

CS:APP2e Web Aside ARCH:VLOG

Verilog Implementation of a Pipelined Y86 Processor*

Randal E. Bryant
David R. O'Hallaron

May 25, 2011

Notice

The material in this document is supplementary material to the book Computer Systems, A Programmer's Perspective, Second Edition, by Randal E. Bryant and David R. O'Hallaron, published by Prentice-Hall and copyrighted 2011. In this document, all references beginning with "CS:APP2e" are to this book. More information about the book is available at csapp.cs.cmu.edu.

This document is being made available to the public, subject to copyright provisions. You are free to copy and distribute it, but you should not use any of this material without attribution.

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,

*Copyright © 2010, R. E. Bryant, D. R. O'Hallaron. All rights reserved.

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`.

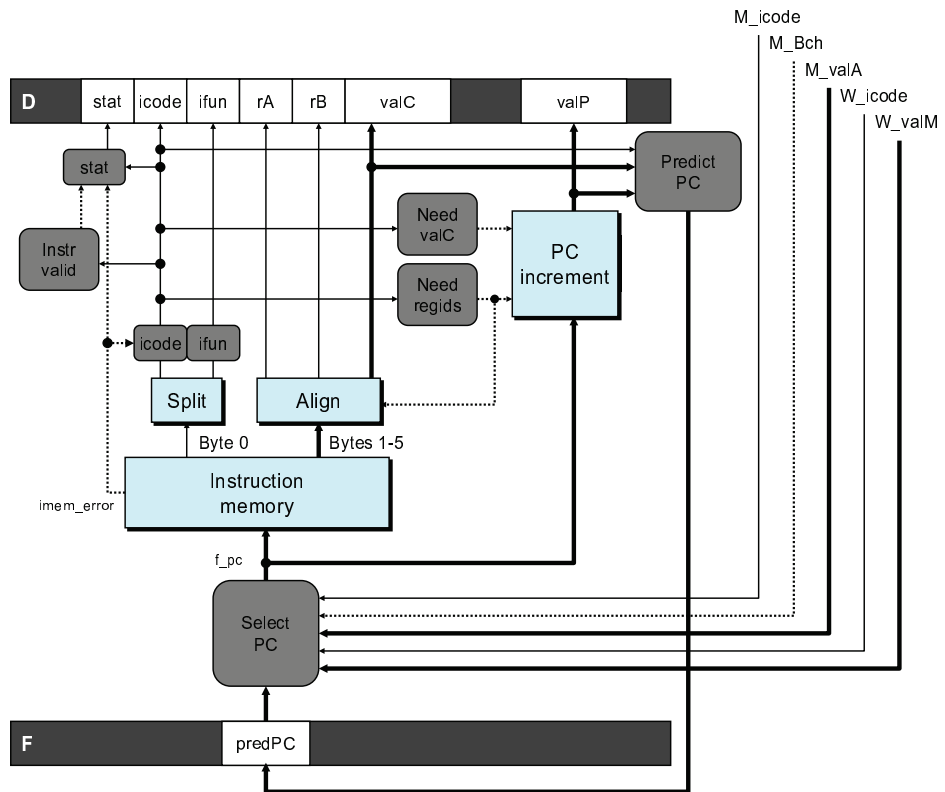


Figure 1: PIPE PC selection and fetch logic.

```

// 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, valeE, new_cc);
    input [31:0] aluA, aluB;    // Data inputs
    input [3:0] alufun;        // ALU function
    output [31:0] valeE;       // 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       valeE =
        alufun == ALUSUB ? aluB - aluA :
        alufun == ALUAND ? aluB & aluA :
        alufun == ALUXOR ? aluB ^ aluA :
        aluB + aluA;

    assign       new_cc[2] = (valeE == 0); // ZF
    assign       new_cc[1] = valeE[31];   // SF
    assign       new_cc[0] =
        alufun == ALUADD ?
            (aluA[31] == aluB[31]) & (aluA[31] != valeE[31]) :
        alufun == ALUSUB ?
            (~aluA[31] == aluB[31]) & (aluB[31] != valeE[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 overridden
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 data output will equal the sum, difference, bitwise EXCLUSIVE-OR, or bitwise AND of the two data inputs. 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;

    cenreg #(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 instantiation 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,
```

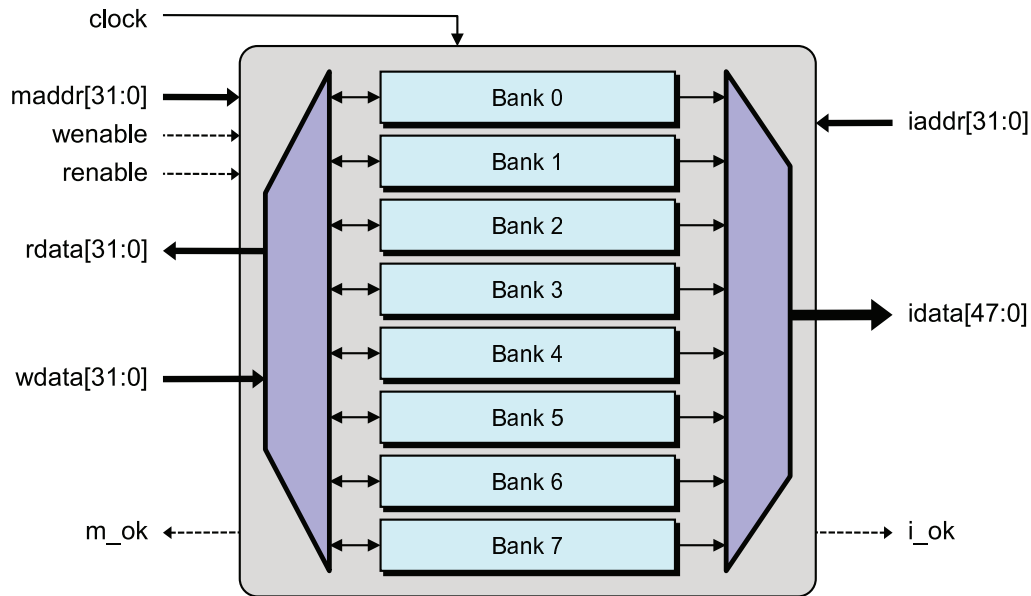


Figure 5: **Memory structure.** The memory consists of eight banks, each performing single-byte reads and writes.

```

        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.

addrsz: 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

