при возникновении прерывания>
    Leave:  <алгоритм, выполняемый при окончании обработки прерывания>
}
```

Для описания конкретного прерывания используется следующая синтаксическая конструкция:

```
Interrupt(<имя прерывания>, <условие срабатывания>, <приоритет>, <вектор>)
{
    Enter:  <алгоритм, выполняемый при возникновении прерывания>
    Leave:  <алгоритм, выполняемый при окончании обработки прерывания>
}
```

Значение поля вектор передается в пользовательский алгоритм CallInt. Данный алгоритм должен быть в описании ядра процессора.

Инструкция исполняется процессором по тактам. За каждый такт процессор выполняет определенный алгоритм. Для описания тактов используется следующая синтаксическая конструкция:

```
tackts {
    <имя 1-го такта>: { <алгоритм 1-го такта> }
    …
    <имя n-го такта>: { <алгоритм n-го такта> }
};
```

Выполнение инструкции обычно разбивается на циклы, каждый цикл состоит из нескольких тактов. Обычно процессор за один такт параллельно выполняет несколько алгоритмов и изменяет значения своих контактов. Для описания циклов используется следующая синтаксическая конструкция:

```
cycle (<имя цикла>,
    (<имя 1-го ресурса>: (<список действий для 1-го ресурса>)),
    …
    (<имя n-го ресурса>: (<список действий для n-го ресурса>))
);
```

Список действий представляет собой перечисленные через запятую значения, которые надо присвоить ресурсу, или имя такта, алгоритм которого надо выполнить.

Для правильной работы процессора необходимо указать алгоритмы, которые будут выполняться при включении питания и при сбросе процессора. Кроме того, следует указать условие сброса процессора. Для этого используется следующая синтаксическая конструкция:

```
PowerOn
{
    <алгоритм>
}

Reset(<условие сброса>)
{
    <алгоритм>
}
```

Наряду с арифметическими и логическими операциями, присущими языку высокого уровня С, операторами условного выполнения и препроцессором в PPDL поддерживаются специальные операции и операторы, призванные облегчить и ускорить описание внутренней периферии процессора. Это опе-

рация выделения бит, операция объединения бит, операция определения перепада и оператор копирования кода с разными значениями константной переменной.

2.3. Интерфейс модели процессора

Интерфейс модели процессора базируется на универсальном интерфейсе моделируемого компонента. Таким образом, модели процессоров могут участвовать в процессе моделирования наряду с другими моделями устройств. Свойства модели процессора хранятся в структуре CProcessorData — производной от CDeviceData, а методы представлены интерфейсом IProcessor — производным от IDevice.

Для получения указателя на объект, представляющий модель процессора, необходимо сначала получить указатель на объект, представляющий модель соответствующего устройства. Так как CDeviceData является базовым классом для CProcessorData, то для получения указателя на CProcessorData необходимо воспользоваться методом QueryInterfaceData:

```
CDeviceData *pDD;
...
CProcessorData *pPD;
if (pDD->D->QueryInterfaceData(IID_IProcessor,
   reinterpret_cast<void **>(&pPD)) == S_OK) {
   // Взаимодействие с моделью процессора
   ...
}
```

Объект CProcessorData, вдобавок к унаследованным от CDeviceData, содержит следующие свойства:

- P — указатель на интерфейс с процессором;
- Instructions — количество выполненных инструкций.

Кроме свойств объекта в структуре хранится указатель на обработчик события OnBeforeExecute. Этот обработчик устанавливается подсистемой моделирования и вызывается моделью процессора перед выполнением каждой инструкции. В обработчик передается:

- указатель на кодовую память, из которой будет браться код очередной инструкции;
- количество тактов, затраченных на выполнение предыдущей инструкции;
- тип предыдущей инструкции.

С использованием этого обработчика осуществляется поддержка точек действия, пошаговое исполнение и мультипроцессорное моделирование.

Интерфейс объекта CProcessorData, вдобавок к унаследованным от CDeviceData, содержит следующие методы:

- Trace — выполнять инструкции, с вызовом OnBeforeExeceute перед выполнением каждой инструкции;
- Run — выполнять инструкции, с вызовом OnBeforeExecute перед инструкциями, для которых указанны точки действия;
- Stop — остановить выполнение инструкций;
- SetBreakpoint — установить признак наличия точки действия для инструкции по указанному адресу;
- ReSetBreakpoint — сбросить признак наличия точки действия для инструкции по указанному адресу.

