CS:APP2e Web Aside ARCH:VLOG
Verilog Implementation of a Pipelined Y86 Processor*

Randal E. Bryant
David R. O'Hallaron

September 17, 2009

Notice

The material in this document is supplementary material to the book Computer Systems, A Programmer's Persepctive, Second Edition, *by Randal E. Bryant and David R. O'Hallaron, published by Prentice-Hall and copyrighted 2010. In this document, all references beginning with "CS:APP2e " are to this book. More information about the book is available at* `www.csapp.cs.cmu.edu`.

1 Introduction

Modern logic design involves writing a textual representation of a hardware design in a *hardware description language*. The design can then be tested by both simulation and by a variety of formal verification tools. Once we have confidence in the design, we can use *logic synthesis* tools to translate the design into actual logic circuits.

In this document, we describe an implementation of the PIPE processor in the Verilog hardware description language. This design combines modules implementing the basic building blocks of the processor, with control logic generated directly from the HCL description developed in CS:APP2e Chapter 4 and presented in Web Aside ARCH:HCL. We have been able to synthesize this design, download the logic circuit description onto field-programmable gate array (FPGA) hardware, and have the processor execute Y86 programs.

Aside: A Brief History of Verilog
Many different hardware description languages (HDLs) have been developed over the years, but Verilog was the first to achieve widespread success. It was developed originally by Philip Moorby, working at a company started in 1983 by Prabhu Goel to produce software that would assist hardware designers in designing and testing digital hardware. They gave their company what seemed at the time like a clever name: Automated Integrated Design Systems,

or “AIDS.” When that acronym became better known to stand for Acquired Immune Deficiency Syndrome, they renamed their company Gateway Design Automation in 1985. Gateway was acquired by Cadence Design Systems in 1990, which remains one of the major companies in Electronic Design Automation (EDA). Cadence transferred the Verilog language into the public domain, and it became IEEE Standard 1364-1995. Since then it has undergone several revisions, as well.

Verilog was originally conceived as a language for writing simulation models for hardware. The task of designing actual hardware was still done by more manual means of drawing logic schematics, with some assistance provided by software for drawing circuits on a computer.

Starting in the 1980s, researchers developed efficient means of automatically synthesizing logic circuits from more abstract descriptions. Given the popularity of Verilog for writing simulation models, it was natural to use this language as the basis for synthesis tools. The first, and still most widely used such tool is the Design Compiler, marked by Synopsys, Inc., another major EDA company. **End Aside.**

Since Verilog was originally designed to create simulation models, it has many features that cannot be synthesized into hardware. For example, it is possible to describe the detailed timing of different events, whereas this would depend greatly on the hardware technology for which the design is synthesized. As a result, there is a recognized *synthesizable subset* of the Verilog language, and hardware designers must restrict how they write Verilog descriptions to ensure they can be synthesized. Our Verilog stays well within the bounds of the synthesizable subset.

This document is not intended to be a complete description of Verilog, but just to convey enough about it to see how we can readily translate our Y86 processor designs into actual hardware. A comprehensive description of Verilog is provided by Thomas and Moorby’s book [1]

A complete Verilog implementation of PIPE suitable for logic synthesis is given in Appendix A of this document. We will go through some parts of this description, using the fetch stage of the PIPE processor as our main source of examples. For reference, a diagram of this stage is shown in Figure 1.

2 Combinational Logic

The basic data type for Verilog is the *bit vector*, a collection of bits having a range of indices. The standard notation for bit vectors is to specify the indices as a range of the form `[`*hi*`:`*lo*`]`, where integers *hi* and *lo* give the index values of the most and least significant bits, respectively. Here are some examples of signal declarations:

```
wire [31:0] aluA;
wire  [3:0] alufun;
wire        stall;
```

These declarations specify that the signals are of type `wire`, indicating that they serve as connections in a combinational circuit, rather than storing any information. We see that signals aluA and alufun are vectors of 32 and 4 bits, respectively, and that stall is a single bit (indicated when no index range is given.)

The operations on Verilog bit vectors are similar to those on C integers: arithmetic and bit-wise operations, shifting, and testing for equality or ordering relationships. In addition, it is possible to create new bit vectors by extracting ranges of bits from other vectors. For example, the expression `aluA[31:24]` creates an 8-bit wide vector equal to the most significant byte of aluA.

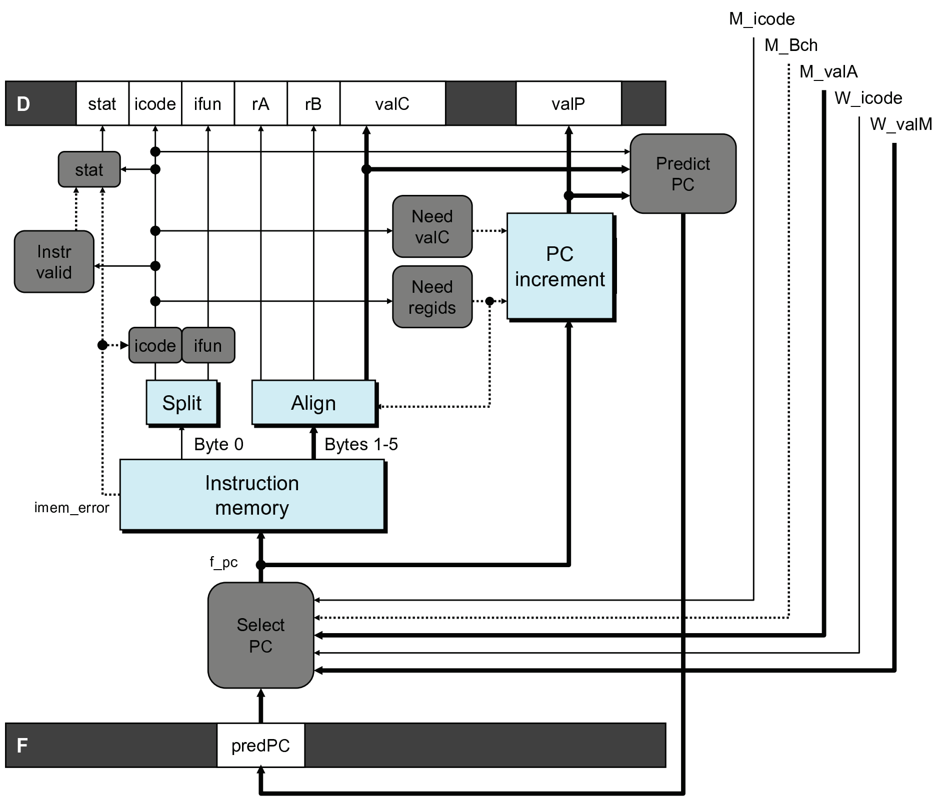

Figure 1: **PIPE PC selection and fetch logic.**

```
// Split instruction byte into icode and ifun fields
module split(ibyte, icode, ifun);
   input  [7:0] ibyte;
   output [3:0] icode;
   output [3:0] ifun;

   assign       icode = ibyte[7:4];
   assign       ifun  = ibyte[3:0];
endmodule

// Extract immediate word from 5 bytes of instruction
module align(ibytes, need_regids, rA, rB, valC);
   input [39:0]  ibytes;
   input         need_regids;
   output [3:0]  rA;
   output [3:0]  rB;
   output [31:0] valC;
   assign        rA = ibytes[7:4];
   assign        rB = ibytes[3:0];
   assign        valC = need_regids ? ibytes[39:8] : ibytes[31:0];
endmodule

// PC incrementer
module pc_increment(pc, need_regids, need_valC, valP);
   input [31:0]  pc;
   input         need_regids;
   input         need_valC;
   output [31:0] valP;
   assign        valP = pc + 1 + 4*need_valC + need_regids;
endmodule
```

Figure 2: **Hardware Units for Fetch Stage.** These illustrate the use of modules and bit vector operations in Verilog.

Verilog allows a system to be described as a hierarchy of *modules*. These modules are similar to procedures, except that they do not define an action to be performed when invoked, but rather they describe a portion of a system that can be *instantiated* as a block of hardware. Each module declares a set of interface signals—the inputs and outputs of the block—and a set of interconnected hardware components, consisting of either other module instantiations or primitive logic operations.

As an example of Verilog modules implementing simple combinational logic, Figure 2 shows Verilog descriptions of the hardware units required by the fetch stage of PIPE. For example, the module `split` serves to split the first byte of an instruction into the instruction code and function fields. We see that this module has a single eight-bit input ibyte and two four-bit outputs icode and ifun. Output icode is defined to be the high-order four bits of ibyte, while ifun is defined to be the low-order four bits. Verilog has several different forms of *assignment* operators. An assignment starting with the keyword `assign` is known as a *continuous assignment*. It can be thought of as a way to connect two signals via simple wires, as when constructing

```
module alu(aluA, aluB, alufun, valE, new_cc);
   input [31:0]  aluA, aluB;    // Data inputs
   input [3:0]   alufun;        // ALU function
   output [31:0] valE;          // Data Output
   output [2:0]  new_cc;        // New values for ZF, SF, OF

   parameter     ALUADD = 4'h0;
   parameter     ALUSUB = 4'h1;
   parameter     ALUAND = 4'h2;
   parameter     ALUXOR = 4'h3;

   assign        valE =
                 alufun == ALUSUB ? aluB - aluA :
                 alufun == ALUAND ? aluB & aluA :
                 alufun == ALUXOR ? aluB ^ aluA :
                 aluB + aluA;
   assign        new_cc[2] = (valE == 0);  // ZF
   assign        new_cc[1] = valE[31];     // SF
   assign        new_cc[0] =               // OF
                   alufun == ALUADD ?
                      (aluA[31] == aluB[31])  & (aluA[31] != valE[31]) :
                   alufun == ALUSUB ?
                      (~aluA[31] == aluB[31]) & (aluB[31] != valE[31]) :
                   0;
endmodule
```

Figure 3: **Verilog implementation of Y86 ALU.** This illustrates arithmetic and logical operations, as well as the Verilog notation for bit-vector constants.

combinational logic. Unlike an assignment in a programming language such as C, continuous assignment does not specify a single updating of a value, but rather it creates a permanent connection from the output of one block of logic to the input of another. So, for example, the description in the `split` module states that the two outputs are directly connected to the relevant fields of the input.

