|  |  |
| --- | --- |
| ДИСЦИЛИНА | **Схемотехника устройств компьютерных систем Часть 2** |
|  |  |
| ИНСТИТУТ | **ИТ** |
| КАФЕДРА | **вычислительной техники** |
|  |  |
| ВИД УЧЕБНОГО | **Лекция** |
| МАТЕРИАЛА |  |
| ПРЕПОДАВАТЕЛЬ | **Тарасов И.Е.** |
|  |  |
| СЕМЕСТР | 3 |
|  |  |

2

11. Процессорное ядро

11.1. Основные сведения о процессорных ядрах.

Процессоры широко применяются в самых разных устройствах и являются основой вычислительных систем. Удешевление процессоров по мере развития микроэлектронной технологии привело к тому, что даже в простых устройствах используются недорогие процессоры. Вместо изменения схемы (т.е. повторного изготовления печатной платы и монтажа компонентов) для коррекции работы устройства часто достаточно изменить программу управляющего им процессора. Это существенно быстрее и открывает перед разработчиками больше возможностей, поэтому недорогой процессор часто используется для управления самыми разными приборами, образуя огромный класс *встраиваемых* (embedded) устройств.

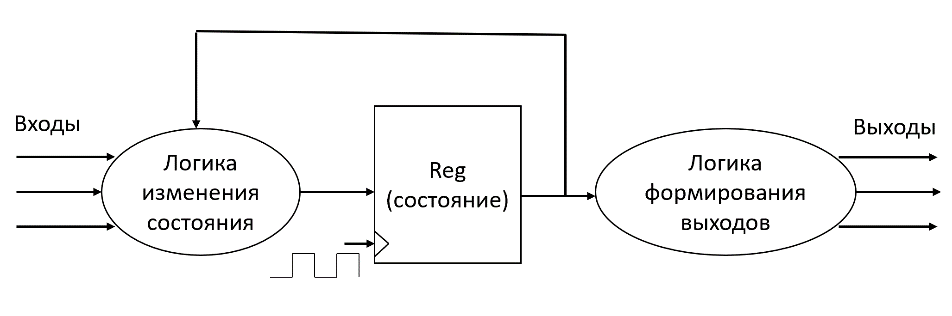
Процессор не обязан быть единственным в системе. Например, современный автомобиль содержит десятки процессоров различных типов. Кроме устройств, которые очевидно распознаются как элементы бортового компьютера (например, приборная панель автомобиля), простые процессоры используются для управления стеклоподъемниками, освещением, зажиганием и другими функциями автомобиля. Все эти процессоры имеют разные требования по производительности, объему памяти и подключаемым устройствам.

Поэтому в настоящее время вполне актуальна разработка процессоров для решения отдельных задач. Не следует полагать, что такие системы, как x86, ARM, RISC-V, AVR в определенном сочетании достаточны и оптимальны для решения любой задачи. Разработка новых процессорных архитектур продолжается в мировой микроэлектронике до сих пор, поэтому получение такого навыка может оказаться полезным.

Понятие «процессорное ядро» до сих пор не получило однозначного определения. Тем не менее, при проектировании вычислительных устройств можно разделить процессор на внутренние компоненты и подключенные к ним периферийные устройства. На практике оказывается, что имея некий минимальный набор узлов («ядро»), можно несложным образом подключать к ним дополнительные модули, например, UART, SPI или ШИМ. Такие модули ничего не изменяют в конструкции ядра. Поэтому под ядром процессора понимается набор, который обычно включает в себя доступные программисту регистры, арифметико-логическое устройство и систему управления выполнением команд. Эти компоненты должны правильно взаимодействовать друг с другом, поэтому они требуют согласованного проектирования.

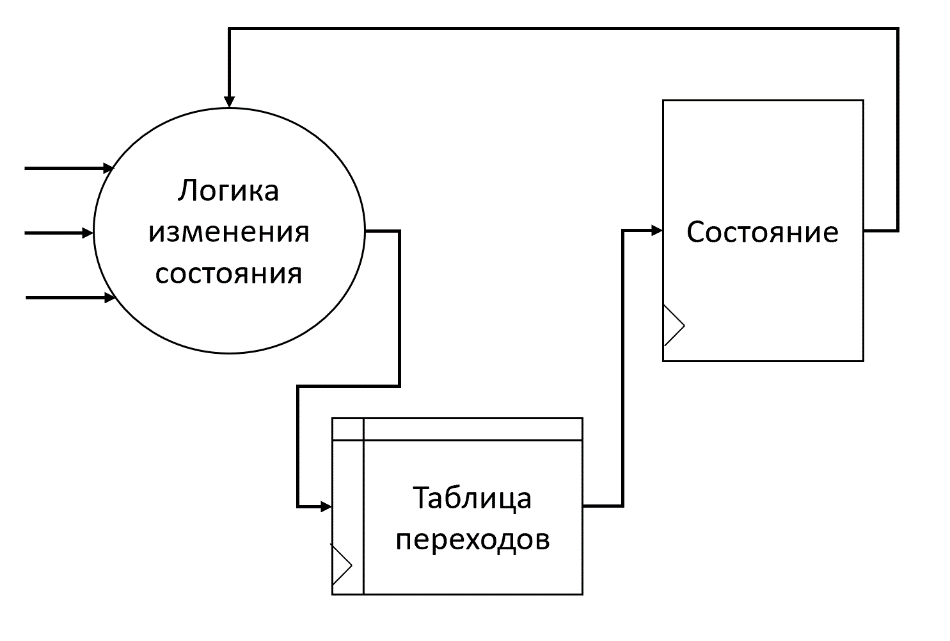
11.2. Преобразование конечного автомата в процессор.