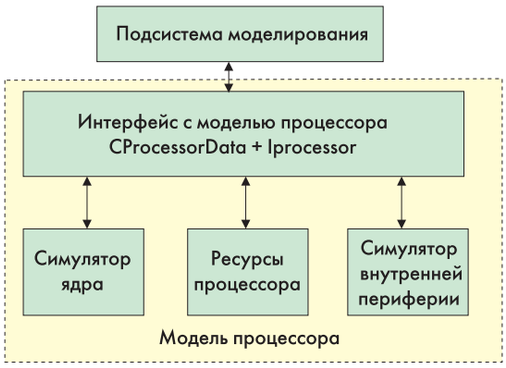

Рис. 3. Структурно-функциональная схема модели процессора

2.4. Структурно-функциональная схема модели процессора

Модель процессора состоит из трех основных блоков:

- ресурсы процессора;
- симулятор ядра;
- симулятор внутренней периферии.

С этими блоками взаимодействует подсистема моделирования через универсальный интерфейс моделируемого компонента. Схематически взаимодействие представлено на рис. 3. Для ускорения моделирования взаимодействие между симулятором ядра и симулятором внутренней периферии с ресурсами процессора происходит напрямую, без участия универсального интерфейса. Симулятор ядра напрямую не связан с симулятором внутренней периферии, это позволяет варьировать компоненты внутренней периферии, настраиваясь на конкретный процессор данного семейства.

Структура ресурсов процессора ничем не отличается от структуры ресурсов универсального моделируемого компонента. Симулятор ядра и симулятор внутренней периферии является детализацией алгоритма моделируемого компонента.

Симулятор ядра включает в себя подсистемы:

- декодирования параметров инструкций;
- исполнения алгоритмов инструкций.

Для взаимодействия с симулятором ядра используются методы Trace, Run, Stop, SetBreakpoint, ReSetBreakpoint и обработчики OnExecute и OnBeforeExecute.

Симулятор внутренней периферии состоит из подсистем:

- инициализации;
- обработки событий;
- моделирования счетчиков и таймеров;
- обработки прерываний;
- потактового исполнения.

Взаимодействие с симулятором периферии осуществляется через ресурсы процессора, сброс осуществляется вызовом метода PowerOn.

3. Средства автоматизации тестирования и верификации

3.1. Теневые команды

Полученные модели процессоров перед использованием необходимо протестировать на соответствие реальному образцу. Для решения этой задачи предлагается методика тестирования и верификации, основанная на применении «теневых» команд.

«Теневой» командой называется специальным образом оформленный комментарий в тексте программы, начинающийся с символа '$'. «Теневые» команды не влияют на полученный машинный код, но на основе их трансляторы, поддерживающие эту технологию, генерируют информацию, которая в дальнейшем используется средой моделирования.

Механизм действия «теневых» команд следующий: перед исполнением строки программы, для которой указана одна или несколько «теневых» команд, эти команды интерпретируются и производятся необходимые действия, затем выполняется машинный код, соответствующий текущей строке программы.

Для осуществления полного тестирования моделей процессоров разработан следующий набор «теневых команд»:

- S — установить значение ресурса процессора;
- T — проверить значение ресурса процессора, и если не совпадает с эталоном, остановить выполнение и сообщить об ошибке;
- ERR — остановить выполнение и сообщить об ошибке;
- OK — продолжить выполнение;
- B — остановить выполнение и сообщить об успешном прохождении тестов.

Команды ERR и OK используются при тестировании инструкций переходов и циклов.

Этап тестирования модели процессора следует разбить на два подэтапа: тестирование обычных инструкций и тестирование инструкций переходов и циклов.

Тестирование простых инструкций предлагается осуществлять по следующей схеме. Первая строка программы должна содержать «пустую» инструкцию, которая не выполняет никаких действий. Обычно в языках ассемблера эта инструкция имеет мнемонику NOP. В комментариях к этой строке должна быть указана «теневая» команда S, а затем, через запятую — список присвоений значений ресурсам процессора. Во второй строке должна быть указана тестируемая инструкция, а в третьей — опять инструкция NOP с «теневой» командой T и списком сравнений значений ресурсов с эталонными значениями. Завершаться каждый тест должен инструкцией NOP с «теневой» командой B.