The `align` module describes how the processor extracts the remaining fields from an instruction, depending on whether or not the instruction has a register specifier byte. Again we see the use of continuous assignments and bit vector subranges. This module also includes a *conditional expression*, similar to the conditional expressions of C. In Verilog, however, this expression provides a way of creating a multiplexor—combinational logic that chooses between two data inputs based on a one-bit control signal.

The `pc_increment` module demonstrates some arithmetic operations in Verilog. These are similar to the arithmetic operations of C. Originally, Verilog only supported unsigned arithmetic on bit vectors. Two's complement arithmetic was introduced in the 2001 revision of the language. All operations in our description involve unsigned arithmetic.

As another example of combinational logic, Figure 3 shows an implementation of an ALU for the Y86 execute stage. We see that it has as inputs two 32-bit data words and a 4-bit function code. For outputs, it has a 32-bit data word and the three bits used to create condition codes. The `parameter` statement

```
// Clocked register with enable signal and synchronous reset
// Default width is 8, but can be overriden
module cenrreg(out, in, enable, reset, resetval, clock);
   parameter width = 8;
   output [width-1:0] out;
   reg [width-1:0]    out;
   input [width-1:0]  in;
   input              enable;
   input              reset;
   input [width-1:0]  resetval;
   input              clock;

   always
     @(posedge clock)
     begin
        if (reset)
          out <= resetval;
        else if (enable)
          out <= in;
     end
endmodule
```

Figure 4: **Basic Clocked Register.**

provides a way to give names to constant values, much as the way constants can be defined in C using `#define`. In Verilog, a bit-vector constant has a specific width, and a value given in either decimal (the default), hexadecimal (specified with 'h'), or binary (specified with 'b') form. For example, the notation `4'h2` indicates a 4-bit wide vector having hexadecimal value 2. The rest of the module describes the functionality of the ALU. We see that the output data will either equal an expression over the two data inputs (sum, difference, bitwise EXCLUSIVE-OR, or bitwise AND), or it will equal 0. The output conditions are computed using the values of the input and output data words, based on the properties of a two's complement representation of the data (CS:APP2e Section 2.3.2.)

3 Registers

Thus far, we have considered only combinational logic, expressed using continuous assignments. Verilog has many different ways to express sequential behavior, event sequencing, and time-based waveforms. We will restrict our presentation to ways to express the simple clocking methods required by the Y86 processor.

Figure 4 shows a clocked register `cenrreg` (short for "conditionally-enabled, resettable register") that we will use as a building block for the hardware registers in our processor. The idea is to have a register that can be loaded with the value on its input in response to a clock. Additionally, it is possible to *reset* the register, causing it to be set to a fixed constant value.

Some features of this module are worth highlighting. First, we see that the module is *parameterized* by a value width, indicating the number of bits comprising the input and output words. By default, the module has a width of 8 bits, but this can be overridden by instantiating the module with a different width.

We see that the register data output `out` is declared to be of type `reg` (short for "register"). That means that it will hold its value until it is explicitly updated. This contrasts to the signals of type `wire` that are used to implement combinational logic.

The statement beginning `always @(posedge clock)` describes a set of actions that will be triggered every time the clock signal goes for 0 to 1 (this is considered to be the positive edge of a clock signal.) Within this statement, we see that the output may be updated to be either its input or its reset value. The assignment operator `<=` is known as a *non-blocking* assignment. That means that the actual updating of the output will only take place when a new event is triggered, in this case the transition of the clock from 0 to 1. We can see that the output may be updated as the clock rises. Observe, however, that if neither the reset nor the enable signals are 1, then the output will remain at its current value.

The following module `preg` shows how we can use our basic register to construct a pipeline register:

```
// Pipeline register.  Uses reset signal to inject bubble
// When bubbling, must specify value that will be loaded
module preg(out, in, stall, bubble, bubbleval, clock);
   parameter width = 8;
   output [width-1:0] out;
   input [width-1:0]  in;
   input              stall, bubble;
   input [width-1:0]  bubbleval;
   input              clock;

   cenrreg #(width) r(out, in, ~stall, bubble, bubbleval, clock);
endmodule
```

We see that a pipeline register is created by instantiating a clocked register, but making the enable signal be the complement of the stall signal. We see here also the way modules are instantiated in Verilog. A module instantiatioin gives the name of the module, an optional list of parametric values, (in this case, we want the width of the register to be the width specified by the module's parameter), an instance name (used when debugging a design by simulation), and a list of module parameters.

The register file is implemented using eight clocked registers for the eight program registers. Combinational logic is used to select which program register values are routed to the register file outputs, and which program registers to update by a write operation. The Verilog code for this is found in Appendix A, lines 135–221.

4 Memory

The memory module, illustrated in Figure 5, implements both the instruction and the data memory. The Verilog code for the module can be found in Appendix A, lines 223–501.

The module interface is defined as follows:

```
module bmemory(maddr, wenable, wdata, renable, rdata, m_ok,
```

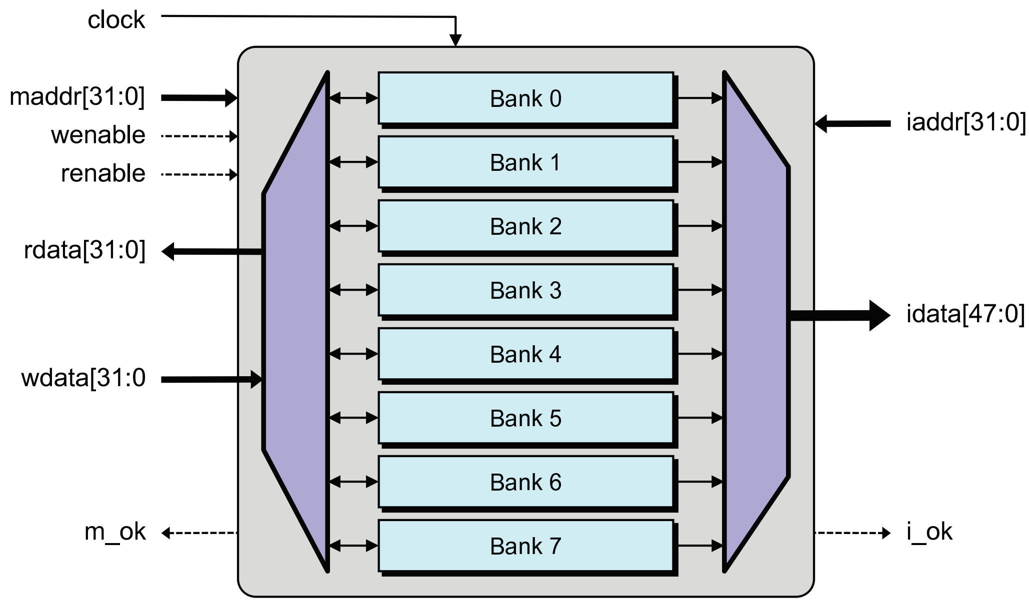

Figure 5: **Memory structure.** The memory consists of eight banks, each performing single-byte reads and writes.

```
              iaddr, instr, i_ok, clock);
parameter memsize = 4096; // Number of bytes in memory
input [31:0]  maddr;    // Read/Write address
input         wenable;  // Write enable
input [31:0]  wdata;    // Write data
input         renable;  // Read enable
output [31:0] rdata;    // Read data
output        m_ok;     // Read & write addresses within range
input [31:0]  iaddr;    // Instruction address
output [47:0] instr;    // 6 bytes of instruction
output        i_ok;     // Instruction address within range
input         clock;
```

In Figure 5, we adopt the Verilog convention of indicating the index ranges for each of the multi-bit signals. The left-hand side of the figure shows the port used for reading and writing data. We see that it has an address input maddr, data output rdata and input wdata, and enable signals for reading and writing. The output signal m_ok indicates whether or not the address input is within the range of valid addresses for the memory.

The right-hand side of the figure shows the port used for fetching instructions. It has just an address input iaddr, a 48-byte wide data output idata, and a signal i_ok indicating whether or not the address is within the range of valid addresses.

We require a method for accessing groups of four or six successive bytes in the memory, and we cannot assume any particular alignment for the addresses. We therefore implement the memory with a set of eight *banks*, each of which is a random-access memory that can be used to store, read, and write individual bytes.

A byte with memory address i is stored in bank $i \bmod 8$, and the address of the byte within the bank is $\lfloor i/8 \rfloor$. Some advantages of this organization are:

- Any six successive bytes will be stored in separate banks. Thus, the processor can read all six instruction bytes using single-byte bank reads. Similarly, the processor can read or write all four data bytes using single-byte bank reads or writes.
- The bank number is given by the low-order three bits of the memory address.
- The address of a byte within the bank is given by the remaining bits of the memory address.

Figure 6 gives a Verilog description of a "combinational" RAM module suitable for implementing the memory banks. This RAM stores data in units of "words," where we will set the word size to be eight bits. We see that the module has three associated parametric values:

wordsize: The number of bits in each "word" of the memory. The default value is eight.

wordcount: The number of words stored in the memory. The default value of 512 creates a memory capable of storing $8 \cdot 512 = 4096$ bytes.

addrsize: The number of bits in the address input. If the memory contains n words, this parameter must be at least $\log_2 n$.