Для понимания принципов работы процессора можно вернуться к понятию конечного автомата, который показан на рис. 11.1.



*Рис. 11.1. Конечный автомат*

Корректируя схему «логика изменения состояния», можно добиться изменения поведения конечного автомата. Это неудобно, потому что для реализации сложного алгоритма потребуется постоянно модифицировать эту схему и повторять моделирование или загрузку конфигурации ПЛИС. В то же время видно, что формирование выходных сигналов осуществляется довольно регулярным способом – на основе анализа состояния автомата. Сделать КА более гибкой можно, если добавить таблицу с правилами перехода между состояниями, как показано на рис. 11.2.



*Рис. 11.2. Преобразование конечного автомата*

Показанная на рис. 11.2 схема очень близка к процессору. Действительно, представим, что таблица переходов представляет собой память программы. Входной сигнал является адресом этой памяти, а содержимое (данные) – номером состояния, в который должен перейти КА. Тогда можно будет изменять таблицу переходов, записывая в память новые значения, и автомат начнет переходить в совершенно новые последовательности состояний.

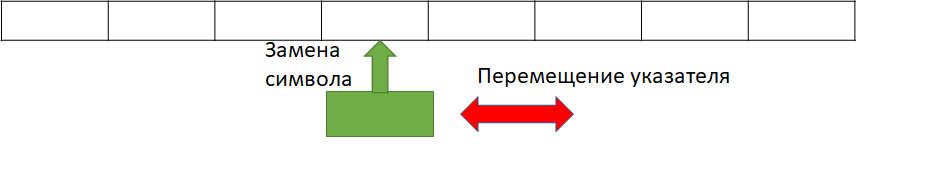
При рассмотрении процессора важным понятием является абстрактная модель – машина Тьюринга. Она представляет собой бесконечную ленту с «символами», под которыми понимается некое абстрактное понятие данных, которое может принимать любую требуемую форму (например, байт, 32 или 64 бита и т.д.). Имеется указатель, который может:

1. Прочитать символ.

2. Заменить его (или не заменять).

3. Перейти к другой позиции на ленте.

В целом считается, что любой алгоритм может быть выполнен с помощью машины Тьюринга. Это утверждение будет убедительнее, если учесть, что машина может выполнять с символом любое требуемое действие, переходить на другой символ по любому требуемому алгоритмом правилу, а сама лента имеет сколько угодно ячеек.



*Рис. 11.3. Абстрактная модель – машина Тьюринга*

Поскольку машина Тьюринга предполагает бесконечную ленту, она не может быть реализована на практике. Тем не менее, если у вычислительного устройства есть достаточно памяти для выполнения конкретного алгоритма, то в данном частном случае он может рассматриваться как подобие машины Тьюринга. Можно представить, что лента с символами – это память с данными, действия над символами задаются набором команд арифметико-логического устройства, а перемещение указателя – командами вычисления адреса и командами перехода к новой операции.

Для процессора рассматриваются два важных понятия, которые и определяют его возможности:

1. Архитектура системы команд (АСК) – какие команды есть у процессора.

2. Микроархитектура – как эти команды выполняются на аппаратном уровне.

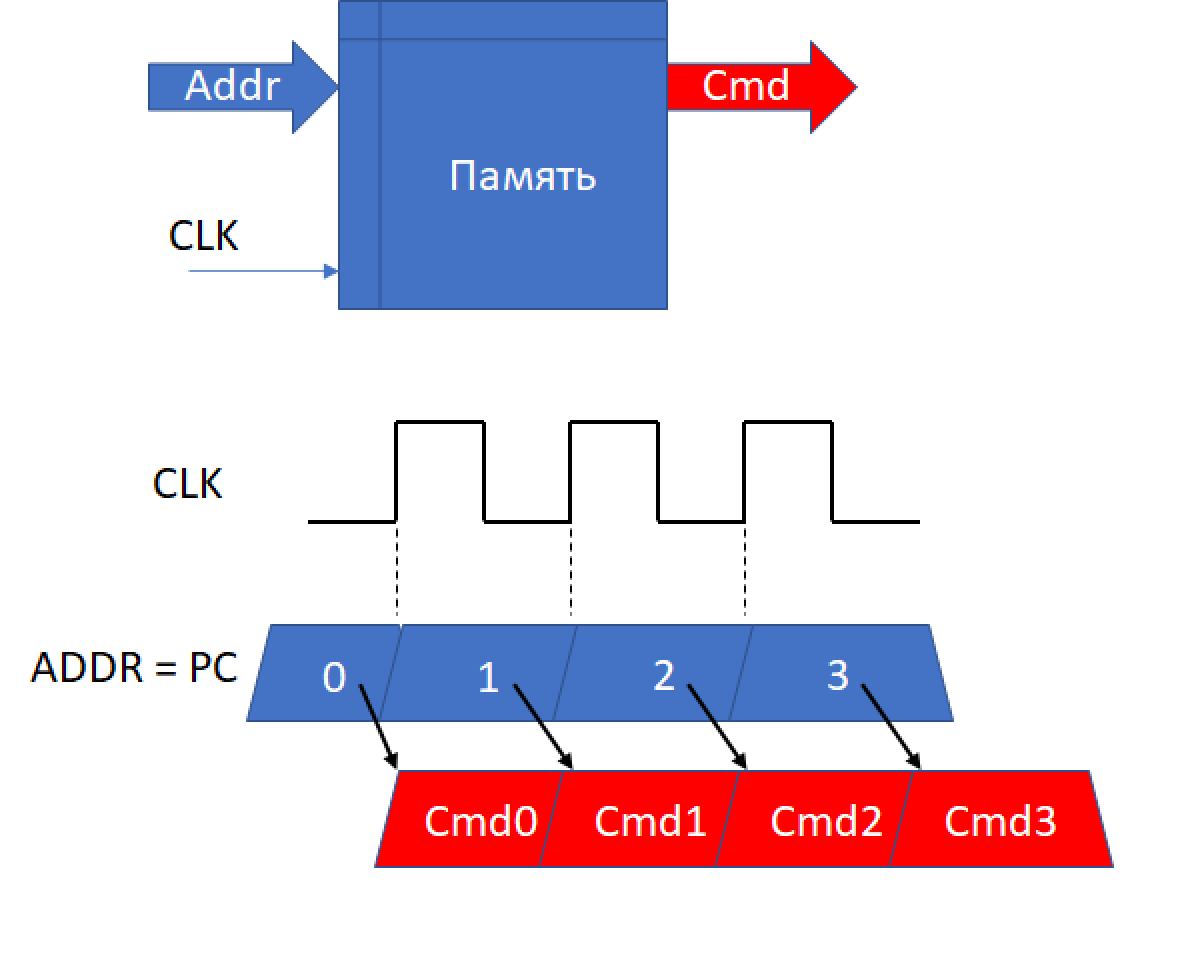
В целом микроархитектура и АСК не обязаны строго соответствовать друг другу. Например, один из первых вариантов процессора Intel 8051 использовал команды, выполняющиеся за 12 или 24 такта (1 или 2 «машинных цикла», каждый из которых состоял из 12 тактов). Процессор оказался настолько популярным, что многие разработчики продолжали его использовать, даже когда оригинальный вариант перестал выпускаться. Ряд компаний выпустил свои аналоги 8051, причем какие-то полностью сохраняли поведение, а другие сохраняли только систему команд, которые стали выполняться уже за 1 или 2 такта. Это пример того, как при одной и той же системе команд изменилась микроархитектура.