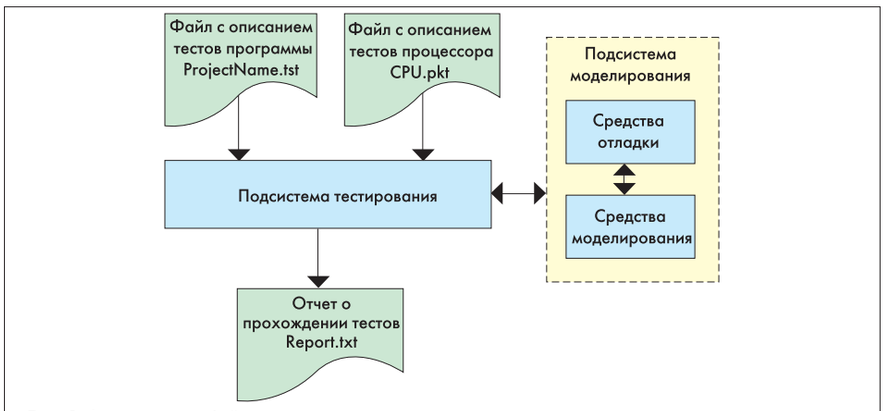

Рис. 4. Функционально-файловая структура подсистемы тестирования

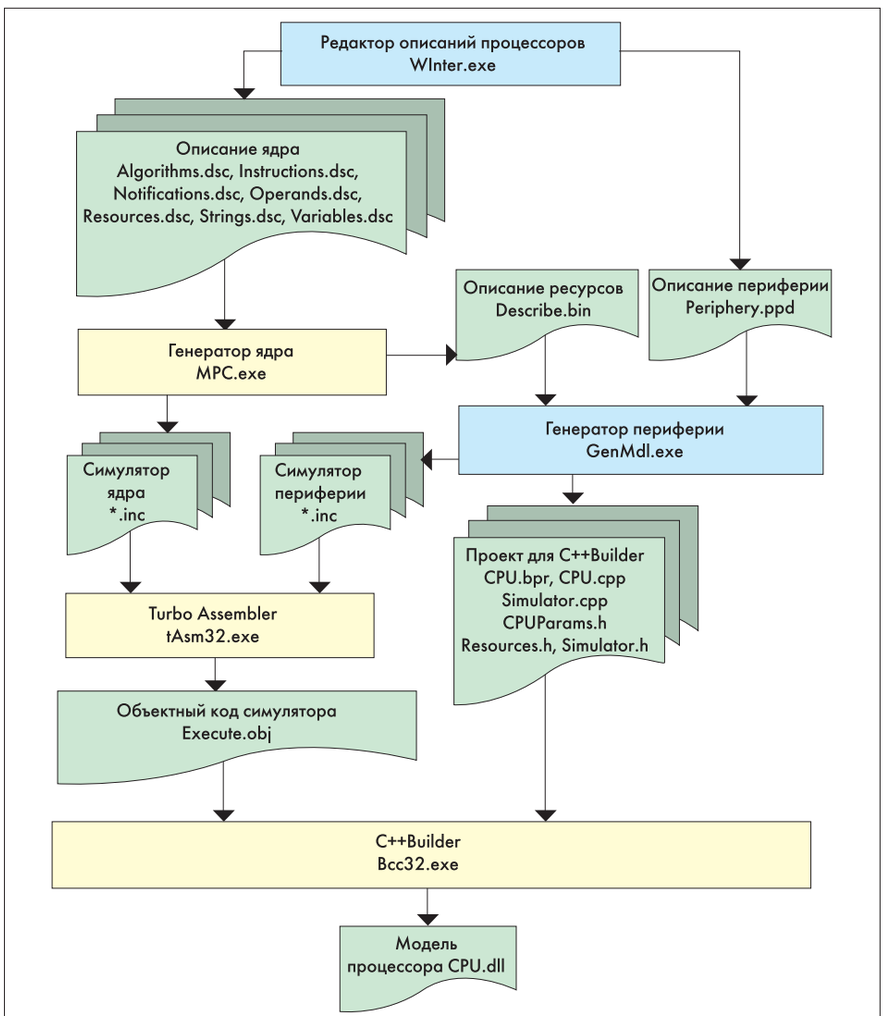

Рис. 5. Функционально-файловая структура подсистемы генерации моделей процессоров

Например, для тестирования инструкции сложения можно написать следующую программу:

```
nop             ;$S ax = 3, bx = 2
add       ax, bx
nop             ;$T ax = 5
nop             ;$B
```

Тестирование инструкций переходов и циклов предлагается осуществлять по следующей схеме. Если инструкция зависит от ресурсов процессора (например, условный переход или цикл), то в первой строке должна стоять команда NOP с «теневой» командой S. Во второй строке должна стоять тестируемая команда. Строку, на которую должно перейти управление, необходимо пометить «теневой» командой OK. Строки, на которые не должно перейти управление, необходимо пометить «теневой» командой ERR. После строки с «теневой» командой OK должна идти строка с инструкцией NOP и «теневой» командой B. Например, для тестирования инструкции перехода при установленном флаге Z, можно написать следующую программу:

```
        nop             ;$S flg.Z = 1
        breq  label1
        nop             ;$ERR
label1: nop             ;$OK
        nop             ;$B
```

Префикс flg при указании имени флага Z в «теневой» команде S использовался в связи с тем, что в процессоре AT90S2313 есть еще и регистр с таким же именем. Префиксы могут быть следующими:

- bit — бит;
- flg — флаг;
- reg — регистр;
- mem — память;
- pin — контакт.