This module implements the model we have assumed in Chapter 4: memory writes occur when the clock goes high, but memory reads operate as if the memory were a block of combinational logic.

Several features of the combinational RAM module are worth noting. We see the declaration of the actual memory array on line 28. It declares `mem` to be an array with elements numbered from 0 to the word count minus 1, where each array element is bit vector with bits numbered from 0 to the word size minus 1. Furthermore, each bit is of type `reg`, and therefore acts as a storage element.

The combinational RAM has two ports, labeled "A" and "B," that can be independently written on each cycle. We see the writes occurring within `always` blocks, and each involving a nonblocking assignment (lines 34 and 44.) The memory array is addressed using an array notation. We see also the two reads are expressed as continuous assignments (lines 38 and 48), meaning that these outputs will track the values of whatever memory elements are being addressed.

The combinational RAM is fine for running simulations of the processor using a Verilog simulator. In real life, however, most random-access memories require a clock to trigger a sequence of events that carries out a read operation (see CS:APP2e Section 6.1.1), and so we must modify our design slightly to work with a *synchronous* RAM, meaning that both read and write operations occur in response to a clock signal. Fortunately, a simple timing trick allows us to use a synchronous RAM module in the PIPE processor.

We design the RAM blocks used to implement the memory banks, such that the read and write operations are triggered by the *falling* edge of the clock, as it makes the transition for 1 to 0. This yields a timing illustrated in Figure 7. We see that the regular registers (including the pipeline registers, the condition code register, and the register file) are updated when the clock goes from 0 to 1. At this point, values propagate through combinational logic to the address, data, and control inputs of the memory. The clock transition from 1 to

```
// This module implements a dual-ported RAM.
// with clocked write and combinational read operations.
// This version matches the conceptual model presented in the CS:APP book,

module ram(clock, addrA, wEnA, wDatA, rEnA, rDatA,
           addrB, wEnB, wDatB, rEnB, rDatB);

   parameter wordsize = 8;    // Number of bits per word
   parameter wordcount = 512; // Number of words in memory
   // Number of address bits.  Must be >= log wordcount
   parameter addrsize = 9;

   input     clock;              // Clock
   // Port A
   input [addrsize-1:0]  addrA;  // Read/write address
   input                 wEnA;   // Write enable
   input [wordsize-1:0]  wDatA;  // Write data
   input                 rEnA;   // Read enable
   output [wordsize-1:0] rDatA;  // Read data
   // Port B
   input [addrsize-1:0]  addrB;  // Read/write address
   input                 wEnB;   // Write enable
   input [wordsize-1:0]  wDatB;  // Write data
   input                 rEnB;   // Read enable
   output [wordsize-1:0] rDatB;  // Read data

   // Actual storage
   reg [wordsize-1:0]    mem[wordcount-1:0];

   always @(posedge clock)
     begin
        if (wEnA)
          begin
             mem[addrA] <= wDatA;
          end
     end
   // Combinational reads
   assign rDatA = mem[addrA];

   always @(posedge clock)
     begin
        if (wEnB)
          begin
             mem[addrB] <= wDatB;
          end
     end
   // Combinational reads
   assign rDatB = mem[addrB];

endmodule
```

Figure 6: **Combinational RAM Module.** This module implements the memory banks, following the read/write model we have assumed for Y86.

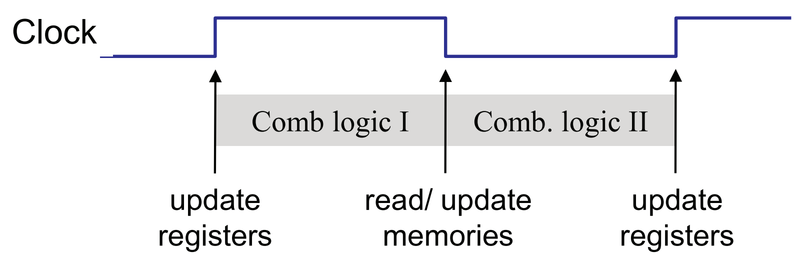

Figure 7: **Timing of synchronous RAM.** By having the memory be read and written on the falling clock edge, the combinational logic can be active both before (A) and after (B) the memory operation.

0 causes the designated memory operations to take place. More combinational logic is then activated to propagate values to the register inputs, arriving there in time for the next clock transition.

With this timing, we can therefore classify each combinational logic block as being either in group I, meaning that it depends only on the values stored in registers, and group II, meaning that it depends on the values read from memory.

> **Practice Problem 1**:
>
> Determine which combination logic blocks in the fetch stage (Figure 1) are in group I, and which are in group II.

Figure 8 shows a synchronous RAM module that better reflects the random-access memories available to hardware designers. Comparing this module to the combinational RAM (Figure 6), we see two differences. First the data outputs rDatA and rDatB are both declared to be of type `reg`, meaning that they will hold the value assigned to them until they are explicitly updated (lines 20 and 27.) Second, the updating of these two outputs occur via nonblocking assignments within `always` blocks (lines 38 and 48).

The remaining portions of the memory module are implemented as combinational logic, and so changing the underlying bank memory design is the only modification required to shift the memory from having combinational read operations to having synchronous ones. This is the only modification required to our processor design to make it synthesizable as actual hardware.

5 Overall Processor Design

We have now created the basic building blocks for a Y86 processor. We are ready to assemble these pieces into an actual processor. Figure 9 shows the input and output connections we will design for our processor, allowing the processor to be operated by an external controller. The Verilog declaration for the processor module is shown in Figure 10. The mode input specifies what the processor should be doing. The possible values are

RUN: Execute instructions in the normal manner.

```
// This module implements a dual-ported RAM.
// with clocked write and read operations.

module ram(clock, addrA, wEnA, wDatA, rEnA, rDatA,
           addrB, wEnB, wDatB, rEnB, rDatB);

parameter wordsize = 8;    // Number of bits per word
parameter wordcount = 512; // Number of words in memory
// Number of address bits.  Must be >= log wordcount
parameter addrsize = 9;


  input  clock;                 // Clock
  // Port A
  input [addrsize-1:0] addrA;   // Read/write address
  input  wEnA;                  // Write enable
  input [wordsize-1:0] wDatA;   // Write data
  input  rEnA;                  // Read enable
  output [wordsize-1:0] rDatA;  // Read data
  reg [wordsize-1:0] rDatA;
  // Port B
  input [addrsize-1:0] addrB;   // Read/write address
  input  wEnB;                  // Write enable
  input [wordsize-1:0] wDatB;   // Write data
  input  rEnB;                  // Read enable
  output [wordsize-1:0] rDatB;  // Read data
  reg [wordsize-1:0] rDatB;

  reg[wordsize-1:0] mem[wordcount-1:0];  // Actual storage

   always @(negedge clock)
   begin
     if (wEnA)
     begin
       mem[addrA] <= wDatA;
     end
     if (rEnA)
       rDatA <= mem[addrA];
   end

   always @(negedge clock)
   begin
     if (wEnB)
     begin
       mem[addrB] <= wDatB;
     end
     if (rEnB)
       rDatB <= mem[addrB];
   end
endmodule
```

Figure 8: **Synchronous RAM Module.** This module implements the memory banks using synchronous read operations.

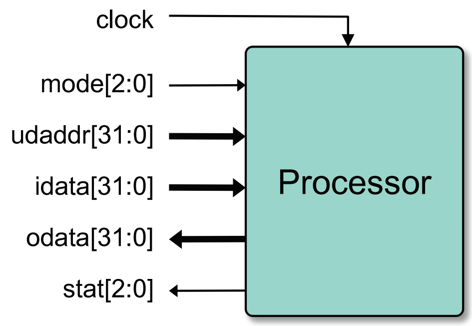

Figure 9: **Processor interface.** Mechanisms are included to upload and download memory data and processor state, and to operate the processor in different modes.

```
module processor(mode, udaddr, idata, odata, stat, clock);
  input  [2:0]  mode;   // Signal operating mode to processor
  input  [31:0] udaddr; // Upload/download address
  input  [31:0] idata;  // Download data word
  output [31:0] odata;  // Upload data word
  output [2:0]  stat;   // Status
  input         clock;  // Clock input
```

Figure 10: **Declaration of processor module.**

RESET: All registers are set to their initial values, clearing the pipeline registers and setting the program counter to 0.

DOWNLOAD: The processor memory can be loaded using the udaddr address input and the idata data input to specify addresses and values. By this means, we can load a program into the processor.

UPLOAD: Data can be extracted from the processor memory, using the address input udaddr to specify an address and the odata output to provide the data stored at that address.

STATUS: Similar to UPLOAD mode, except that the values of the program registers, and the condition codes can be extracted. Each program register and the condition codes have associated addresses for this operation.

The stat output is a copy of the Stat signal generated by the processor.

A typical operation of the processor involves the following sequence: 1) first, a program is downloaded into memory, downloading four bytes per cycle in DOWNLOAD mode. The processor is then put into RESET mode for one clock cycle. The processor is operated in RUN mode until the stat output indicates that some type of exception has occurred (normally when the processor executes a `halt` instruction.) The results are then read from the processor over multiple cycles in the UPLOAD and STATUS modes.

6 Implementation Highlights

The following are samples of the Verilog code for our implementation of PIPE, showing the implementation of the fetch stage.