С другой стороны, некоторые команды требуют определенной микроархитектуры. Например, операции с данными в памяти требуют и чтения, и записи памяти, а какая-то микроархитектура может допускать только одно действие с памятью для каждой команды.

Кроме того, микроархитектуры могут по-разному сочетаться с разными наборами команд. Взаимное согласование микроархитектуры и системы команд является важным процессом в проектировании ядра и представляет собой в чем-то элемент искусства, поскольку трудно сформулировать четкие критерии оптимальности итоговой конструкции.

11.3. Двухтактная архитектура.

Двухтактная архитектура процессора является наиболее простой для понимания. Как следует из названия, она реализует работу процессора за два такта. Первый – это чтение команды из памяти. Этот процесс иллюстрирован рис. 11.4.



*Рис. 11.4. Чтение команд из памяти*

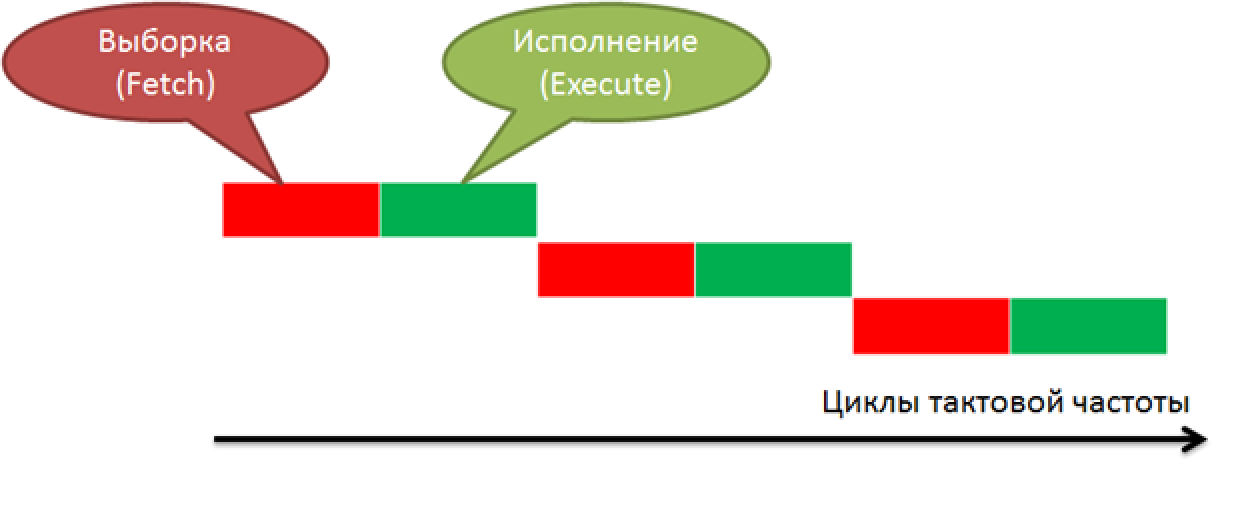
Работа памяти происходит по обычным правилам. По фронту тактового сигнала на выходе памяти появляется содержимое ячейки с номером addr. В случае процессора адрес – это номер команды. Традиционно этот адрес хранится в регистре PC (Program Counter, счетчик команд). Имя регистра общеупотребительно, а самым известным исключением является аналогичный регистр IP (Instruction Pointer, указатель инструкций), использованный в процессоре Intel 8086. Корпорация Intel поясняет это тем, что в процессоре 8086 был реализован внутренний буфер для накопления команд, поэтому регистр IP содержал адрес команды, которая выполнялась в данный момент, причем в это время процессор мог выполнять чтение другой команды, и адрес, подаваемый им на внешнюю память, мог быть другим.