Последний алгоритм модели, который нуждается в тестировании, это алгоритм инициализации при включении питания, тестировать этот алгоритм предлагается путем многократного прогона каждого теста без перезапуска среды моделирования.

3.2. Подсистема тестирования

Файловая структура подсистемы тестирования приведена на рис. 4.

Процесс тестирования начинается с загрузки описания тестов из соответствующего файла. Данный файл подготавливается с помощью подсистемы редактирования тестов.

При тестировании разрабатываемой модели процессора создается список тестовых исходных файлов, а затем эти файлы по очереди обрабатываются. При тестировании разрабатываемой программы обрабатывается только текущий проект.

После загрузки информации о тестах и подготовительного этапа подсистема тестирования посылает команду о начале исполнения в подсистему моделирования. Подсистема моделирования загружает тестируемый исполняемый файл. При необходимости этот файл строится с помощью средств компиляции.

Если указаны ограничения на время компиляции, то подсистема моделирования ждет указанное количество секунд, а затем, если процесс компиляции не был завершен, прекращает его, и в отчет записывается текст соответствующей ошибки.

После начала моделирования подсистема тестирования: проверяет, подходят ли тесты данному исполняемому файлу, устанавливает начальные значения и посылает команду о запуске моделирования в подсистему моделирования.

В процессе моделирования подсистема тестирования проверяет, не превышено ли время моделирования или не дошло ли исполнение программы до заданного в тесте адреса. Если условие выполнилось, то исполнение останавливается.

После окончания моделирования подсистема тестирования проверяет результаты исполнения с эталонными данными и формирует файл с отчетом о прохождении тестов. Если время исполнения программы превысило заданное, то эта ошибка также добавляется в отчет.

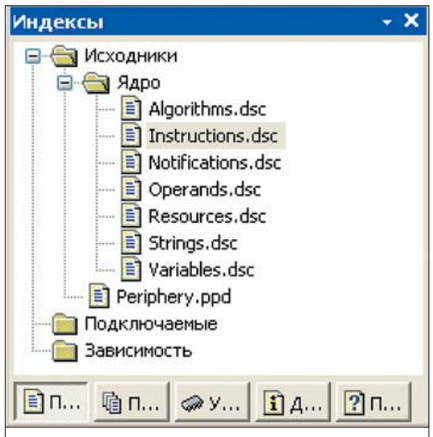

Рис. 6. Пример окна со списком файлов

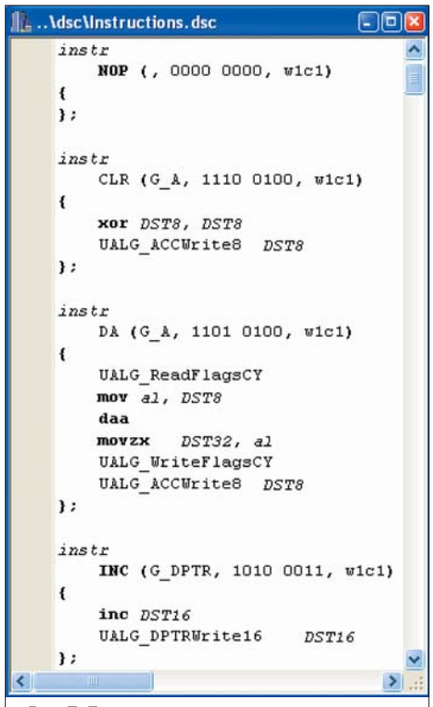

Рис. 7. Пример окна с исходным текстом описания

4. Технология разработки и отладки модели микроконтроллера

Разработку и отладку модели микроконтроллера предлагается осуществлять в четыре этапа:

- описание ядра процессора на языке PCDL;
- описание периферии процессора на языке PPDL;
- компиляция и исправление синтаксических ошибок в описании;
- тестирование и исправление алгоритмических ошибок в описании.