The following are declarations of the internal signals of the fetch stage. They are all of type `wire`, meaning that they are simply connectors from one logic block to another.

```
  wire [31:0] f_predPC, F_predPC, f_pc;
  wire        f_ok;
  wire        imem_error;
  wire [2:0]  f_stat;
  wire [47:0] f_instr;
  wire [3:0]  imem_icode;
  wire [3:0]  imem_ifun;
  wire [3:0]  f_icode;
  wire [3:0]  f_ifun;
  wire [3:0]  f_rA;
  wire [3:0]  f_rB;
  wire [31:0] f_valC;
  wire [31:0] f_valP;
  wire        need_regids;
  wire        need_valC;
  wire        instr_valid;
  wire        F_stall, F_bubble;
```

The following signals must be included to allow pipeline registers F and D to be reset when either the processor is in RESET mode or the bubble signal is set for the pipeline register.

```
  wire resetting = (mode == RESET_MODE);
  wire F_reset = F_bubble | resetting;
  wire D_reset = D_bubble | resetting;
```

The different elements of pipeline registers F and D are generated as instantiations of the `preg` register module. Observe how these are instantiated with different widths, according to the number of bits in each element:

```
// All pipeline registers are implemented with module
//    preg(out, in, stall, bubble, bubbleval, clock)
// F Register
  preg #(32) F_predPC_reg(F_predPC, f_predPC, F_stall, F_reset, 0, clock);
// D Register
  preg #(3)  D_stat_reg(D_stat, f_stat, D_stall, D_reset, SBUB, clock);
  preg #(32) D_pc_reg(D_pc, f_pc, D_stall, D_reset, 0, clock);
  preg #(4)  D_icode_reg(D_icode, f_icode, D_stall, D_reset, INOP, clock);
  preg #(4)  D_ifun_reg(D_ifun, f_ifun, D_stall, D_reset, FNONE, clock);
  preg #(4)  D_rA_reg(D_rA, f_rA, D_stall, D_reset, RNONE, clock);
```

```
  preg #(4)  D_rB_reg(D_rB, f_rB, D_stall, D_reset, RNONE, clock);
  preg #(32) D_valC_reg(D_valC, f_valC, D_stall, D_reset, 0, clock);
  preg #(32) D_valP_reg(D_valP, f_valP, D_stall, D_reset, 0, clock);
```

We want to generate the Verilog descriptions of the control logic blocks directly from their HCL descriptions. For example, the following are HCL representations of blocks found in the fetch stage:

```
## What address should instruction be fetched at
int f_pc = [
        # Mispredicted branch.  Fetch at incremented PC
        M_icode == IJXX && !M_Cnd : M_valA;
        # Completion of RET instruction.
        W_icode == IRET : W_valM;
        # Default: Use predicted value of PC
        1 : F_predPC;
];

## Determine icode of fetched instruction
int f_icode = [
        imem_error : INOP;
        1: imem_icode;
];

# Determine ifun
int f_ifun = [
        imem_error : FNONE;
        1: imem_ifun;
];

# Is instruction valid?
bool instr_valid = f_icode in
        { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
          IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };

# Determine status code for fetched instruction
int f_stat = [
        imem_error: SADR;
        !instr_valid : SINS;
        f_icode == IHALT : SHLT;
        1 : SAOK;
];

# Does fetched instruction require a regid byte?
bool need_regids =
        f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                     IIRMOVL, IRMMOVL, IMRMOVL };

# Does fetched instruction require a constant word?
bool need_valC =
```

```
        f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };

# Predict next value of PC
int f_predPC = [
        f_icode in { IJXX, ICALL } : f_valC;
        1 : f_valP;
];
```

We have implemented a program HCL2V (short for "HCL to Verilog") to generate Verilog code from HCL expressions. The following are examples of code generated from the HCL descriptions of blocks found in the fetch stage. These are not formatted in a way that makes them easily readable, but it can be seen that the conversion from HCL to Verilog is fairly straightforward:

```
assign f_pc =
    (((M_icode == IJXX) & ~M_Cnd) ? M_valA : (W_icode == IRET) ? W_valM :
      F_predPC);

assign f_icode =
    (imem_error ? INOP : imem_icode);

assign f_ifun =
    (imem_error ? FNONE : imem_ifun);

assign instr_valid =
    (f_icode == INOP | f_icode == IHALT | f_icode == IRRMOVL | f_icode ==
      IIRMOVL | f_icode == IRMMOVL | f_icode == IMRMOVL | f_icode == IOPL
       | f_icode == IJXX | f_icode == ICALL | f_icode == IRET | f_icode ==
      IPUSHL | f_icode == IPOPL);

assign f_stat =
    (imem_error ? SADR : ~instr_valid ? SINS : (f_icode == IHALT) ? SHLT :
      SAOK);

assign need_regids =
    (f_icode == IRRMOVL | f_icode == IOPL | f_icode == IPUSHL | f_icode ==
      IPOPL | f_icode == IIRMOVL | f_icode == IRMMOVL | f_icode == IMRMOVL)
    ;

assign need_valC =
    (f_icode == IIRMOVL | f_icode == IRMMOVL | f_icode == IMRMOVL | f_icode
       == IJXX | f_icode == ICALL);

assign f_predPC =
    ((f_icode == IJXX | f_icode == ICALL) ? f_valC : f_valP);
```

Finally, we must instantiate the different modules implementing the hardware units we examined earlier:

```
  split split(f_instr[7:0], imem_icode, imem_ifun);
```

```
align align(f_instr[47:8], need_regids, f_rA, f_rB, f_valC);
pc_increment pci(f_pc, need_regids, need_valC, f_valP);
```

7 Summary

We have successfully generated a synthesizable Verilog description of a pipelined Y86 processor. We see from this exercise that the processor design we created in CS:APP2e Chapter 4 is sufficiently complete that it leads directly to a hardware realization. We have successfully run this Verilog through synthesis tools and mapped the design onto FPGA-based hardware.

Problem Solutions

Problem 1 Solution: [Pg. 11]

We see that only the PC selection block is in group I. All others depend, in part, on the value read from the instruction memory and therefore are in group II.

Acknowledgments

James Hoe of Carnegie Mellon University has been instrumental in the design of the Y86 processor, in helping us learn Verilog, and in using synthesis tools to generate a working microprocessor.

A Complete Verilog for PIPE

The following is a complete Verilog description of our implementation of PIPE. It was generated by combining a number of different module descriptions, and incorporating logic generated automatically from the HCL description. This model uses the synchronous RAM module suitable for both simulation and synthesis.

```
// --------------------------------------------------------------------
// Verilog representation of PIPE processor
// --------------------------------------------------------------------

// --------------------------------------------------------------------
// Memory module for implementing bank memories
// --------------------------------------------------------------------
// This module implements a dual-ported RAM.
// with clocked write and read operations.

module ram(clock, addrA, wEnA, wDatA, rEnA, rDatA,
           addrB, wEnB, wDatB, rEnB, rDatB);