```

1 // This module implements a dual-ported RAM.
2 // with clocked write and combinational read operations.
3 // This version matches the conceptual model presented in the CS:APP book,
4
5 module ram(clock, addrA, wEnA, wDatA, rEnA, rDatA,
6           addrB, wEnB, wDatB, rEnB, rDatB);
7
8     parameter wordsize = 8;    // Number of bits per word
9     parameter wordcount = 512; // Number of words in memory
10    // Number of address bits. Must be >= log wordcount
11    parameter addrsize = 9;
12
13    input    clock;                // Clock
14    // Port A
15    input [addrsize-1:0] addrA;    // Read/write address
16    input          wEnA;          // Write enable
17    input [wordsize-1:0] wDatA;    // Write data
18    input          rEnA;          // Read enable
19    output [wordsize-1:0] rDatA;   // Read data
20    // Port B
21    input [addrsize-1:0] addrB;    // Read/write address
22    input          wEnB;          // Write enable
23    input [wordsize-1:0] wDatB;    // Write data
24    input          rEnB;          // Read enable
25    output [wordsize-1:0] rDatB;   // Read data
26
27    // Actual storage
28    reg [wordsize-1:0] mem[wordcount-1:0];
29
30    always @(posedge clock)
31        begin
32            if (wEnA)
33                begin
34                    mem[addrA] <= wDatA;
35                end
36        end
37    // Combinational reads
38    assign rDatA = mem[addrA];
39
40    always @(posedge clock)
41        begin
42            if (wEnB)
43                begin
44                    mem[addrB] <= wDatB;
45                end
46        end
47    // Combinational reads
48    assign rDatB = mem[addrB];
49
50 endmodule

```

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

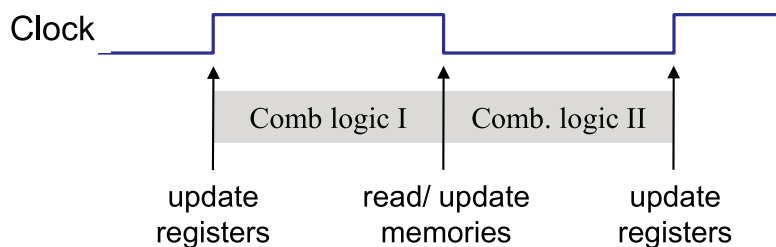


Figure 7: **Timing of synchronous RAM.** By having the memory be read and written on the falling clock edge, the combinational logic can be active both before (A) and after (B) the memory operation.

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.

```

1 // This module implements a dual-ported RAM.
2 // with clocked write and read operations.
3
4 module ram(clock, addrA, wEnA, wDatA, rEnA, rDatA,
5           addrB, wEnB, wDatB, rEnB, rDatB);
6
7 parameter wordsize = 8;    // Number of bits per word
8 parameter wordcount = 512; // Number of words in memory
9 // Number of address bits. Must be >= log wordcount
10 parameter addrsize = 9;
11
12
13 input  clock;                // Clock
14 // Port A
15 input [addrsize-1:0] addrA; // Read/write address
16 input  wEnA;                // Write enable
17 input [wordsize-1:0] wDatA; // Write data
18 input  rEnA;                // Read enable
19 output [wordsize-1:0] rDatA; // Read data
20 reg [wordsize-1:0] rDatA;
21 // Port B
22 input [addrsize-1:0] addrB; // Read/write address
23 input  wEnB;                // Write enable
24 input [wordsize-1:0] wDatB; // Write data
25 input  rEnB;                // Read enable
26 output [wordsize-1:0] rDatB; // Read data
27 reg [wordsize-1:0] rDatB;
28
29 reg[wordsize-1:0] mem[wordcount-1:0]; // Actual storage
30
31 always @(negedge clock)
32 begin
33     if (wEnA)
34     begin
35         mem[addrA] <= wDatA;
36     end
37     if (rEnA)
38         rDatA <= mem[addrA];
39 end
40
41 always @(negedge clock)
42 begin
43     if (wEnB)
44     begin
45         mem[addrB] <= wDatB;
46     end
47     if (rEnB)
48         rDatB <= mem[addrB];
49 end
50 endmodule

```

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

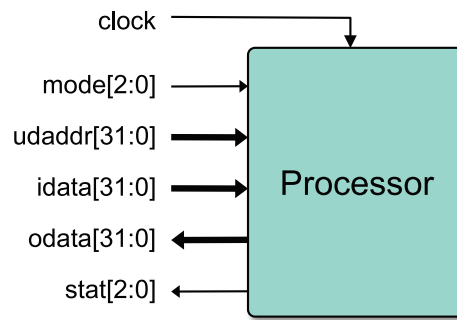


Figure 9: **Processor interface.** Mechanisms are included to upload and download memory data and processor state, and to operate the processor in different modes.

```

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.

Homework Problems

Homework Problem 2 ◆◆◆:

Generate a Verilog description of the SEQ processor suitable for simulation. You can use the same blocks as shown here for the PIPE processor, and you can generate the control logic from the HCL representation using the HCL2V program. Use the combinational RAM module (Figure 6) to implement the memory banks.

Homework Problem 3 ◆◆:

Suppose we wish to create a synthesizable version of the SEQ processor.

- A. Analyze what would happen if you were to use the synchronous RAM module (Figure 8) in an implementation of the SEQ processor (Problem 2.)
- B. Devise and implement (in Verilog) a clocking scheme for the registers and the memory banks that would enable the use of a synchronous RAM in an implementation of the SEQ processor.