Когда команда уже известна, следующим тактом можно выполнить ее. С точки зрения языков описания аппаратуры, можно реализовать требуемые действия в виде оператора case, где выбор производится на основе кода прочитанной команды. Таким образом, работа простого ядра состоит из повторения двух действий:

1. Чтение команды, или ее «выборка» (fetch).

2. Исполнение команды (Execute).

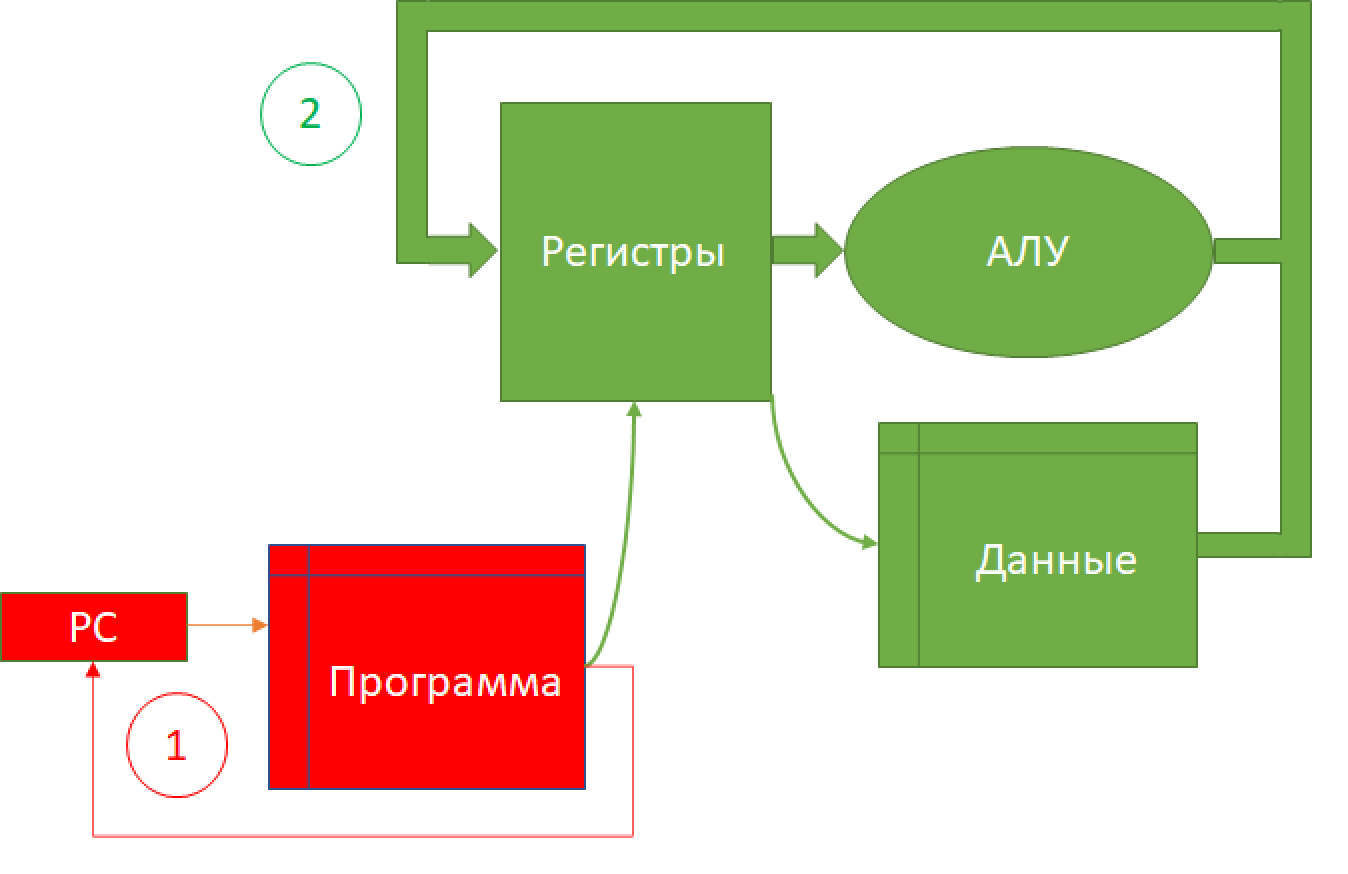
Последовательность таких циклов показана на рис. 11.5.



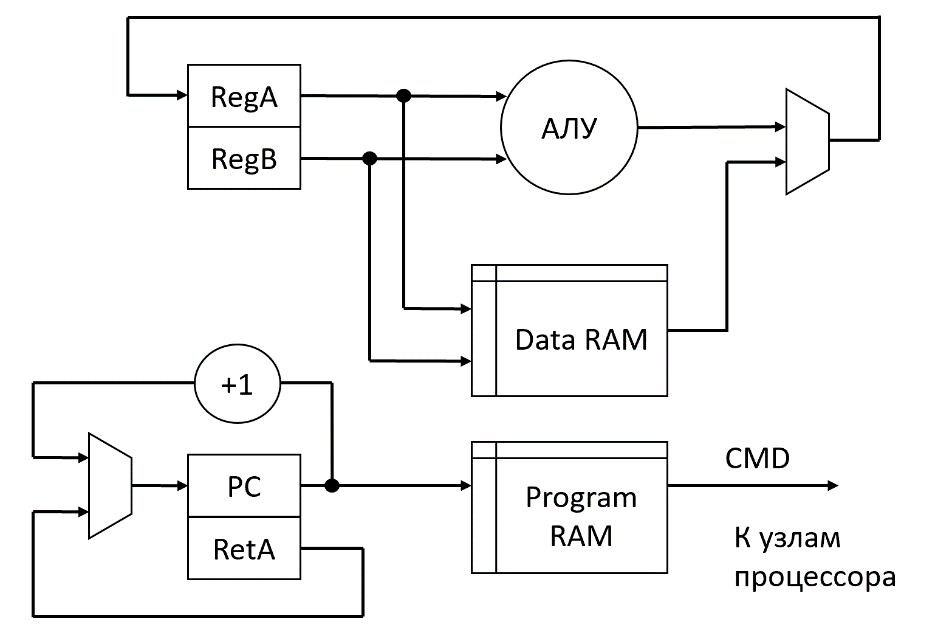
*Рис. 11.5. Двухтактная работа процессора*