```

```
parameter wordsize = 8;    // Number of bits per word
parameter wordcount = 512; // Number of words in memory
// Number of address bits.  Must be >= log wordcount
parameter addrsize = 9;


  input  clock;                  // Clock
  // Port A
  input [addrsize-1:0] addrA;   // Read/write address
  input  wEnA;                   // Write enable
  input [wordsize-1:0] wDatA;   // Write data
  input  rEnA;                   // Read enable
  output [wordsize-1:0] rDatA;  // Read data
  reg [wordsize-1:0] rDatA; //= line:arch:synchram:rDatA
  // Port B
  input [addrsize-1:0] addrB;   // Read/write address
  input  wEnB;                   // Write enable
  input [wordsize-1:0] wDatB;   // Write data
  input  rEnB;                   // Read enable
  output [wordsize-1:0] rDatB;  // Read data
  reg [wordsize-1:0] rDatB; //= line:arch:synchram:rDatB

  reg[wordsize-1:0] mem[wordcount-1:0];  // Actual storage

   // To make the pipeline processor work with synchronous reads, we
   // operate the memory read operations on the negative
   // edge of the clock.  That makes the reading occur in the middle
   // of the clock cycle---after the address inputs have been set
   // and such that the results read from the memory can flow through
   // more combinational logic before reaching the clocked registers

   // For uniformity, we also make the memory write operation
   // occur on the negative edge of the clock.  That works OK
   // in this design, because the write can occur as soon as the
   // address & data inputs have been set.
   always @(negedge clock)
   begin
     if (wEnA)
     begin
       mem[addrA] <= wDatA;
     end
     if (rEnA)
       rDatA <= mem[addrA]; //= line:arch:synchram:readA
   end

   always @(negedge clock)
   begin
     if (wEnB)
     begin
       mem[addrB] <= wDatB;
```

```
    end
    if (rEnB)
      rDatB <= mem[addrB]; //= line:arch:synchram:readB
  end
endmodule

// ----------------------------------------------------------------------
// Other building blocks
// ----------------------------------------------------------------------

// Basic building blocks for constructing a Y86 processor.

// Different types of registers, all derivatives of module cenrreg

// Clocked register with enable signal and synchronous reset
// Default width is 8, but can be overriden
module cenrreg(out, in, enable, reset, resetval, clock);
   parameter width = 8;
   output [width-1:0] out;
   reg [width-1:0]    out;
   input [width-1:0]  in;
   input              enable;
   input              reset;
   input [width-1:0]  resetval;
   input              clock;

   always
     @(posedge clock)
     begin
        if (reset)
          out <= resetval;
        else if (enable)
          out <= in;
     end
endmodule

// Clocked register with enable signal.
// Default width is 8, but can be overriden
module cenreg(out, in, enable, clock);
   parameter width = 8;
   output [width-1:0] out;
   input [width-1:0]  in;
   input              enable;
   input              clock;

   cenrreg #(width) c(out, in, enable, 0, 0, clock);
endmodule

// Basic clocked register.  Default width is 8.
module creg(out, in, clock);
```

```
   parameter width = 8;
   output [width-1:0] out;
   input [width-1:0]  in;
   input              clock;

   cenreg #(width) r(out, in, 1, clock);
endmodule

// Pipeline register.  Uses reset signal to inject bubble
// When bubbling, must specify value that will be loaded
module preg(out, in, stall, bubble, bubbleval, clock);
   parameter width = 8;
   output [width-1:0] out;
   input [width-1:0]  in;
   input              stall, bubble;
   input [width-1:0]  bubbleval;
   input              clock;

   cenrreg #(width) r(out, in, ~stall, bubble, bubbleval, clock);
endmodule

// Register file
module regfile(dstE, valE, dstM, valM, srcA, valA, srcB, valB,
         reset, clock, eax, ecx, edx, ebx, esp, ebp, esi, edi);
   input [3:0] dstE;
   input [31:0] valE;
   input [3:0]  dstM;
   input [31:0] valM;
   input [3:0]  srcA;
   output [31:0] valA;
   input [3:0]   srcB;
   output [31:0] valB;
   input         reset;  // Set registers to 0
   input         clock;
   // Make individual registers visible for debugging
   output [31:0] eax, ecx, edx, ebx, esp, ebp, esi, edi;

   // Define names for registers used in HCL code
   parameter     REAX =  4'h0;
   parameter     RECX =  4'h1;
   parameter     REDX =  4'h2;
   parameter     REBX =  4'h3;
   parameter     RESP =  4'h4;
   parameter     REBP =  4'h5;
   parameter     RESI =  4'h6;
   parameter     REDI =  4'h7;
   parameter     RNONE =  4'hf;

   // Input data for each register
   wire [31:0]   eax_dat, ecx_dat, edx_dat, ebx_dat,
```

```
                 esp_dat, ebp_dat, esi_dat, edi_dat;

  // Input write controls for each register
  wire           eax_wrt, ecx_wrt, edx_wrt, ebx_wrt,
                 esp_wrt, ebp_wrt, esi_wrt, edi_wrt;

  // Implement with clocked registers
  cenrreg #(32) eax_reg(eax, eax_dat, eax_wrt, reset, 0, clock);
  cenrreg #(32) ecx_reg(ecx, ecx_dat, ecx_wrt, reset, 0, clock);
  cenrreg #(32) edx_reg(edx, edx_dat, edx_wrt, reset, 0, clock);
  cenrreg #(32) ebx_reg(ebx, ebx_dat, ebx_wrt, reset, 0, clock);
  cenrreg #(32) esp_reg(esp, esp_dat, esp_wrt, reset, 0, clock);
  cenrreg #(32) ebp_reg(ebp, ebp_dat, ebp_wrt, reset, 0, clock);
  cenrreg #(32) esi_reg(esi, esi_dat, esi_wrt, reset, 0, clock);
  cenrreg #(32) edi_reg(edi, edi_dat, edi_wrt, reset, 0, clock);

  // Reads occur like combinational logic
  assign         valA =
                 srcA == REAX ? eax :
                 srcA == RECX ? ecx :
                 srcA == REDX ? edx :
                 srcA == REBX ? ebx :
                 srcA == RESP ? esp :
                 srcA == REBP ? ebp :
                 srcA == RESI ? esi :
                 srcA == REDI ? edi :
                 0;

  assign         valB =
                 srcB == REAX ? eax :
                 srcB == RECX ? ecx :
                 srcB == REDX ? edx :
                 srcB == REBX ? ebx :
                 srcB == RESP ? esp :
                 srcB == REBP ? ebp :
                 srcB == RESI ? esi :
                 srcB == REDI ? edi :
                 0;

  assign         eax_dat = dstM == REAX ? valM : valE;
  assign         ecx_dat = dstM == RECX ? valM : valE;
  assign         edx_dat = dstM == REDX ? valM : valE;
  assign         ebx_dat = dstM == REBX ? valM : valE;
  assign         esp_dat = dstM == RESP ? valM : valE;
  assign         ebp_dat = dstM == REBP ? valM : valE;
  assign         esi_dat = dstM == RESI ? valM : valE;
  assign         edi_dat = dstM == REDI ? valM : valE;

  assign         eax_wrt = dstM == REAX | dstE == REAX;
  assign         ecx_wrt = dstM == RECX | dstE == RECX;
```

```
   assign        edx_wrt = dstM == REDX | dstE == REDX;
   assign        ebx_wrt = dstM == REBX | dstE == REBX;
   assign        esp_wrt = dstM == RESP | dstE == RESP;
   assign        ebp_wrt = dstM == REBP | dstE == REBP;
   assign        esi_wrt = dstM == RESI | dstE == RESI;
   assign        edi_wrt = dstM == REDI | dstE == REDI;

endmodule

// Memory.  This memory design uses 8 memory banks, each
// of which is one byte wide.  Banking allows us to select an
// arbitrary set of 6 contiguous bytes for instruction reading
// and an arbitrary set of 4 contiguous bytes
// for data reading & writing.
// It uses an external RAM module from either the file
// combram.v (using combinational reads)
// or synchram.v (using clocked reads)
// The SEQ & SEQ+ processors only work with combram.v.
// PIPE works with either.

module bmemory(maddr, wenable, wdata, renable, rdata, m_ok,
               iaddr, instr, i_ok, clock);
   parameter memsize = 4096; // Number of bytes in memory
   input [31:0]  maddr;    // Read/Write address
   input         wenable;  // Write enable
   input [31:0]  wdata;    // Write data
   input         renable;  // Read enable
   output [31:0] rdata;    // Read data
   output        m_ok;     // Read & write addresses within range
   input [31:0]  iaddr;    // Instruction address
   output [47:0] instr;    // 6 bytes of instruction
   output        i_ok;     // Instruction address within range
   input         clock;

   wire [7:0]    ib0, ib1, ib2, ib3, ib4, ib5; // Instruction bytes
   wire [7:0]    db0, db1, db2, db3;           // Data bytes

   wire [2:0]    ibid = iaddr[2:0];     // Instruction Bank ID
   wire [28:0]   iindex = iaddr[31:3];  // Address within bank
   wire [28:0]   iip1 = iindex+1;       // Next address within bank

   wire [2:0]    mbid = maddr[2:0];     // Data Bank ID