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.

```

1 // -----
2 // Verilog representation of PIPE processor
3 // -----
4
5 // -----
6 // Memory module for implementing bank memories
7 // -----
8 // This module implements a dual-ported RAM.
9 // with clocked write and read operations.
10
11 module ram(clock, addrA, wEnA, wDatA, rEnA, rDatA,
12           addrB, wEnB, wDatB, rEnB, rDatB);
13
14 parameter wordsize = 8;    // Number of bits per word
15 parameter wordcount = 512; // Number of words in memory
16 // Number of address bits. Must be >= log wordcount
17 parameter addrsize = 9;
18
19
20 input  clock;                // Clock
21 // Port A
22 input [addrsize-1:0] addrA;  // Read/write address
23 input  wEnA;                // Write enable
24 input [wordsize-1:0] wDatA;  // Write data
25 input  rEnA;                // Read enable
26 output [wordsize-1:0] rDatA; // Read data
27 reg [wordsize-1:0] rDatA; //= line:arch:synchram:rDatA
28 // Port B
29 input [addrsize-1:0] addrB;  // Read/write address
30 input  wEnB;                // Write enable
31 input [wordsize-1:0] wDatB;  // Write data
32 input  rEnB;                // Read enable
33 output [wordsize-1:0] rDatB; // Read data
34 reg [wordsize-1:0] rDatB; //= line:arch:synchram:rDatB
35
36 reg[wordsize-1:0] mem[wordcount-1:0]; // Actual storage
37
38 // To make the pipeline processor work with synchronous reads, we
39 // operate the memory read operations on the negative
40 // edge of the clock. That makes the reading occur in the middle
41 // of the clock cycle---after the address inputs have been set
42 // and such that the results read from the memory can flow through
43 // more combinational logic before reaching the clocked registers

```

```

44
45 // For uniformity, we also make the memory write operation
46 // occur on the negative edge of the clock. That works OK
47 // in this design, because the write can occur as soon as the
48 // address & data inputs have been set.
49 always @(negedge clock)
50 begin
51     if (wEnA)
52     begin
53         mem[addrA] <= wDatA;
54     end
55     if (rEnA)
56         rDatA <= mem[addrA]; // = line:arch:synchram:readA
57 end
58
59 always @(negedge clock)
60 begin
61     if (wEnB)
62     begin
63         mem[addrB] <= wDatB;
64     end
65     if (rEnB)
66         rDatB <= mem[addrB]; // = line:arch:synchram:readB
67 end
68 endmodule
69
70 // -----
71 // Other building blocks
72 // -----
73
74 // Basic building blocks for constructing a Y86 processor.
75
76 // Different types of registers, all derivatives of module cenrreg
77
78 // Clocked register with enable signal and synchronous reset
79 // Default width is 8, but can be overridden
80 module cenrreg(out, in, enable, reset, resetval, clock);
81     parameter width = 8;
82     output [width-1:0] out;
83     reg [width-1:0] out;
84     input [width-1:0] in;
85     input enable;
86     input reset;
87     input [width-1:0] resetval;
88     input clock;
89
90     always
91         @(posedge clock)
92         begin
93             if (reset)

```

```

94         out <= resetval;
95     else if (enable)
96         out <= in;
97     end
98 endmodule
99
100 // Clocked register with enable signal.
101 // Default width is 8, but can be overridden
102 module cenreg(out, in, enable, clock);
103     parameter width = 8;
104     output [width-1:0] out;
105     input [width-1:0] in;
106     input enable;
107     input clock;
108
109     cenrreg #(width) c(out, in, enable, 0, 0, clock);
110 endmodule
111
112 // Basic clocked register. Default width is 8.
113 module creg(out, in, clock);
114     parameter width = 8;
115     output [width-1:0] out;
116     input [width-1:0] in;
117     input clock;
118
119     cenreg #(width) r(out, in, 1, clock);
120 endmodule
121
122 // Pipeline register. Uses reset signal to inject bubble
123 // When bubbling, must specify value that will be loaded
124 module preg(out, in, stall, bubble, bubbleval, clock);
125     parameter width = 8;
126     output [width-1:0] out;
127     input [width-1:0] in;
128     input stall, bubble;
129     input [width-1:0] bubbleval;
130     input clock;
131
132     cenrreg #(width) r(out, in, ~stall, bubble, bubbleval, clock);
133 endmodule
134
135 // Register file
136 module regfile(dstE, valE, dstM, valM, srcA, valA, srcB, valB,
137     reset, clock, eax, ecx, edx, ebx, esp, ebp, esi, edi);
138     input [3:0] dstE;
139     input [31:0] valE;
140     input [3:0] dstM;
141     input [31:0] valM;
142     input [3:0] srcA;
143     output [31:0] valA;

```

```

144 input [3:0] srcB;
145 output [31:0] valB;
146 input reset; // Set registers to 0
147 input clock;
148 // Make individual registers visible for debugging
149 output [31:0] eax, ecx, edx, ebx, esp, ebp, esi, edi;
150
151 // Define names for registers used in HCL code
152 parameter REAX = 4'h0;
153 parameter RECX = 4'h1;
154 parameter REDX = 4'h2;
155 parameter REBX = 4'h3;
156 parameter RESP = 4'h4;
157 parameter REBP = 4'h5;
158 parameter RESI = 4'h6;
159 parameter REDI = 4'h7;
160 parameter RNONE = 4'hf;
161
162 // Input data for each register
163 wire [31:0] eax_dat, ecx_dat, edx_dat, ebx_dat,
164 esp_dat, ebp_dat, esi_dat, edi_dat;
165
166 // Input write controls for each register
167 wire eax_wrt, ecx_wrt, edx_wrt, ebx_wrt,
168 esp_wrt, ebp_wrt, esi_wrt, edi_wrt;
169
170 // Implement with clocked registers
171 cenrreg #(32) eax_reg(eax, eax_dat, eax_wrt, reset, 0, clock);
172 cenrreg #(32) ecx_reg(ecx, ecx_dat, ecx_wrt, reset, 0, clock);
173 cenrreg #(32) edx_reg(edx, edx_dat, edx_wrt, reset, 0, clock);
174 cenrreg #(32) ebx_reg(ebx, ebx_dat, ebx_wrt, reset, 0, clock);
175 cenrreg #(32) esp_reg(esp, esp_dat, esp_wrt, reset, 0, clock);
176 cenrreg #(32) ebp_reg(ebp, ebp_dat, ebp_wrt, reset, 0, clock);
177 cenrreg #(32) esi_reg(esi, esi_dat, esi_wrt, reset, 0, clock);
178 cenrreg #(32) edi_reg(edi, edi_dat, edi_wrt, reset, 0, clock);
179
180 // Reads occur like combinational logic
181 assign valA =
182 srcA == REAX ? eax :
183 srcA == RECX ? ecx :
184 srcA == REDX ? edx :
185 srcA == REBX ? ebx :
186 srcA == RESP ? esp :
187 srcA == REBP ? ebp :
188 srcA == RESI ? esi :
189 srcA == REDI ? edi :
190 0;
191
192 assign valB =
193 srcB == REAX ? eax :