Под выполнением команды понимается запись новых значений во все регистры, которые этого требуют. Особенно важно обновить регистр PC, иначе процессор остановится на определенной ячейке памяти. В дальнейшем будет показано, что обновление PC в некоторых случаях можно производить и не дожидаясь получения команды.

На рис. 11.6 показана обобщенная структурная схема простого процессорного ядра с двухтактным циклом работы, который уже имеет все основные элементы. На рис. 11.7 эти элементы детализированы. Например, «регистры» представлены в виде двух регистров – RegA и RegB. Практически используемые процессоры имеют существенно больше регистров, однако простой пример позволяет нагляднее продемонстрировать основные правила проектирования.



*Рис. 11.6. Структурная схема процессора с двухтактным циклом работы*



*Рис. 11.7. Пример простого процессорного ядра*

В схеме, показанной на рис. 11.7, есть важная особенность. Если процессор выполняет команду перехода, без ее чтения непонятно, из какого адреса необходимо читать следующую команду. Пока для простоты можно принять, что каждая команда процессора будет выполняться за два такта. На первом такте процессор читает команду из ячейки памяти программ по адресу PC. На втором такте, имея в своем распоряжении код команды, на его основе можно определить, какой результат (и в какой регистр) записывать, а также адрес следующей команды.

Часто процессор выполняет линейную последовательность команд, поэтому можно заранее предполагать, что будет выполнено действие PC = PC+1. Это позволяет проектировать конвейеризованные процессорные ядра, которые начинают выполнение следующей команды до завершения предыдущей.

Процессор, показанный в качестве примера, разрабатывается следующим образом. Интерфейс процессора объявляется в модуле:

module cpu\_top(

input clk\_in, // тактовый сигнал

input reset, // сброс

input rx, // UART

output tx, // UART

input [3:0] sw, // входы - переключатели

output [3:0] led, // выходы - светодиоды

// интерфейс загрузки программы

input cmdwe,

input [15:0] cmdaddr,

input [35:0] cmddata

);

В процессорном ядре добавлен интерфейс UART, который с большой вероятностью понадобится впоследствии, хотя и не задействован в приведенном примере. Кроме того, если описать неподключенное ядро процессора, синтезатор исключит его из проекта. Поэтому требуется добавить выходы какого-либо простого периферийного устройства – в примере это 4-разрядная шина led.

Чтобы синтезатор не заменил память на небольшой набор констант, память программ должна иметь внешний интерфейс, который в принципе способен записать в нее новые значения. Если оперировать в модели процессора небольшими программами, синтезатор скорее всего будет оптимизировать их содержимое. Если же у памяти программ объявлен второй интерфейс, по которому может производиться запись, такая оптимизация уже не будет состоятельной, поскольку содержимое памяти, с точки зрения синтезатора, сможет впоследствии измениться.

В модуле процессора необходимо сделать объявления регистров и параметров, как показано ниже. Директивы define позволяют впоследствии пользоваться мнемоническими обозначениями, быстро изменяя размеры памяти при необходимости. Это не строго обязательный, но настоятельно рекомендуемый подход в программировании – вместо чисел, смысл которых при последующем обращении к этому коду может оказаться непонятным, используются имена, напрямую указывающие на смысл этого параметра.

Показанные на структурной схеме регистры процессора определены с помощью объявлений reg. Память программ и память данных представлены отдельными двумерными массивами, что позволяет синтезатору распознать шаблон модуля памяти.

`define PCWIDTH 16

`define PROGRAMSIZE 1024

`define DATASIZE 1024

reg [`PCWIDTH - 1 : 0] pc, newpc, RetReg;

reg [35:0] Program [`PROGRAMSIZE - 1 : 0];

reg [35:0] cmd;

reg [31:0] Data [`DATASIZE - 1 : 0];

reg [31:0] RegA, RegB, newRegA, newRegB, dmem;

reg [3:0] RegLED;

initial begin

$readmemh("program.mem", Program);

end

Установка модуля тактового генератора происходит с помощью генератора IP-ядер, входящего в состав САПР. Аппаратный компонент тактового генератора настоятельно рекомендуется, если проект будет программироваться в ПЛИС. Для моделирования этот шаг необязателен. В приведенном ниже примере тактовый сигнал clk, который будет использован в качестве основного для процессора, напрямую соединен с входом clk\_in. Если же тактовый генератор действительно будет установлен, строку с таким прямым соединение необходимо закомментировать, а в объявлении IP-ядра тактового генератора вместо .clk\_out1(null) поместить стоящий за ним закомментированный текст.

clk\_wiz\_0 inst\_gen

( .clk\_out1(null), // .clk\_out1(clk),

.clk\_in1(clk\_in)

);

assign clk = clk\_in; // это объявление не требуется, если подключен выход генератора

Для демонстрации работы процессора можно выбрать небольшую систему команд. Она также задается мнемоническими именами с помощью директивы define.