   wire [28:0]   mindex = maddr[31:3];  // Address within bank
   wire [28:0]   mip1 = mindex+1;       // Next address within bank

   // Instruction addresses for each bank
   wire [28:0]   addrI0, addrI1, addrI2, addrI3, addrI4, addrI5, addrI6, addrI7;
   // Instruction data for each bank
   wire [7:0]    outI0, outI1, outI2, outI3, outI4, outI5, outI6, outI7;
```

```
// Data addresses for each bank
wire [28:0]   addrD0, addrD1, addrD2, addrD3, addrD4, addrD5, addrD6, addrD7;
// Data output for each bank
wire [7:0]    outD0, outD1, outD2, outD3, outD4, outD5, outD6, outD7;
// Data input for each bank
wire [7:0]    inD0, inD1, inD2, inD3, inD4, inD5, inD6, inD7;
// Data write enable signals for each bank
wire          dwEn0, dwEn1, dwEn2, dwEn3, dwEn4, dwEn5, dwEn6, dwEn7;

// The bank memories
ram #(8, memsize/8, 29) bank0(clock, addrI0, 0, 0, 1, outI0, // Instruction
                              addrD0, dwEn0, inD0, renable, outD0); // Data

ram #(8, memsize/8, 29) bank1(clock, addrI1, 0, 0, 1, outI1, // Instruction
                              addrD1, dwEn1, inD1, renable, outD1); // Data

ram #(8, memsize/8, 29) bank2(clock, addrI2, 0, 0, 1, outI2, // Instruction
                              addrD2, dwEn2, inD2, renable, outD2); // Data

ram #(8, memsize/8, 29) bank3(clock, addrI3, 0, 0, 1, outI3, // Instruction
                              addrD3, dwEn3, inD3, renable, outD3); // Data

ram #(8, memsize/8, 29) bank4(clock, addrI4, 0, 0, 1, outI4, // Instruction
                              addrD4, dwEn4, inD4, renable, outD4); // Data

ram #(8, memsize/8, 29) bank5(clock, addrI5, 0, 0, 1, outI5, // Instruction
                              addrD5, dwEn5, inD5, renable, outD5); // Data

ram #(8, memsize/8, 29) bank6(clock, addrI6, 0, 0, 1, outI6, // Instruction
                              addrD6, dwEn6, inD6, renable, outD6); // Data

ram #(8, memsize/8, 29) bank7(clock, addrI7, 0, 0, 1, outI7, // Instruction
                              addrD7, dwEn7, inD7, renable, outD7); // Data


// Determine the instruction addresses for the banks
assign        addrI0 = ibid >= 3 ? iip1 : iindex;
assign        addrI1 = ibid >= 4 ? iip1 : iindex;
assign        addrI2 = ibid >= 5 ? iip1 : iindex;
assign        addrI3 = ibid >= 6 ? iip1 : iindex;
assign        addrI4 = ibid >= 7 ? iip1 : iindex;
assign        addrI5 = iindex;
assign        addrI6 = iindex;
assign        addrI7 = iindex;

// Get the bytes of the instruction
assign        i_ok =
              (iaddr + 5) < memsize;

assign        ib0 = !i_ok ? 0 :
```

```
              ibid == 0 ? outI0 :
              ibid == 1 ? outI1 :
              ibid == 2 ? outI2 :
              ibid == 3 ? outI3 :
              ibid == 4 ? outI4 :
              ibid == 5 ? outI5 :
              ibid == 6 ? outI6 :
              outI7;
assign        ib1 = !i_ok ? 0 :
              ibid == 0 ? outI1 :
              ibid == 1 ? outI2 :
              ibid == 2 ? outI3 :
              ibid == 3 ? outI4 :
              ibid == 4 ? outI5 :
              ibid == 5 ? outI6 :
              ibid == 6 ? outI7 :
              outI0;
assign        ib2 = !i_ok ? 0 :
              ibid == 0 ? outI2 :
              ibid == 1 ? outI3 :
              ibid == 2 ? outI4 :
              ibid == 3 ? outI5 :
              ibid == 4 ? outI6 :
              ibid == 5 ? outI7 :
              ibid == 6 ? outI0 :
              outI1;
assign        ib3 = !i_ok ? 0 :
              ibid == 0 ? outI3 :
              ibid == 1 ? outI4 :
              ibid == 2 ? outI5 :
              ibid == 3 ? outI6 :
              ibid == 4 ? outI7 :
              ibid == 5 ? outI0 :
              ibid == 6 ? outI1 :
              outI2;
assign        ib4 = !i_ok ? 0 :
              ibid == 0 ? outI4 :
              ibid == 1 ? outI5 :
              ibid == 2 ? outI6 :
              ibid == 3 ? outI7 :
              ibid == 4 ? outI0 :
              ibid == 5 ? outI1 :
              ibid == 6 ? outI2 :
              outI3;
assign        ib5 = !i_ok ? 0 :
              ibid == 0 ? outI5 :
              ibid == 1 ? outI6 :
              ibid == 2 ? outI7 :
              ibid == 3 ? outI0 :
              ibid == 4 ? outI1 :
```

```
                ibid == 5 ? outI2 :
                ibid == 6 ? outI3 :
                outI4;

assign          instr[ 7: 0] = ib0;
assign          instr[15: 8] = ib1;
assign          instr[23:16] = ib2;
assign          instr[31:24] = ib3;
assign          instr[39:32] = ib4;
assign          instr[47:40] = ib5;

assign          m_ok =
                (!renable & !wenable | (maddr + 3) < memsize);

assign          addrD0 = mbid >= 5 ? mip1 : mindex;
assign          addrD1 = mbid >= 6 ? mip1 : mindex;
assign          addrD2 = mbid >= 7 ? mip1 : mindex;
assign          addrD3 = mindex;
assign          addrD4 = mindex;
assign          addrD5 = mindex;
assign          addrD6 = mindex;
assign          addrD7 = mindex;

// Get the bytes of data;
assign          db0 = !m_ok ? 0 :
                mbid == 0 ? outD0 :
                mbid == 1 ? outD1 :
                mbid == 2 ? outD2 :
                mbid == 3 ? outD3 :
                mbid == 4 ? outD4 :
                mbid == 5 ? outD5 :
                mbid == 6 ? outD6 :
                outD7;
assign          db1 = !m_ok ? 0 :
                mbid == 0 ? outD1 :
                mbid == 1 ? outD2 :
                mbid == 2 ? outD3 :
                mbid == 3 ? outD4 :
                mbid == 4 ? outD5 :
                mbid == 5 ? outD6 :
                mbid == 6 ? outD7 :
                outD0;
assign          db2 = !m_ok ? 0 :
                mbid == 0 ? outD2 :
                mbid == 1 ? outD3 :
                mbid == 2 ? outD4 :
                mbid == 3 ? outD5 :
                mbid == 4 ? outD6 :
                mbid == 5 ? outD7 :
                mbid == 6 ? outD0 :
```

```
                outD1;
assign          db3 = !m_ok ? 0 :
                mbid == 0 ? outD3 :
                mbid == 1 ? outD4 :
                mbid == 2 ? outD5 :
                mbid == 3 ? outD6 :
                mbid == 4 ? outD7 :
                mbid == 5 ? outD0 :
                mbid == 6 ? outD1 :
                outD2;

assign          rdata[ 7: 0] = db0;
assign          rdata[15: 8] = db1;
assign          rdata[23:16] = db2;
assign          rdata[31:24] = db3;

wire [7:0]      wd0 = wdata[7:0];
wire [7:0]      wd1 = wdata[15:8];
wire [7:0]      wd2 = wdata[23:16];
wire [7:0]      wd3 = wdata[31:24];

assign          inD0 =
                mbid == 5 ? wd3 :
                mbid == 6 ? wd2 :
                mbid == 7 ? wd1 :
                mbid == 0 ? wd0 :
                0;

assign          inD1 =
                mbid == 6 ? wd3 :
                mbid == 7 ? wd2 :
                mbid == 0 ? wd1 :
                mbid == 1 ? wd0 :
                0;

assign          inD2 =
                mbid == 7 ? wd3 :
                mbid == 0 ? wd2 :
                mbid == 1 ? wd1 :
                mbid == 2 ? wd0 :
                0;

assign          inD3 =
                mbid == 0 ? wd3 :
                mbid == 1 ? wd2 :
                mbid == 2 ? wd1 :
                mbid == 3 ? wd0 :
                0;

assign          inD4 =
```

```
                 mbid == 1 ? wd3 :
                 mbid == 2 ? wd2 :
                 mbid == 3 ? wd1 :
                 mbid == 4 ? wd0 :
                 0;

   assign        inD5 =
                 mbid == 2 ? wd3 :
                 mbid == 3 ? wd2 :
                 mbid == 4 ? wd1 :
                 mbid == 5 ? wd0 :
                 0;

   assign        inD6 =
                 mbid == 3 ? wd3 :
                 mbid == 4 ? wd2 :
                 mbid == 5 ? wd1 :
                 mbid == 6 ? wd0 :
                 0;

   assign        inD7 =
                 mbid == 4 ? wd3 :
                 mbid == 5 ? wd2 :
                 mbid == 6 ? wd1 :
                 mbid == 7 ? wd0 :
                 0;

   // Which banks get written
   assign        dwEn0 = wenable & (mbid <= 0 | mbid >= 5);
   assign        dwEn1 = wenable & (mbid <= 1 | mbid >= 6);
   assign        dwEn2 = wenable & (mbid <= 2 | mbid >= 7);
   assign        dwEn3 = wenable & (mbid <= 3);
   assign        dwEn4 = wenable & (mbid >= 1 & mbid <= 4);
   assign        dwEn5 = wenable & (mbid >= 2 & mbid <= 5);
   assign        dwEn6 = wenable & (mbid >= 3 & mbid <= 6);
   assign        dwEn7 = wenable & (mbid >= 4);

endmodule


// Combinational blocks

// Fetch stage

// Split instruction byte into icode and ifun fields
module split(ibyte, icode, ifun);
   input  [7:0] ibyte;
   output [3:0] icode;
   output [3:0] ifun;