```

```

194         srcB == RECX ? ecx :
195         srcB == REDX ? edx :
196         srcB == REBX ? ebx :
197         srcB == RESP ? esp :
198         srcB == REBP ? ebp :
199         srcB == RESI ? esi :
200         srcB == REDI ? edi :
201         0;
202
203     assign     eax_dat = dstM == REAX ? valM : vale;
204     assign     ecx_dat = dstM == RECX ? valM : vale;
205     assign     edx_dat = dstM == REDX ? valM : vale;
206     assign     ebx_dat = dstM == REBX ? valM : vale;
207     assign     esp_dat = dstM == RESP ? valM : vale;
208     assign     ebp_dat = dstM == REBP ? valM : vale;
209     assign     esi_dat = dstM == RESI ? valM : vale;
210     assign     edi_dat = dstM == REDI ? valM : vale;
211
212     assign     eax_wrt = dstM == REAX | dstE == REAX;
213     assign     ecx_wrt = dstM == RECX | dstE == RECX;
214     assign     edx_wrt = dstM == REDX | dstE == REDX;
215     assign     ebx_wrt = dstM == REBX | dstE == REBX;
216     assign     esp_wrt = dstM == RESP | dstE == RESP;
217     assign     ebp_wrt = dstM == REBP | dstE == REBP;
218     assign     esi_wrt = dstM == RESI | dstE == RESI;
219     assign     edi_wrt = dstM == REDI | dstE == REDI;
220
221 endmodule
222
223 // Memory. This memory design uses 8 memory banks, each
224 // of which is one byte wide. Banking allows us to select an
225 // arbitrary set of 6 contiguous bytes for instruction reading
226 // and an arbitrary set of 4 contiguous bytes
227 // for data reading & writing.
228 // It uses an external RAM module from either the file
229 // combram.v (using combinational reads)
230 // or synchram.v (using clocked reads)
231 // The SEQ & SEQ+ processors only work with combram.v.
232 // PIPE works with either.
233
234 module bmemory(maddr, wenable, wdata, renable, rdata, m_ok,
235               iaddr, instr, i_ok, clock);
236     parameter memsize = 4096; // Number of bytes in memory
237     input [31:0] maddr; // Read/Write address
238     input wenable; // Write enable
239     input [31:0] wdata; // Write data
240     input renable; // Read enable
241     output [31:0] rdata; // Read data
242     output m_ok; // Read & write addresses within range
243     input [31:0] iaddr; // Instruction address

```

```

244 output [47:0] instr;    // 6 bytes of instruction
245 output      i_ok;     // Instruction address within range
246 input      clock;
247
248 wire [7:0]   ib0, ib1, ib2, ib3, ib4, ib5; // Instruction bytes
249 wire [7:0]   db0, db1, db2, db3;         // Data bytes
250
251 wire [2:0]   ibid = iaddr[2:0];         // Instruction Bank ID
252 wire [28:0] iindex = iaddr[31:3];      // Address within bank
253 wire [28:0] iip1 = iindex+1;          // Next address within bank
254
255 wire [2:0]   mbid = maddr[2:0];         // Data Bank ID
256 wire [28:0] mindex = maddr[31:3];      // Address within bank
257 wire [28:0] mip1 = mindex+1;          // Next address within bank
258
259 // Instruction addresses for each bank
260 wire [28:0]  addrI0, addrI1, addrI2, addrI3, addrI4, addrI5, addrI6, addrI7;
261 // Instruction data for each bank
262 wire [7:0]   outI0, outI1, outI2, outI3, outI4, outI5, outI6, outI7;
263
264 // Data addresses for each bank
265 wire [28:0]  addrD0, addrD1, addrD2, addrD3, addrD4, addrD5, addrD6, addrD7;
266 // Data output for each bank
267 wire [7:0]   outD0, outD1, outD2, outD3, outD4, outD5, outD6, outD7;
268 // Data input for each bank
269 wire [7:0]   inD0, inD1, inD2, inD3, inD4, inD5, inD6, inD7;
270 // Data write enable signals for each bank
271 wire         dwEn0, dwEn1, dwEn2, dwEn3, dwEn4, dwEn5, dwEn6, dwEn7;
272
273 // The bank memories
274 ram #(8, memsize/8, 29) bank0(clock, addrI0, 0, 0, 1, outI0, // Instruction
275                               addrD0, dwEn0, inD0, renable, outD0); // Data
276
277 ram #(8, memsize/8, 29) bank1(clock, addrI1, 0, 0, 1, outI1, // Instruction
278                               addrD1, dwEn1, inD1, renable, outD1); // Data
279
280 ram #(8, memsize/8, 29) bank2(clock, addrI2, 0, 0, 1, outI2, // Instruction
281                               addrD2, dwEn2, inD2, renable, outD2); // Data
282
283 ram #(8, memsize/8, 29) bank3(clock, addrI3, 0, 0, 1, outI3, // Instruction
284                               addrD3, dwEn3, inD3, renable, outD3); // Data
285
286 ram #(8, memsize/8, 29) bank4(clock, addrI4, 0, 0, 1, outI4, // Instruction
287                               addrD4, dwEn4, inD4, renable, outD4); // Data
288
289 ram #(8, memsize/8, 29) bank5(clock, addrI5, 0, 0, 1, outI5, // Instruction
290                               addrD5, dwEn5, inD5, renable, outD5); // Data
291
292 ram #(8, memsize/8, 29) bank6(clock, addrI6, 0, 0, 1, outI6, // Instruction
293                               addrD6, dwEn6, inD6, renable, outD6); // Data

```

```

294
295 ram #(8, memsize/8, 29) bank7(clock, addrI7, 0, 0, 1, outI7, // Instruction
296 addrD7, dwEn7, ind7, renable, outD7); // Data
297
298
299 // Determine the instruction addresses for the banks
300 assign      addrI0 = ibid >= 3 ? iip1 : iindex;
301 assign      addrI1 = ibid >= 4 ? iip1 : iindex;
302 assign      addrI2 = ibid >= 5 ? iip1 : iindex;
303 assign      addrI3 = ibid >= 6 ? iip1 : iindex;
304 assign      addrI4 = ibid >= 7 ? iip1 : iindex;
305 assign      addrI5 = iindex;
306 assign      addrI6 = iindex;
307 assign      addrI7 = iindex;
308
309 // Get the bytes of the instruction
310 assign      i_ok =
311            (iaddr + 5) < memsize;
312
313 assign      ib0 = !i_ok ? 0 :
314            ibid == 0 ? outI0 :
315            ibid == 1 ? outI1 :
316            ibid == 2 ? outI2 :
317            ibid == 3 ? outI3 :
318            ibid == 4 ? outI4 :
319            ibid == 5 ? outI5 :
320            ibid == 6 ? outI6 :
321            outI7;
322 assign      ib1 = !i_ok ? 0 :
323            ibid == 0 ? outI1 :
324            ibid == 1 ? outI2 :
325            ibid == 2 ? outI3 :
326            ibid == 3 ? outI4 :
327            ibid == 4 ? outI5 :
328            ibid == 5 ? outI6 :
329            ibid == 6 ? outI7 :
330            outI0;
331 assign      ib2 = !i_ok ? 0 :
332            ibid == 0 ? outI2 :
333            ibid == 1 ? outI3 :
334            ibid == 2 ? outI4 :
335            ibid == 3 ? outI5 :
336            ibid == 4 ? outI6 :
337            ibid == 5 ? outI7 :
338            ibid == 6 ? outI0 :
339            outI1;
340 assign      ib3 = !i_ok ? 0 :
341            ibid == 0 ? outI3 :
342            ibid == 1 ? outI4 :
343            ibid == 2 ? outI5 :