Для сокращения времени разработки предлагается использовать специальную интегрированную среду, построенную на базе комплекса WInter, которая обеспечивает эффективное редактирование описаний и автоматическую генерацию моделей (рис. 5). Для удобства редактирования интегрированная среда предоставляет пользователю окно со списком файлов (рис. 6) и окно с исходными текстами описания (рис. 7).

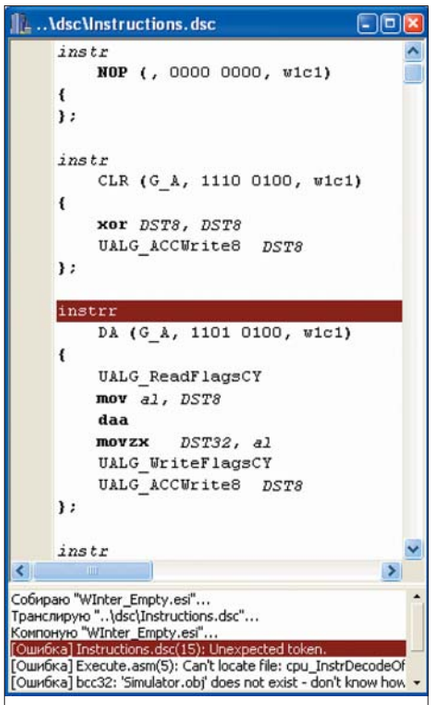

Рис. 8. Пример окна с индикацией ошибок компиляции

В папке «Ядро» в окне со списком файлов находятся исходные тексты описания ядра:

- Algorithms.dsc — файл с описанием пользовательских алгоритмов;
- Instructions.dsc — файл с описанием инструкций процессора;
- Notifications.dsc — файл с описанием алгоритмов после записи и до чтения;
- Operands.dsc — файл с описанием групп декодирования операндов инструкций;
- Resources.dsc — файл с описанием параметров и ресурсов процессора;
- Strings.dsc — файл с описанием строк (используется при описании дизассемблера);
- Variables.dsc — файл с описанием пользовательских переменных.

После редактирования описания ядра необходимо описать внутреннюю периферию

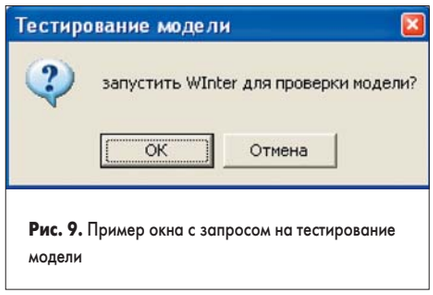

Рис. 9. Пример окна с запросом на тестирование модели

процессора, для этого следует из корня папки «Исходники» открыть файл Periphery.ppd, а затем в появившемся окне ввести исходный текст.

Для получения модели процессора следует выбрать пункт меню «Проект|Собрать» или нажать Ctrl+F9. Если в процессе компиляции будут найдены ошибки, то внизу окна с исходным текстом появится список ошибок, а в самом исходном тексте строка с ошибкой подсветится красным цветом (рис. 8).

Если ошибок при компиляции не будет обнаружено, то появится запрос на проведение тестирования модели (рис. 9).

После подтверждения запуска тестирования загрузится вторая копия комплекса WInter, настроенная на интерактивное тестирование модели. Для тестирования полученной модели на заранее подготовленных тестовых файлах необходимо выбрать пункт меню «Тестирование|Проверить файлы».