```

```
   assign       icode = ibyte[7:4];
   assign       ifun  = ibyte[3:0];
endmodule

// Extract immediate word from 5 bytes of instruction
module align(ibytes, need_regids, rA, rB, valC);
   input [39:0]  ibytes;
   input         need_regids;
   output [3:0]  rA;
   output [3:0]  rB;
   output [31:0] valC;
   assign        rA = ibytes[7:4];
   assign        rB = ibytes[3:0];
   assign        valC = need_regids ? ibytes[39:8] : ibytes[31:0];
endmodule

// PC incrementer
module pc_increment(pc, need_regids, need_valC, valP);
   input [31:0]  pc;
   input         need_regids;
   input         need_valC;
   output [31:0] valP;
   assign        valP = pc + 1 + 4*need_valC + need_regids;
endmodule

// Execute Stage

// ALU
module alu(aluA, aluB, alufun, valE, new_cc);
   input [31:0]  aluA, aluB;    // Data inputs
   input [3:0]   alufun;        // ALU function
   output [31:0] valE;          // Data Output
   output [2:0]  new_cc;        // New values for ZF, SF, OF

   parameter     ALUADD = 4'h0;
   parameter     ALUSUB = 4'h1;
   parameter     ALUAND = 4'h2;
   parameter     ALUXOR = 4'h3;

   assign        valE =
                 alufun == ALUSUB ? aluB - aluA :
                 alufun == ALUAND ? aluB & aluA :
                 alufun == ALUXOR ? aluB ^ aluA :
                 aluB + aluA;
   assign        new_cc[2] = (valE == 0);  // ZF
   assign        new_cc[1] = valE[31];     // SF
   assign        new_cc[0] =               // OF
                   alufun == ALUADD ?
                      (aluA[31] == aluB[31])  & (aluA[31] != valE[31]) :
                   alufun == ALUSUB ?
```

```
                      (~aluA[31] == aluB[31]) & (aluB[31] != valE[31]) :
                   0;
endmodule


// Condition code register
module cc(cc, new_cc, set_cc, reset, clock);
   output[2:0] cc;
   input [2:0] new_cc;
   input       set_cc;
   input       reset;
   input       clock;

   cenrreg #(3) c(cc, new_cc, set_cc, reset, 3'b100, clock);
endmodule

// branch condition logic
module cond(ifun, cc, Cnd);
   input [3:0] ifun;
   input [2:0] cc;
   output      Cnd;

   wire        zf = cc[2];
   wire        sf = cc[1];
   wire        of = cc[0];

   // Jump & move conditions.
   parameter   C_YES  = 4'h0;
   parameter   C_LE   = 4'h1;
   parameter   C_L    = 4'h2;
   parameter   C_E    = 4'h3;
   parameter   C_NE   = 4'h4;
   parameter   C_GE   = 4'h5;
   parameter   C_G    = 4'h6;

   assign      Cnd =
               (ifun == C_YES) |               //
               (ifun == C_LE & ((sf^of)|zf)) | // <=
               (ifun == C_L  & (sf^of)) |      // <
               (ifun == C_E  & zf) |           // ==
               (ifun == C_NE & ~zf) |          // !=
               (ifun == C_GE & (~sf^of)) |     // >=
               (ifun == C_G  & (~sf^of)&~zf);  // >

endmodule

// --------------------------------------------------------------------
// Processor implementation
// --------------------------------------------------------------------
```

```
// The processor can run in 5 different modes:
// RUN:      Normal operation
// RESET:    Sets PC to 0, clears all pipe registers;
//           Initializes condition codes
// DOWNLOAD: Download bytes from controller into memory
// UPLOAD:   Upload bytes from memory to controller
// STATUS:   Upload other status information to controller

// Processor module
module processor(mode, udaddr, idata, odata, stat, clock);
  input  [2:0]  mode;   // Signal operating mode to processor
  input  [31:0] udaddr; // Upload/download address
  input  [31:0] idata;  // Download data word
  output [31:0] odata;  // Upload data word
  output [2:0]  stat;   // Status
  input         clock;  // Clock input

// Define modes
  parameter RUN_MODE = 0;      // Normal operation
  parameter RESET_MODE = 1;    // Resetting processor;
  parameter DOWNLOAD_MODE = 2; // Transfering to memory
  parameter UPLOAD_MODE = 3;   // Reading from memory
  // Uploading register & other status information
  parameter STATUS_MODE = 4;

// Constant values

  // Instruction codes
  parameter IHALT =   4'h0;
  parameter INOP =    4'h1;
  parameter IRRMOVL = 4'h2;
  parameter IIRMOVL = 4'h3;
  parameter IRMMOVL = 4'h4;
  parameter IMRMOVL = 4'h5;
  parameter IOPL =    4'h6;
  parameter IJXX =    4'h7;
  parameter ICALL =   4'h8;
  parameter IRET =    4'h9;
  parameter IPUSHL =  4'hA;
  parameter IPOPL =   4'hB;
  parameter IIADDL =  4'hC;
  parameter ILEAVE =  4'hD;
  parameter IPOP2 =   4'hE;

  // Function codes
  parameter FNONE = 4'h0;

  // Jump conditions
  parameter UNCOND = 4'h0;
```

```
  // Register IDs
  parameter RESP =  4'h4;
  parameter REBP =  4'h5;
  parameter RNONE = 4'hF;

  // ALU operations
  parameter ALUADD = 4'h0;

  // Status conditions
  parameter SBUB = 3'h0;
  parameter SAOK = 3'h1;
  parameter SHLT = 3'h2;
  parameter SADR = 3'h3;
  parameter SINS = 3'h4;
  parameter SPIP = 3'h5;

// Fetch stage signals
  wire [31:0] f_predPC, F_predPC, f_pc;
  wire        f_ok;
  wire        imem_error;
  wire [2:0]  f_stat;
  wire [47:0] f_instr;
  wire [3:0]  imem_icode;
  wire [3:0]  imem_ifun;
  wire [3:0]  f_icode;
  wire [3:0]  f_ifun;
  wire [3:0]  f_rA;
  wire [3:0]  f_rB;
  wire [31:0] f_valC;
  wire [31:0] f_valP;
  wire        need_regids;
  wire        need_valC;
  wire        instr_valid;
  wire        F_stall, F_bubble;

// Decode stage signals
  wire [2:0]  D_stat;
  wire [31:0] D_pc;
  wire [3:0]  D_icode;
  wire [3:0]  D_ifun;
  wire [3:0]  D_rA;
  wire [3:0]  D_rB;
  wire [31:0] D_valC;
  wire [31:0] D_valP;

  wire [31:0] d_valA;
  wire [31:0] d_valB;
  wire [31:0] d_rvalA;
  wire [31:0] d_rvalB;
```

```
  wire [3:0] d_dstE;
  wire [3:0] d_dstM;
  wire [3:0] d_srcA;
  wire [3:0] d_srcB;
  wire       D_stall, D_bubble;

// Execute stage signals
  wire [2:0]  E_stat;
  wire [31:0] E_pc;
  wire [3:0]  E_icode;
  wire [3:0]  E_ifun;
  wire [31:0] E_valC;
  wire [31:0] E_valA;
  wire [31:0] E_valB;
  wire [3:0]  E_dstE;
  wire [3:0]  E_dstM;
  wire [3:0]  E_srcA;
  wire [3:0]  E_srcB;

  wire [31:0] aluA;
  wire [31:0] aluB;
  wire        set_cc;
  wire [2:0]  cc;
  wire [2:0]  new_cc;
  wire [3:0]  alufun;
  wire        e_Cnd;
  wire [31:0] e_valE;
  wire [31:0] e_valA;
  wire [3:0]  e_dstE;
  wire        E_stall, E_bubble;

// Memory stage
  wire [2:0]  M_stat;
  wire [31:0] M_pc;
  wire [3:0]  M_icode;
  wire [3:0]  M_ifun;
  wire        M_Cnd;
  wire [31:0] M_valE;
  wire [31:0] M_valA;
  wire [3:0]  M_dstE;
  wire [3:0]  M_dstM;

  wire [2:0]  m_stat;
  wire [31:0] mem_addr;
  wire [31:0] mem_data;
  wire        mem_read;
  wire        mem_write;
  wire [31:0] m_valM;
  wire        M_stall, M_bubble;
  wire        m_ok;
```

```
// Write-back stage
  wire [2:0]  W_stat;
  wire [31:0] W_pc;
  wire [3:0]  W_icode;
  wire [31:0] W_valE;
  wire [31:0] W_valM;
  wire [3:0]  W_dstE;
  wire [3:0]  W_dstM;
  wire [31:0] w_valE;
  wire [31:0] w_valM;
  wire [3:0]  w_dstE;
  wire [3:0]  w_dstM;
  wire        W_stall, W_bubble;

  // Global status
  wire [2:0] Stat;

// Debugging logic
  wire [31:0] eax, ecx, edx, ebx, esp, ebp, esi, edi;
  wire zf = cc[2];
  wire sf = cc[1];
  wire of = cc[0];

// Control signals
  wire resetting = (mode == RESET_MODE);
  wire uploading = (mode == UPLOAD_MODE);
  wire downloading = (mode == DOWNLOAD_MODE);
  wire running = (mode == RUN_MODE);
  wire getting_info = (mode == STATUS_MODE);
// Logic to control resetting of pipeline registers
  wire F_reset = F_bubble | resetting;
  wire D_reset = D_bubble | resetting;
  wire E_reset = E_bubble | resetting;
  wire M_reset = M_bubble | resetting;
  wire W_reset = W_bubble | resetting;

  // Processor status
  assign stat = Stat;
  // Output data
  assign odata =
   // When getting status, get either register or special status value
    getting_info ?
     (udaddr == 0 ? eax :
      udaddr == 4 ? ecx :
      udaddr == 8 ? edx :
      udaddr == 12 ? ebx :
      udaddr == 16 ? esp :
      udaddr == 20 ? ebp :
      udaddr == 24 ? esi :
```

```
      udaddr == 28 ? edi :
      udaddr == 32 ? cc :
      udaddr == 36 ? W_pc : 0)
    : m_valM;

// Pipeline registers
// All implemented with module
//    preg(out, in, stall, bubble, bubbleval, clock)

// All pipeline registers are implemented with module
//    preg(out, in, stall, bubble, bubbleval, clock)
// F Register
  preg #(32) F_predPC_reg(F_predPC, f_predPC, F_stall, F_reset, 0, clock);
// D Register
  preg #(3)  D_stat_reg(D_stat, f_stat, D_stall, D_reset, SBUB, clock);
  preg #(32) D_pc_reg(D_pc, f_pc, D_stall, D_reset, 0, clock);
  preg #(4)  D_icode_reg(D_icode, f_icode, D_stall, D_reset, INOP, clock);
  preg #(4)  D_ifun_reg(D_ifun, f_ifun, D_stall, D_reset, FNONE, clock);
  preg #(4)  D_rA_reg(D_rA, f_rA, D_stall, D_reset, RNONE, clock);
  preg #(4)  D_rB_reg(D_rB, f_rB, D_stall, D_reset, RNONE, clock);
  preg #(32) D_valC_reg(D_valC, f_valC, D_stall, D_reset, 0, clock);
  preg #(32) D_valP_reg(D_valP, f_valP, D_stall, D_reset, 0, clock);
// E Register
  preg #(3)  E_stat_reg(E_stat, D_stat, E_stall, E_reset, SBUB, clock);
  preg #(32) E_pc_reg(E_pc, D_pc, E_stall, E_reset, 0, clock);
  preg #(4)  E_icode_reg(E_icode, D_icode, E_stall, E_reset, INOP, clock);
  preg #(4)  E_ifun_reg(E_ifun, D_ifun, E_stall, E_reset, FNONE, clock);
  preg #(32) E_valC_reg(E_valC, D_valC, E_stall, E_reset, 0, clock);
  preg #(32) E_valA_reg(E_valA, d_valA, E_stall, E_reset, 0, clock);
  preg #(32) E_valB_reg(E_valB, d_valB, E_stall, E_reset, 0, clock);
  preg #(4)  E_dstE_reg(E_dstE, d_dstE, E_stall, E_reset, RNONE, clock);
  preg #(4)  E_dstM_reg(E_dstM, d_dstM, E_stall, E_reset, RNONE, clock);
  preg #(4)  E_srcA_reg(E_srcA, d_srcA, E_stall, E_reset, RNONE, clock);
  preg #(4)  E_srcB_reg(E_srcB, d_srcB, E_stall, E_reset, RNONE, clock);
// M Register
  preg #(3)  M_stat_reg(M_stat, E_stat, M_stall, M_reset, SBUB, clock);
  preg #(32) M_pc_reg(M_pc, E_pc, M_stall, M_reset, 0, clock);
  preg #(4)  M_icode_reg(M_icode, E_icode, M_stall, M_reset, INOP, clock);
  preg #(4)  M_ifun_reg(M_ifun, E_ifun, M_stall, M_reset, FNONE, clock);
  preg #(1)  M_Cnd_reg(M_Cnd, e_Cnd, M_stall, M_reset, 0, clock);
  preg #(32) M_valE_reg(M_valE, e_valE, M_stall, M_reset, 0, clock);
  preg #(32) M_valA_reg(M_valA, e_valA, M_stall, M_reset, 0, clock);
  preg #(4)  M_dstE_reg(M_dstE, e_dstE, M_stall, M_reset, RNONE, clock);
  preg #(4)  M_dstM_reg(M_dstM, E_dstM, M_stall, M_reset, RNONE, clock);
// W Register
  preg #(3)  W_stat_reg(W_stat, m_stat, W_stall, W_reset, SBUB, clock);
  preg #(32) W_pc_reg(W_pc, M_pc, W_stall, W_reset, 0, clock);
  preg #(4)  W_icode_reg(W_icode, M_icode, W_stall, W_reset, INOP, clock);
  preg #(32) W_valE_reg(W_valE, M_valE, W_stall, W_reset, 0, clock);
  preg #(32) W_valM_reg(W_valM, m_valM, W_stall, W_reset, 0, clock);
```

```
  preg #(4)  W_dstE_reg(W_dstE, M_dstE, W_stall, W_reset, RNONE, clock);
  preg #(4)  W_dstM_reg(W_dstM, M_dstM, W_stall, W_reset, RNONE, clock);

// Fetch stage logic
  split split(f_instr[7:0], imem_icode, imem_ifun);
  align align(f_instr[47:8], need_regids, f_rA, f_rB, f_valC);
  pc_increment pci(f_pc, need_regids, need_valC, f_valP);

// Decode stage
  regfile regf(w_dstE, w_valE, w_dstM, w_valM,
               d_srcA, d_rvalA, d_srcB, d_rvalB, resetting, clock,
               eax, ecx, edx, ebx, esp, ebp, esi, edi);

// Execute stage
  alu alu(aluA, aluB, alufun, e_valE, new_cc);
  cc  ccreg(cc, new_cc,
            // Only update CC when everything is running normally
            running & set_cc,
            resetting, clock);
  cond cond_check(E_ifun, cc, e_Cnd);

// Memory stage
  bmemory m(
    // Only update memory when everything is running normally
    // or when downloading
    (downloading | uploading) ? udaddr : mem_addr, // Read/Write address
    (running & mem_write) | downloading,  // When to write to memory
    downloading ? idata : M_valA,         // Write data
    (running & mem_read) | uploading,     // When to read memory
    m_valM,                               // Read data
    m_ok,
    f_pc, f_instr, f_ok, clock);          // Instruction memory access

   assign imem_error = ~f_ok;
   assign dmem_error = ~m_ok;

// Write-back stage logic

// Control logic
// --------------------------------------------------------------------
// The following code is generated from the HCL description of the
// pipeline control using the hcl2v program
// --------------------------------------------------------------------
assign f_pc =
    (((M_icode == IJXX) & ~M_Cnd) ? M_valA : (W_icode == IRET) ? W_valM :
      F_predPC);

assign f_icode =
    (imem_error ? INOP : imem_icode);