```

```

344         ibid == 3 ? outI6 :
345         ibid == 4 ? outI7 :
346         ibid == 5 ? outI0 :
347         ibid == 6 ? outI1 :
348         outI2;
349     assign    ib4 = !i_ok ? 0 :
350             ibid == 0 ? outI4 :
351             ibid == 1 ? outI5 :
352             ibid == 2 ? outI6 :
353             ibid == 3 ? outI7 :
354             ibid == 4 ? outI0 :
355             ibid == 5 ? outI1 :
356             ibid == 6 ? outI2 :
357             outI3;
358     assign    ib5 = !i_ok ? 0 :
359             ibid == 0 ? outI5 :
360             ibid == 1 ? outI6 :
361             ibid == 2 ? outI7 :
362             ibid == 3 ? outI0 :
363             ibid == 4 ? outI1 :
364             ibid == 5 ? outI2 :
365             ibid == 6 ? outI3 :
366             outI4;
367
368     assign    instr[ 7: 0] = ib0;
369     assign    instr[15: 8] = ib1;
370     assign    instr[23:16] = ib2;
371     assign    instr[31:24] = ib3;
372     assign    instr[39:32] = ib4;
373     assign    instr[47:40] = ib5;
374
375     assign    m_ok =
376             (!renable & !wenable | (maddr + 3) < memsize);
377
378     assign    addrD0 = mbid >= 5 ? mip1 : mindex;
379     assign    addrD1 = mbid >= 6 ? mip1 : mindex;
380     assign    addrD2 = mbid >= 7 ? mip1 : mindex;
381     assign    addrD3 = mindex;
382     assign    addrD4 = mindex;
383     assign    addrD5 = mindex;
384     assign    addrD6 = mindex;
385     assign    addrD7 = mindex;
386
387     // Get the bytes of data;
388     assign    db0 = !m_ok ? 0 :
389             mbid == 0 ? outD0 :
390             mbid == 1 ? outD1 :
391             mbid == 2 ? outD2 :
392             mbid == 3 ? outD3 :
393             mbid == 4 ? outD4 :

```

```

394         mbid == 5 ? outD5 :
395         mbid == 6 ? outD6 :
396         outD7;
397     assign    db1 = !m_ok ? 0 :
398         mbid == 0 ? outD1 :
399         mbid == 1 ? outD2 :
400         mbid == 2 ? outD3 :
401         mbid == 3 ? outD4 :
402         mbid == 4 ? outD5 :
403         mbid == 5 ? outD6 :
404         mbid == 6 ? outD7 :
405         outD0;
406     assign    db2 = !m_ok ? 0 :
407         mbid == 0 ? outD2 :
408         mbid == 1 ? outD3 :
409         mbid == 2 ? outD4 :
410         mbid == 3 ? outD5 :
411         mbid == 4 ? outD6 :
412         mbid == 5 ? outD7 :
413         mbid == 6 ? outD0 :
414         outD1;
415     assign    db3 = !m_ok ? 0 :
416         mbid == 0 ? outD3 :
417         mbid == 1 ? outD4 :
418         mbid == 2 ? outD5 :
419         mbid == 3 ? outD6 :
420         mbid == 4 ? outD7 :
421         mbid == 5 ? outD0 :
422         mbid == 6 ? outD1 :
423         outD2;
424
425     assign    rdata[ 7: 0] = db0;
426     assign    rdata[15: 8] = db1;
427     assign    rdata[23:16] = db2;
428     assign    rdata[31:24] = db3;
429
430     wire [7:0] wd0 = wdata[7:0];
431     wire [7:0] wd1 = wdata[15:8];
432     wire [7:0] wd2 = wdata[23:16];
433     wire [7:0] wd3 = wdata[31:24];
434
435     assign    inD0 =
436         mbid == 5 ? wd3 :
437         mbid == 6 ? wd2 :
438         mbid == 7 ? wd1 :
439         mbid == 0 ? wd0 :
440         0;
441
442     assign    inD1 =
443         mbid == 6 ? wd3 :