`define NOP 0

`define ATOB 1

`define BTOA 2

`define ADD 3

`define SUB 4

`define OUTLED 5

`define INSW 6

`define WRITEMEM 7

`define READMEM 8

`define RET 9

// Специальные случаи

// 1111xxxxxx call xxxxxx

// 1110xxxxxx jmp xxxxxx

// 1101xxxxxx jmp xxxxxx if A = 0

// 0001xxxxxx загрузка в А xxxxxx

// 0010xxxxxx загрузка в B xxxxxx

В конце списка показаны специальные случаи, которые содержат в поле команды литералы, т.е. двоичные данные, воспринимаемые *буквально* (literally), а не как код определенного действия. Если старшие 4 бита команды принимают показанные значения, то в остальных 32 содержится число, требуемое этой команде. Использование 36-разрядной команды довольно расточительно, однако это позволяет не усложнять пример процессора.

Далее необходимо описать поведение основных компонентов процессора. Для упрощения понимания и отладки кода можно воспользоваться тем же приемом, что и при разбиении описания конечного автомата на несколько процессов. В одном процессе, выполняющемся синхронно по фронту тактового сигнала, в регистр записывается «новое значение». Чему равно это значение, определяет другой процесс, выполняемый в виде комбинационного выражения. Например, процесс обновления регистра pc описан в простейшем виде – по сигналу сброса записывается 0, иначе newpc.

always @(posedge clk)

begin

if (reset) pc <= 0;

else pc <= newpc;

end

Значение сигнала newpc на каждом такте описано ниже. Случаями, когда новое значение вычисляется не как pc + 1, являются команды перехода, условного перехода, вызова подпрограммы и возврата из подпрограммы. В первых трех случаях новое значение – это разряды 15:0 команды, а при возврате из подпрограммы – значение регистра RetReg.

always @ \*

begin

if ((cmd[35:33] == 3'b111) || ((cmd[35:32] == 4'b1101) && (RegA == 0))) newpc <= cmd[15:0];

else if (cmd == `RET) newpc <= RetReg;

else newpc <= pc + 1;

end

Регистр адреса возврата принимает значение pc + 1, если процессор выполняет команду вызова подпрограммы. В этом случае нужно запомнить в регистре адрес, с которого следует продолжать выполнение программы. В данном простом примере регистр адреса возврата только один, поэтому вызывать вложенные подпрограммы нельзя. В практически используемых процессорах адреса возврата хранятся на стеке, что обеспечивает последовательное помещение адресов на стек и снятие их со стека в том же порядке.

always @(posedge clk)

begin

if (reset) RetReg <= 0;

else if (cmd[35:32] == 4'b1111) RetReg <= pc + 1;

end

Память программ описана в виде массива ячеек разрядностью 36 бит. Чтобы этот массив не оказался заменен на нули или небольшую таблицу, для него описывается запись нового значения выражением if (cmdwe) Program[cmdaddr] <= cmddata; В этом случае все ячейки памяти будут подключены к внешним сигналам и оптимизировать такое выражение на этапе синтеза окажется невозможным.

always @(posedge clk)

begin

if (cmdwe) Program[cmdaddr] <= cmddata;

cmd <= Program[pc];

end

Описание регистров данных выполнено по схожей схеме с описанием регистра pc. В регистр записывается 0 в момент сброса или «новое значение». Для упрощения описания в синхронном процессоре описана также и загрузка литерала, чтобы подчеркнуть важность этой операции.

always @(posedge clk)

begin

if (reset) RegA <= 0;

else if (cmd[35:32] == 4'b0001) RegA <= cmd[31:0];

else RegA <= newRegA;

end

always @(posedge clk)

begin

if (reset) RegB <= 0;

else if (cmd[35:32] == 4'b0010) RegB <= cmd[31:0];

else RegB <= newRegB;

end

Описание арифметико-логического устройства представляет собой комбинационное выражение, которое вычисляет результат, помещаемый впоследствии в RegA и в RegB (для каждого регистра используется свой процесс always). Такой подход позволяет наглядно проверить отсутствие конфликта сигналов, если, например, в результате анализа двух процессов окажется, что в RegA должны быть записаны два значения одновременно.

always @ \*

begin

case (cmd)

`NOP : newRegA <= RegA;

`BTOA : newRegA <= RegB;

`ADD : newRegA <= RegA + RegB;

`SUB : newRegA <= RegA - RegB;

`INSW : newRegA <= sw;

`READMEM : newRegA <= dmem;

default : newRegA <= RegA;

endcase

end

always @ \*

begin

case (cmd)

`NOP : newRegB <= RegB;

`ATOB : newRegB <= RegA;

default : newRegB <= RegB;

endcase

end

Память данных так же, как и память программ, описана в виде двумерного массива. Особенностью памяти является то, что по фронту тактового сигнала нельзя одновременно и прочитать значение какой-то ячейки, и записать ее в регистр назначения. По первому фронту требуемое значение появится на выходе блока памяти, и только следующим тактом оно может быть записано в требуемый регистр. Такое поведение характерно для многих процессорных ядер.