```

```
assign f_ifun =
    (imem_error ? FNONE : imem_ifun);

assign instr_valid =
    (f_icode == INOP | f_icode == IHALT | f_icode == IRRMOVL | f_icode ==
      IIRMOVL | f_icode == IRMMOVL | f_icode == IMRMOVL | f_icode == IOPL
       | f_icode == IJXX | f_icode == ICALL | f_icode == IRET | f_icode ==
      IPUSHL | f_icode == IPOPL);

assign f_stat =
    (imem_error ? SADR : ~instr_valid ? SINS : (f_icode == IHALT) ? SHLT :
      SAOK);

assign need_regids =
    (f_icode == IRRMOVL | f_icode == IOPL | f_icode == IPUSHL | f_icode ==
      IPOPL | f_icode == IIRMOVL | f_icode == IRMMOVL | f_icode == IMRMOVL)
    ;

assign need_valC =
    (f_icode == IIRMOVL | f_icode == IRMMOVL | f_icode == IMRMOVL | f_icode
       == IJXX | f_icode == ICALL);

assign f_predPC =
    ((f_icode == IJXX | f_icode == ICALL) ? f_valC : f_valP);

assign d_srcA =
    ((D_icode == IRRMOVL | D_icode == IRMMOVL | D_icode == IOPL | D_icode
        == IPUSHL) ? D_rA : (D_icode == IPOPL | D_icode == IRET) ? RESP :
      RNONE);

assign d_srcB =
    ((D_icode == IOPL | D_icode == IRMMOVL | D_icode == IMRMOVL) ? D_rB : (
       D_icode == IPUSHL | D_icode == IPOPL | D_icode == ICALL | D_icode
        == IRET) ? RESP : RNONE);

assign d_dstE =
    ((D_icode == IRRMOVL | D_icode == IIRMOVL | D_icode == IOPL) ? D_rB : (
       D_icode == IPUSHL | D_icode == IPOPL | D_icode == ICALL | D_icode
        == IRET) ? RESP : RNONE);

assign d_dstM =
    ((D_icode == IMRMOVL | D_icode == IPOPL) ? D_rA : RNONE);

assign d_valA =
    ((D_icode == ICALL | D_icode == IJXX) ? D_valP : (d_srcA == e_dstE) ?
      e_valE : (d_srcA == M_dstM) ? m_valM : (d_srcA == M_dstE) ? M_valE :
      (d_srcA == W_dstM) ? W_valM : (d_srcA == W_dstE) ? W_valE : d_rvalA);

assign d_valB =
    ((d_srcB == e_dstE) ? e_valE : (d_srcB == M_dstM) ? m_valM : (d_srcB
```

```
         == M_dstE) ? M_valE : (d_srcB == W_dstM) ? W_valM : (d_srcB ==
        W_dstE) ? W_valE : d_rvalB);

assign aluA =
    ((E_icode == IRRMOVL | E_icode == IOPL) ? E_valA : (E_icode == IIRMOVL
         | E_icode == IRMMOVL | E_icode == IMRMOVL) ? E_valC : (E_icode ==
        ICALL | E_icode == IPUSHL) ? -4 : (E_icode == IRET | E_icode == IPOPL
        ) ? 4 : 0);

assign aluB =
    ((E_icode == IRMMOVL | E_icode == IMRMOVL | E_icode == IOPL | E_icode
         == ICALL | E_icode == IPUSHL | E_icode == IRET | E_icode == IPOPL)
       ? E_valB : (E_icode == IRRMOVL | E_icode == IIRMOVL) ? 0 : 0);

assign alufun =
    ((E_icode == IOPL) ? E_ifun : ALUADD);

assign set_cc =
    (((E_icode == IOPL) & ~(m_stat == SADR | m_stat == SINS | m_stat ==
          SHLT)) & ~(W_stat == SADR | W_stat == SINS | W_stat == SHLT));

assign e_valA =
    E_valA;

assign e_dstE =
    (((E_icode == IRRMOVL) & ~e_Cnd) ? RNONE : E_dstE);

assign mem_addr =
    ((M_icode == IRMMOVL | M_icode == IPUSHL | M_icode == ICALL | M_icode
         == IMRMOVL) ? M_valE : (M_icode == IPOPL | M_icode == IRET) ?
      M_valA : 0);

assign mem_read =
    (M_icode == IMRMOVL | M_icode == IPOPL | M_icode == IRET);

assign mem_write =
    (M_icode == IRMMOVL | M_icode == IPUSHL | M_icode == ICALL);

assign m_stat =
    (dmem_error ? SADR : M_stat);

assign w_dstE =
    W_dstE;

assign w_valE =
    W_valE;

assign w_dstM =
    W_dstM;
```

```
assign w_valM =
    W_valM;

assign Stat =
    ((W_stat == SBUB) ? SAOK : W_stat);

assign F_bubble =
    0;

assign F_stall =
    (((E_icode == IMRMOVL | E_icode == IPOPL) & (E_dstM == d_srcA | E_dstM
           == d_srcB)) | (IRET == D_icode | IRET == E_icode | IRET ==
        M_icode));

assign D_stall =
    ((E_icode == IMRMOVL | E_icode == IPOPL) & (E_dstM == d_srcA | E_dstM
         == d_srcB));

assign D_bubble =
    (((E_icode == IJXX) & ~e_Cnd) | (~((E_icode == IMRMOVL | E_icode ==
            IPOPL) & (E_dstM == d_srcA | E_dstM == d_srcB)) & (IRET ==
          D_icode | IRET == E_icode | IRET == M_icode)));

assign E_stall =
    0;

assign E_bubble =
    (((E_icode == IJXX) & ~e_Cnd) | ((E_icode == IMRMOVL | E_icode == IPOPL
          ) & (E_dstM == d_srcA | E_dstM == d_srcB)));

assign M_stall =
    0;

assign M_bubble =
    ((m_stat == SADR | m_stat == SINS | m_stat == SHLT) | (W_stat == SADR
         | W_stat == SINS | W_stat == SHLT));

assign W_stall =
    (W_stat == SADR | W_stat == SINS | W_stat == SHLT);

assign W_bubble =
    0;

// ---------------------------------------------------------------------
// End of code generated by hcl2v
// ---------------------------------------------------------------------
endmodule
```

References

[1] D. Thomas and P. Moorby. *The Verilog Hardware Description Language, Fifth Edition*. Springer, 2008.

Index