```

```
444         mbid == 7 ? wd2 :
445         mbid == 0 ? wd1 :
446         mbid == 1 ? wd0 :
447         0;
448
449     assign      inD2 =
450         mbid == 7 ? wd3 :
451         mbid == 0 ? wd2 :
452         mbid == 1 ? wd1 :
453         mbid == 2 ? wd0 :
454         0;
455
456     assign      inD3 =
457         mbid == 0 ? wd3 :
458         mbid == 1 ? wd2 :
459         mbid == 2 ? wd1 :
460         mbid == 3 ? wd0 :
461         0;
462
463     assign      inD4 =
464         mbid == 1 ? wd3 :
465         mbid == 2 ? wd2 :
466         mbid == 3 ? wd1 :
467         mbid == 4 ? wd0 :
468         0;
469
470     assign      inD5 =
471         mbid == 2 ? wd3 :
472         mbid == 3 ? wd2 :
473         mbid == 4 ? wd1 :
474         mbid == 5 ? wd0 :
475         0;
476
477     assign      inD6 =
478         mbid == 3 ? wd3 :
479         mbid == 4 ? wd2 :
480         mbid == 5 ? wd1 :
481         mbid == 6 ? wd0 :
482         0;
483
484     assign      inD7 =
485         mbid == 4 ? wd3 :
486         mbid == 5 ? wd2 :
487         mbid == 6 ? wd1 :
488         mbid == 7 ? wd0 :
489         0;
490
491     // Which banks get written
492     assign      dwEn0 = wenable & (mbid <= 0 | mbid >= 5);
493     assign      dwEn1 = wenable & (mbid <= 1 | mbid >= 6);
```

```

494     assign          dwEn2 = wenable & (mbid <= 2 | mbid >= 7);
495     assign          dwEn3 = wenable & (mbid <= 3);
496     assign          dwEn4 = wenable & (mbid >= 1 & mbid <= 4);
497     assign          dwEn5 = wenable & (mbid >= 2 & mbid <= 5);
498     assign          dwEn6 = wenable & (mbid >= 3 & mbid <= 6);
499     assign          dwEn7 = wenable & (mbid >= 4);
500
501 endmodule
502
503
504 // Combinational blocks
505
506 // Fetch stage
507
508 // Split instruction byte into icode and ifun fields
509 module split(itype, icode, ifun);
510     input  [7:0] itype;
511     output [3:0] icode;
512     output [3:0] ifun;
513
514     assign          icode = itype[7:4];
515     assign          ifun  = itype[3:0];
516 endmodule
517
518 // Extract immediate word from 5 bytes of instruction
519 module align(itypes, need_regids, rA, rB, valC);
520     input [39:0] itypes;
521     input          need_regids;
522     output [3:0]  rA;
523     output [3:0]  rB;
524     output [31:0] valC;
525     assign          rA = itypes[7:4];
526     assign          rB = itypes[3:0];
527     assign          valC = need_regids ? itypes[39:8] : itypes[31:0];
528 endmodule
529
530 // PC incrementer
531 module pc_increment(pc, need_regids, need_valC, valP);
532     input [31:0] pc;
533     input          need_regids;
534     input          need_valC;
535     output [31:0] valP;
536     assign          valP = pc + 1 + 4*need_valC + need_regids;
537 endmodule
538
539 // Execute Stage
540
541 // ALU
542 module alu(aluA, aluB, alufun, valE, new_cc);
543     input [31:0] aluA, aluB; // Data inputs

```

```

544 input [3:0]  alufun;          // ALU function
545 output [31:0] valE;          // Data Output
546 output [2:0] new_cc;         // New values for ZF, SF, OF
547
548 parameter    ALUADD = 4'h0;
549 parameter    ALUSUB = 4'h1;
550 parameter    ALUAND = 4'h2;
551 parameter    ALUXOR = 4'h3;
552
553 assign       valE =
554             alufun == ALUSUB ? aluB - aluA :
555             alufun == ALUAND ? aluB & aluA :
556             alufun == ALUXOR ? aluB ^ aluA :
557             aluB + aluA;
558 assign       new_cc[2] = (valE == 0); // ZF
559 assign       new_cc[1] = valE[31];   // SF
560 assign       new_cc[0] =
561             alufun == ALUADD ?
562             (aluA[31] == aluB[31]) & (aluA[31] != valE[31]) :
563             alufun == ALUSUB ?
564             (~aluA[31] == aluB[31]) & (aluB[31] != valE[31]) :
565             0;
566 endmodule
567
568
569 // Condition code register
570 module cc(cc, new_cc, set_cc, reset, clock);
571 output[2:0] cc;
572 input [2:0] new_cc;
573 input      set_cc;
574 input      reset;
575 input      clock;
576
577 cenrreg #(3) c(cc, new_cc, set_cc, reset, 3'b100, clock);
578 endmodule
579
580 // branch condition logic
581 module cond(ifun, cc, Cnd);
582 input [3:0] ifun;
583 input [2:0] cc;
584 output      Cnd;
585
586 wire       zf = cc[2];
587 wire       sf = cc[1];
588 wire       of = cc[0];
589
590 // Jump & move conditions.
591 parameter  C_YES  = 4'h0;
592 parameter  C_LE   = 4'h1;
593 parameter  C_L    = 4'h2;

```

```

594 parameter C_E = 4'h3;
595 parameter C_NE = 4'h4;
596 parameter C_GE = 4'h5;
597 parameter C_G = 4'h6;
598
599 assign Cnd =
600 (ifun == C_YES) | //
601 (ifun == C_LE & ((sf^of)|zf)) | // <=
602 (ifun == C_L & (sf^of)) | // <
603 (ifun == C_E & zf) | // ==
604 (ifun == C_NE & ~zf) | // !=
605 (ifun == C_GE & (~sf^of)) | // >=
606 (ifun == C_G & (~sf^of)&~zf); // >
607
608 endmodule
609
610 // -----
611 // Processor implementation
612 // -----
613
614
615 // The processor can run in 5 different modes:
616 // RUN: Normal operation
617 // RESET: Sets PC to 0, clears all pipe registers;
618 // Initializes condition codes
619 // DOWNLOAD: Download bytes from controller into memory
620 // UPLOAD: Upload bytes from memory to controller
621 // STATUS: Upload other status information to controller
622
623 // Processor module
624 module processor(mode, udaddr, idata, odata, stat, clock);
625 input [2:0] mode; // Signal operating mode to processor
626 input [31:0] udaddr; // Upload/download address
627 input [31:0] idata; // Download data word
628 output [31:0] odata; // Upload data word
629 output [2:0] stat; // Status
630 input clock; // Clock input
631
632 // Define modes
633 parameter RUN_MODE = 0; // Normal operation
634 parameter RESET_MODE = 1; // Resetting processor;
635 parameter DOWNLOAD_MODE = 2; // Transferring to memory
636 parameter UPLOAD_MODE = 3; // Reading from memory
637 // Uploading register & other status information
638 parameter STATUS_MODE = 4;
639
640 // Constant values
641
642 // Instruction codes
643 parameter IHALT = 4'h0;