always @(posedge clk)

begin

if (cmd == `WRITEMEM) Data[RegB] <= RegA;

dmem <= Data[RegA];

end

Описание периферийных устройств состоит из описания регистра, хранящего данные, записываемые из RegA. В систему команд процессора добавлена специальная команда OUTLED, хотя обычно периферийные устройства не имеют специальных команд для доступа к их конкретному регистру. Следует использовать подключение периферийных устройств к системной шине процессора, определяя отдельно адрес устройства, и отдельно данные для него. Однако для простого примера специальная команда упростит запуск процессорного ядра и понимание принципов его работы.

always @(posedge clk)

begin

if (cmd == `OUTLED) RegLED <= RegA[3:0];

end

assign led = RegLED;

Можно видеть, что описание разбито на описание поведение регистра и его подключение к выходным сигналам оператором assign. Это сделано потому, что выходы led не имеют спецификатора reg и записать в них данные по фронту тактового сигнала из-за этого невозможно. Добавить спецификатор reg в описание верхнего уровня в принципе можно, однако это не даст возможности прочитать значение этого сигнала, если впоследствии при модификации процессора возникнет такая необходимость. Показанный пример является более корректным с точки зрения сопровождения проекта.

11.4. Организация моделирования процессора

Чтобы проверить работу описанного процессора, можно воспользоваться поведенческим моделированием. Для этого необходимо создать файл модели, содержащий внутри подключенный модуль процессора и описание поведение внешних сигналов, подаваемых на этот модуль. Файл модели не имеет внешних сигналов, поэтому их список пуст. В показанном ниже примере объявлен модуль cpu\_top\_tb, где символы \_tb не являются обязательными, но стилистически показывают, что это не файл для синтеза схемы, а файл модели (сокращение tb от «testbench»). Внутри необходимо объявить те же сигналы, что используются в процессоре в качестве внешних, однако для входов процессора необходимы сигналы reg (поскольку они будут хранить значения, подаваемые на входы процессора), а сигналы, подключаемые к выходам, объявлены как wire (поскольку управляющие ими регистры находятся внутри моделируемого процессора).

module cpu\_top\_tb(

);

reg clk\_in;

reg reset;

reg rx;

wire tx;

reg [3:0] sw;

wire [3:0] led;

reg cmdwe;

reg [15:0] cmdaddr;

reg [35:0] cmddata;

После объявления сигналов к ним подключается процессор. Указать симулятору Verilog, что требуется определенный файл, можно по имени модуля этого файла. В данном случае это cpu\_top, поскольку именно такое имя было приведено в примере выше. Запись «cpu\_top uut» можно сопоставить с объявлением в языке Си «int x», где сначала указан тип создаваемого элемента, а затем выбранное для него имя. В данном случае cpu\_top – это «тип» модуля, а uut – выбранное имя. Оно также не регламентируется строго, а означает в данном случае сокращение от Unit Under Test.

cpu\_top uut (

.clk\_in(clk\_in),

.reset(reset),

.rx(rx),

.tx(tx),

.sw(sw),

.led(led),

.cmdwe(cmdwe),

.cmdaddr(cmdaddr),

.cmddata(cmddata)

);

После «установки» в модель разработанного процессора необходимо описать сигналы, подаваемые на него. Обязательным является описание тактового сигнала. Оно показано в процессе always. Сигналу clk\_in последовательно присваиваются значения 0 и 1, с паузой в 5 «единиц времени» между ними. По умолчанию в симуляторах это наносекунды. Такое назначение было бы невозможным для синтезированного кода (вызывая «короткое замыкание» 0 и 1 в одной и той же точке), однако при моделировании симулятор Verilog обрабатывает данные конструкции иначе. Процесс с ключевым словом always будет повторяться бесконечно по мере моделирования.

always

begin

clk\_in <= 1'b0;

#5;

clk\_in <= 1'b1;

#5;

end

Начальная установка сигналов процессора определяется процессом initial. Этот процесс, в отличие от always, выполняется только один раз при старте моделирования. Необходимо установить начальные значения для всех сигналов, в том числе и тех, которые не будут использоваться для проверки работы процессора. Это необходимо потому, что неопределенное состояние какого-либо сигнала вызовет при моделировании неопределенное состояние и того сигнала, который каким-либо образом от него зависит. В результате каскадного распространения таких неопределенностей симулятор может показать неопределенное состояние всего процессора.

initial begin

rx <= 1'b1;

sw <= 4'b0000;

cmdwe <= 1'b0;

cmdaddr <= 16'h0000;

cmddata <= 36'h000000000;

reset <= 1'b1;

#20;

reset <= 1'b0;

end

Для сигнала сброса задается установка в 1 на 20 нс с последующим переходом в 0. Это обеспечивает сброс регистров процессора.

В синтезированном коде была использована специальная директива Verilog readmemh. Она позволяет инициализировать описанную память из текстового файла. Можно задать содержимое памяти программ, инициализируемое файлом «program.mem», в котором будет выполняться простая программа, складывающая два числа, выводящая результат на светодиоды в двоичном коде, а затем демонстрирующая вызов подпрограммы и возврат из нее.