В результате будет показан отчет о прохождении тестов (рис. 10). В отчете указывается: сколько тестов прошло, сколько тестов не прошло компиляцию, сколько тестов не прошло загрузку и сколько тестов не прошло выполнение. Далее приведены основные параметры подсистемы тестирования: тип модели процессора, командная строка для запуска транслятора, каталог с тестами и расширение тестовых файлов. После параметров приводится подробный отчет о прохождении каждого из тестовых файлов. Символом «+» отмечаются файлы, при компиляции, загрузке и выполнении которых не было найдено ошибок. Символами «C1», «C2»,

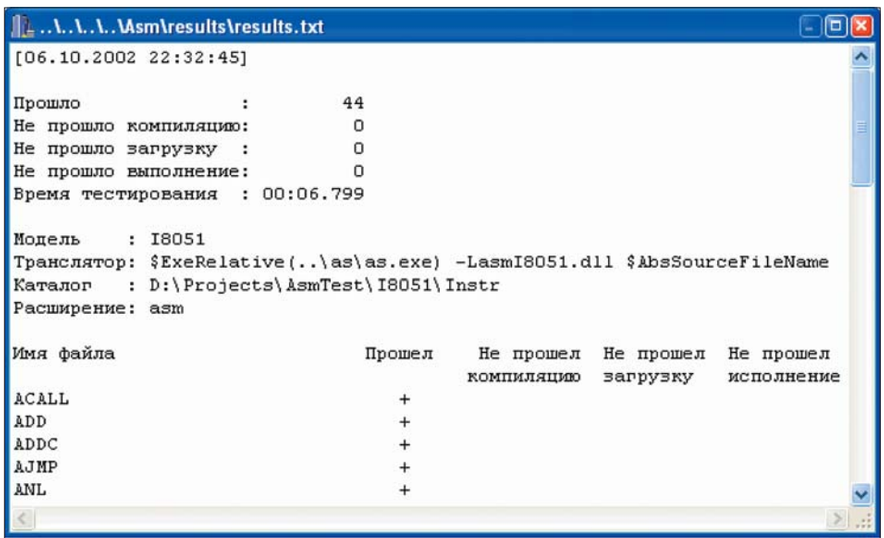

Рис. 10. Пример окна с отчетом о тестировании модели

«L1», «L2», «L3», «E1» и «E2» отмечаются файлы с ошибками. Расшифровка этих символов следующая:

- C1 — ошибки при компиляции;
- C2 — превышен лимит времени при компиляции;
- L1 — ошибки при загрузке исполняемого файла;
- L2 — сгенерирован неверный код;
- L3 — неверная информация о тестовых командах;
- E1 — не прошли заданные тесты;
- E2 — превышен лимит времени при исполнении.

Если какой-нибудь тест не прошел, то с помощью этой же среды тестирования можно исследовать причину неверного исполнения программы.

После нахождения ошибки в описании модели, следует выйти из среды тестирования и вернуться в среду описания, с помощью, которой необходимо исправить ошибку и повторить весь цикл компиляции и тестирования заново.

Заключение

Наличие разработанной технологии создания и отладки моделей микроконтроллеров позволяет получать модели в сроки от недели до месяца в зависимости от сложности микроконтроллера, уровня подготовки разработчика в программировании и понимания им алгоритмов функционирования микроконтроллера.

Литература

1. Долинский М. Концептуальные основы и компонентный состав IEESD-2000 — интегрированной среды сквозной совместной разработки аппаратного и программного обеспечения встроенных цифровых систем // Компоненты и технологии. 2002. № 8.
2. Долинский М., Литвинов В., Галатин А., Ермолаев И. HLCCAD — среда редактирования, симуляции и отладки аппаратного обеспечения // Компоненты и технологии. 2003. № 1.
3. Долинский М., Литвинов В., Толкачев А., Корнейчук А. Система высокоуровневого проектирования аппаратного обеспечения HLCCAD: тестирование // Компоненты и технологии. 2003. № 3.
4. Долинский М., Литвинов В., Галатин А., Шалаханова Н. Система высокоуровневого проектирования аппаратного обеспечения HLCCAD: открытый универсальный интерфейс моделируемых компонентов // Компоненты и технологии. 2003. № 4.
5. Долинский М., Литвинов В., Ермолаев И, Федорцов А. Система высокоуровневого проектирования аппаратного обеспечения HLCCAD: технология разработки потактовой модели микроконтроллера с использованием языка программирования высокого уровня на примере Intel 8051/Object Pascal // Компоненты и технологии. 2003. № 5.
6. Долинский М., Ермолаев И., Толкачев А., Гончаренко И. WInter — среда отладки программного обеспечения мультипроцессорных систем // Компоненты и технологии. 2003. № 2.
7. http://NewIT.gsu.unibel.by