```

```
644 parameter INOP = 4'h1;
645 parameter IRRMOVL = 4'h2;
646 parameter IIRMOVL = 4'h3;
647 parameter IRMMOVL = 4'h4;
648 parameter IMRMOVL = 4'h5;
649 parameter IOPL = 4'h6;
650 parameter IJXX = 4'h7;
651 parameter ICALL = 4'h8;
652 parameter IRET = 4'h9;
653 parameter IPUSHL = 4'hA;
654 parameter IPOPL = 4'hB;
655 parameter IIADDL = 4'hC;
656 parameter ILEAVE = 4'hD;
657 parameter IPOP2 = 4'hE;
658
659 // Function codes
660 parameter FNONE = 4'h0;
661
662 // Jump conditions
663 parameter UNCOND = 4'h0;
664
665 // Register IDs
666 parameter RESP = 4'h4;
667 parameter REBP = 4'h5;
668 parameter RNONE = 4'hF;
669
670 // ALU operations
671 parameter ALUADD = 4'h0;
672
673 // Status conditions
674 parameter SBUB = 3'h0;
675 parameter SAOK = 3'h1;
676 parameter SHLT = 3'h2;
677 parameter SADR = 3'h3;
678 parameter SINS = 3'h4;
679 parameter SPIP = 3'h5;
680
681 // Fetch stage signals
682 wire [31:0] f_predPC, F_predPC, f_pc;
683 wire f_ok;
684 wire imem_error;
685 wire [2:0] f_stat;
686 wire [47:0] f_instr;
687 wire [3:0] imem_icode;
688 wire [3:0] imem_ifun;
689 wire [3:0] f_icode;
690 wire [3:0] f_ifun;
691 wire [3:0] f_rA;
692 wire [3:0] f_rB;
693 wire [31:0] f_valC;
```

```

694 wire [31:0] f_valP;
695 wire      need_regids;
696 wire      need_valC;
697 wire      instr_valid;
698 wire      F_stall, F_bubble;
699
700 // Decode stage signals
701 wire [2:0] D_stat;
702 wire [31:0] D_pc;
703 wire [3:0] D_icode;
704 wire [3:0] D_ifun;
705 wire [3:0] D_rA;
706 wire [3:0] D_rB;
707 wire [31:0] D_valC;
708 wire [31:0] D_valP;
709
710 wire [31:0] d_valA;
711 wire [31:0] d_valB;
712 wire [31:0] d_rvalA;
713 wire [31:0] d_rvalB;
714 wire [3:0] d_dstE;
715 wire [3:0] d_dstM;
716 wire [3:0] d_srcA;
717 wire [3:0] d_srcB;
718 wire      D_stall, D_bubble;
719
720 // Execute stage signals
721 wire [2:0] E_stat;
722 wire [31:0] E_pc;
723 wire [3:0] E_icode;
724 wire [3:0] E_ifun;
725 wire [31:0] E_valC;
726 wire [31:0] E_valA;
727 wire [31:0] E_valB;
728 wire [3:0] E_dstE;
729 wire [3:0] E_dstM;
730 wire [3:0] E_srcA;
731 wire [3:0] E_srcB;
732
733 wire [31:0] aluA;
734 wire [31:0] aluB;
735 wire      set_cc;
736 wire [2:0] cc;
737 wire [2:0] new_cc;
738 wire [3:0] alufun;
739 wire      e_Cnd;
740 wire [31:0] e_valE;
741 wire [31:0] e_valA;
742 wire [3:0] e_dstE;
743 wire      E_stall, E_bubble;

```

```

744
745 // Memory stage
746 wire [2:0] M_stat;
747 wire [31:0] M_pc;
748 wire [3:0] M_icode;
749 wire [3:0] M_ifun;
750 wire M_Cnd;
751 wire [31:0] M_valE;
752 wire [31:0] M_valA;
753 wire [3:0] M_dstE;
754 wire [3:0] M_dstM;
755
756 wire [2:0] m_stat;
757 wire [31:0] mem_addr;
758 wire [31:0] mem_data;
759 wire mem_read;
760 wire mem_write;
761 wire [31:0] m_valM;
762 wire M_stall, M_bubble;
763 wire m_ok;
764
765 // Write-back stage
766 wire [2:0] W_stat;
767 wire [31:0] W_pc;
768 wire [3:0] W_icode;
769 wire [31:0] W_valE;
770 wire [31:0] W_valM;
771 wire [3:0] W_dstE;
772 wire [3:0] W_dstM;
773 wire [31:0] w_valE;
774 wire [31:0] w_valM;
775 wire [3:0] w_dstE;
776 wire [3:0] w_dstM;
777 wire W_stall, W_bubble;
778
779 // Global status
780 wire [2:0] Stat;
781
782 // Debugging logic
783 wire [31:0] eax, ecx, edx, ebx, esp, ebp, esi, edi;
784 wire zf = cc[2];
785 wire sf = cc[1];
786 wire of = cc[0];
787
788 // Control signals
789 wire resetting = (mode == RESET_MODE);
790 wire uploading = (mode == UPLOAD_MODE);
791 wire downloading = (mode == DOWNLOAD_MODE);
792 wire running = (mode == RUN_MODE);
793 wire getting_info = (mode == STATUS_MODE);

```

```

794 // Logic to control resetting of pipeline registers
795 wire F_reset = F_bubble | resetting;
796 wire D_reset = D_bubble | resetting;
797 wire E_reset = E_bubble | resetting;
798 wire M_reset = M_bubble | resetting;
799 wire W_reset = W_bubble | resetting;
800
801 // Processor status
802 assign stat = Stat;
803 // Output data
804 assign odata =
805 // When getting status, get either register or special status value
806 getting_info ?
807 (udaddr == 0 ? eax :
808  udaddr == 4 ? ecx :
809  udaddr == 8 ? edx :
810  udaddr == 12 ? ebx :
811  udaddr == 16 ? esp :
812  udaddr == 20 ? ebp :
813  udaddr == 24 ? esi :
814  udaddr == 28 ? edi :
815  udaddr == 32 ? cc :
816  udaddr == 36 ? W_pc : 0)
817 : m_valM;
818
819 // Pipeline registers
820 // All implemented with module
821 //   preg(out, in, stall, bubble, bubbleval, clock)
822
823 // All pipeline registers are implemented with module
824 //   preg(out, in, stall, bubble, bubbleval, clock)
825 // F Register
826 preg #(32) F_predPC_reg(F_predPC, f_predPC, F_stall, F_reset, 0, clock);
827 // D Register
828 preg #(3) D_stat_reg(D_stat, f_stat, D_stall, D_reset, SBUB, clock);
829 preg #(32) D_pc_reg(D_pc, f_pc, D_stall, D_reset, 0, clock);
830 preg #(4) D_icode_reg(D_icode, f_icode, D_stall, D_reset, INOP, clock);
831 preg #(4) D_ifun_reg(D_ifun, f_ifun, D_stall, D_reset, FNONE, clock);
832 preg #(4) D_rA_reg(D_rA, f_rA, D_stall, D_reset, RNONE, clock);
833 preg #(4) D_rB_reg(D_rB, f_rB, D_stall, D_reset, RNONE, clock);
834 preg #(32) D_valC_reg(D_valC, f_valC, D_stall, D_reset, 0, clock);
835 preg #(32) D_valP_reg(D_valP, f_valP, D_stall, D_reset, 0, clock);
836 // E Register
837 preg #(3) E_stat_reg(E_stat, D_stat, E_stall, E_reset, SBUB, clock);
838 preg #(32) E_pc_reg(E_pc, D_pc, E_stall, E_reset, 0, clock);
839 preg #(4) E_icode_reg(E_icode, D_icode, E_stall, E_reset, INOP, clock);
840 preg #(4) E_ifun_reg(E_ifun, D_ifun, E_stall, E_reset, FNONE, clock);
841 preg #(32) E_valC_reg(E_valC, D_valC, E_stall, E_reset, 0, clock);
842 preg #(32) E_valA_reg(E_valA, d_valA, E_stall, E_reset, 0, clock);
843 preg #(32) E_valB_reg(E_valB, d_valB, E_stall, E_reset, 0, clock);