100000002 // RegA = 2

000000001 // B <- A

100000003 // RegA = 3

000000003 // A = A + B

000000005 // A->LED

f00000010 // call 0x10

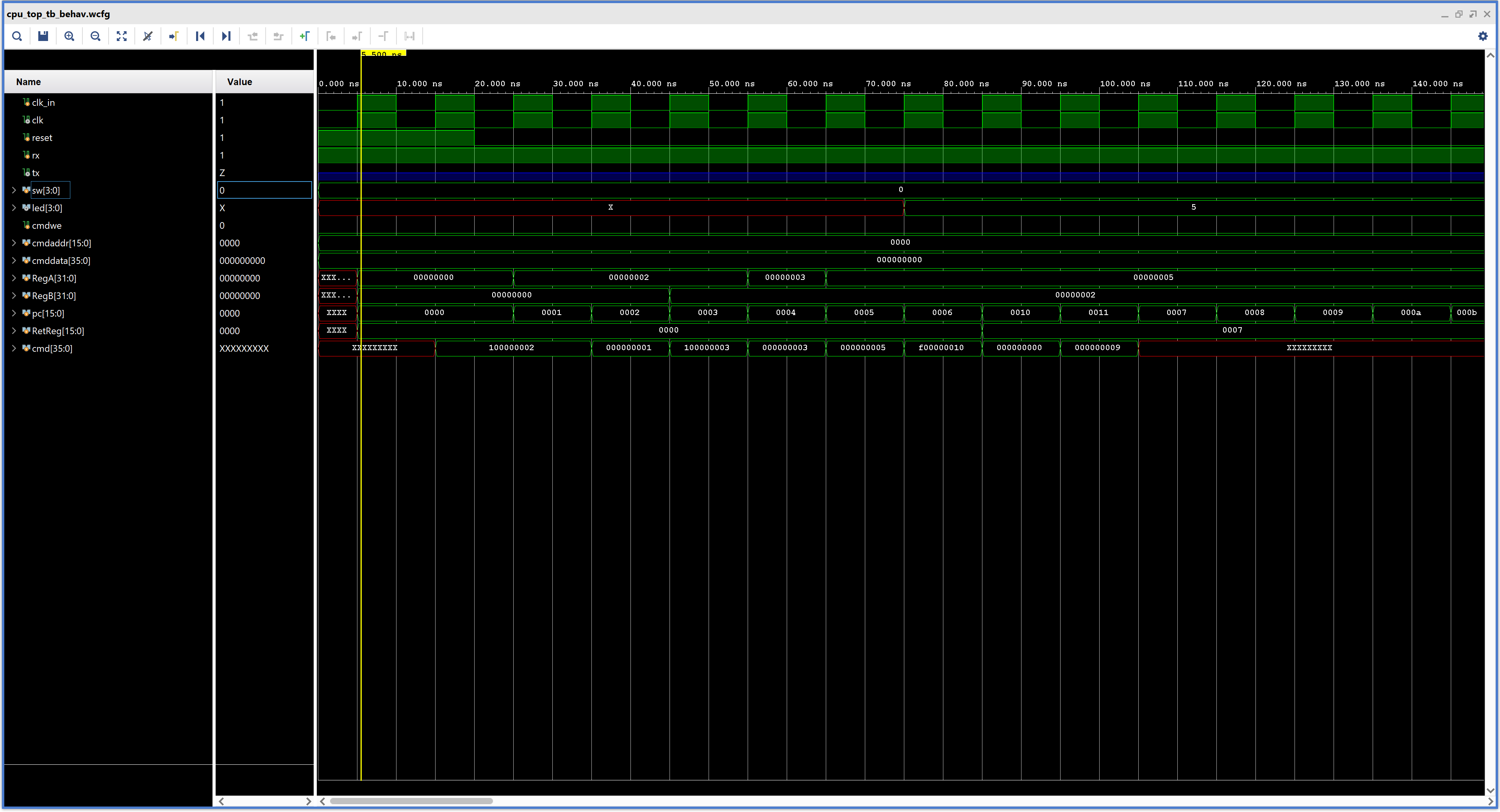
000000000 // nop

@10 // символ @ задает адрес, с которого следует продолжать

// заполнение памяти

000000009 // RET

Завершив описание, можно запустить моделирование. Результаты моделирования показанного примера приведены на рис. 11.8.



*Рис. 11.8. Пример моделирования процессорного ядра*

Таким образом, показан пример разработки простого процессорного ядра и организации его моделирования. Приведенный пример специально выбран крайне простым (однако способным выполнить типичные для процессора операции) и очевидно требует развития.

11.5. Выводы по разделу.

Процессорное ядро является полезным на практике инструментом для управления работой других узлов проекта. В отличие от конечного автомата, изменение работы процессора происходит не путем коррекции схемы, а путем замены содержимого памяти программ. Это можно сделать существенно быстрее, чем произвести еще одну итерацию синтеза, трассировки и программирования ПЛИС. Поэтому для организации сложных протоколов или задания последовательности смены сигналов часто полезно иметь в схеме процессор, даже если он и не является высокопроизводительным или совместимым с распространенными языками программирования.

При проектировании процессора важно начинать с общей регистровой структуры и формата команды. В следующей лекции будет проведена систематизация этих вопросов.

При описании RTL-представления следует придерживаться «от выходов» - always @ (posedge clk) reg <= newreg; вместо описания процессора «как программы». Удобнее начинать с поведения счетчика команд и загрузки литералов, они сильнее влияют на свойства процессора

Моделирование на системном уровне – удобный способ организовать анализ поведения схемы при выполнении программы. Наблюдать за процессором, выполняющим программу, существенно информативнее, чем тестировать отдельные узлы процессора, не имея возможности увидеть конечный результат их взаимодействия в процессе работы. Такое моделирование организуется с помощью нескольких несложных шагов.

1. Память заполняется программой

2. Подается тактовый сигнал

3. Подается сброс

4. Производится наблюдение смены состояния регистров и выходов процессора.

Контрольные вопросы:

1. Какие основные компоненты содержит процессор?

2. Как описать регистровый файл из N регистров с двумя выходами?

3. Какие действия выполняет процессорное ядро на каждом из тактов в 3-ступенчатом конвейере?

4. Что такое литерал и как он реализуется в процессоре?

5. Как организовать моделирование работы процессора?