```

```

844  preg #(4)  E_dstE_reg(E_dstE, d_dstE, E_stall, E_reset, RNONE, clock);
845  preg #(4)  E_dstM_reg(E_dstM, d_dstM, E_stall, E_reset, RNONE, clock);
846  preg #(4)  E_srcA_reg(E_srcA, d_srcA, E_stall, E_reset, RNONE, clock);
847  preg #(4)  E_srcB_reg(E_srcB, d_srcB, E_stall, E_reset, RNONE, clock);
848  // M Register
849  preg #(3)  M_stat_reg(M_stat, E_stat, M_stall, M_reset, SBUB, clock);
850  preg #(32) M_pc_reg(M_pc, E_pc, M_stall, M_reset, 0, clock);
851  preg #(4)  M_icode_reg(M_icode, E_icode, M_stall, M_reset, INOP, clock);
852  preg #(4)  M_ifun_reg(M_ifun, E_ifun, M_stall, M_reset, FNONE, clock);
853  preg #(1)  M_Cnd_reg(M_Cnd, e_Cnd, M_stall, M_reset, 0, clock);
854  preg #(32) M_valE_reg(M_valE, e_valE, M_stall, M_reset, 0, clock);
855  preg #(32) M_valA_reg(M_valA, e_valA, M_stall, M_reset, 0, clock);
856  preg #(4)  M_dstE_reg(M_dstE, e_dstE, M_stall, M_reset, RNONE, clock);
857  preg #(4)  M_dstM_reg(M_dstM, E_dstM, M_stall, M_reset, RNONE, clock);
858  // W Register
859  preg #(3)  W_stat_reg(W_stat, m_stat, W_stall, W_reset, SBUB, clock);
860  preg #(32) W_pc_reg(W_pc, M_pc, W_stall, W_reset, 0, clock);
861  preg #(4)  W_icode_reg(W_icode, M_icode, W_stall, W_reset, INOP, clock);
862  preg #(32) W_valE_reg(W_valE, M_valE, W_stall, W_reset, 0, clock);
863  preg #(32) W_valM_reg(W_valM, m_valM, W_stall, W_reset, 0, clock);
864  preg #(4)  W_dstE_reg(W_dstE, M_dstE, W_stall, W_reset, RNONE, clock);
865  preg #(4)  W_dstM_reg(W_dstM, M_dstM, W_stall, W_reset, RNONE, clock);
866
867  // Fetch stage logic
868  split split(f_instr[7:0], imem_icode, imem_ifun);
869  align align(f_instr[47:8], need_regids, f_rA, f_rB, f_valC);
870  pc_increment pci(f_pc, need_regids, need_valC, f_valP);
871
872  // Decode stage
873  regfile regf(w_dstE, w_valE, w_dstM, w_valM,
874             d_srcA, d_rvalA, d_srcB, d_rvalB, resetting, clock,
875             eax, ecx, edx, ebx, esp, ebp, esi, edi);
876
877  // Execute stage
878  alu alu(aluA, aluB, alufun, e_valE, new_cc);
879  cc  ccreg(cc, new_cc,
880         // Only update CC when everything is running normally
881         running & set_cc,
882         resetting, clock);
883  cond cond_check(E_ifun, cc, e_Cnd);
884
885  // Memory stage
886  bmemory m(
887     // Only update memory when everything is running normally
888     // or when downloading
889     (downloading | uploading) ? uaddr : mem_addr, // Read/Write address
890     (running & mem_write) | downloading, // When to write to memory
891     downloading ? idata : M_valA, // Write data
892     (running & mem_read) | uploading, // When to read memory
893     m_valM, // Read data

```

```

894     m_ok,
895     f_pc, f_instr, f_ok, clock);           // Instruction memory access
896
897     assign imem_error = ~f_ok;
898     assign dmem_error = ~m_ok;
899
900 // Write-back stage logic
901
902 // Control logic
903 // -----
904 // The following code is generated from the HCL description of the
905 // pipeline control using the hcl2v program
906 // -----
907 assign f_pc =
908     ((M_icode == IJXX) & ~M_Cnd) ? M_valA : (W_icode == IRET) ? W_valM :
909     F_predPC);
910
911 assign f_icode =
912     (imem_error ? INOP : imem_icode);
913
914 assign f_ifun =
915     (imem_error ? FNONE : imem_ifun);
916
917 assign instr_valid =
918     (f_icode == INOP | f_icode == IHALT | f_icode == IRRMOVL | f_icode ==
919     IIRMOVL | f_icode == IRMMOVL | f_icode == IMRMOVL | f_icode == IOPL
920     | f_icode == IJXX | f_icode == ICALL | f_icode == IRET | f_icode ==
921     IPUSHL | f_icode == IPOPL);
922
923 assign f_stat =
924     (imem_error ? SADR : ~instr_valid ? SINS : (f_icode == IHALT) ? SHLT :
925     SAOK);
926
927 assign need_regids =
928     (f_icode == IRRMOVL | f_icode == IOPL | f_icode == IPUSHL | f_icode ==
929     IPOPL | f_icode == IIRMOVL | f_icode == IRMMOVL | f_icode == IMRMOVL)
930     ;
931
932 assign need_valC =
933     (f_icode == IIRMOVL | f_icode == IRMMOVL | f_icode == IMRMOVL | f_icode
934     == IJXX | f_icode == ICALL);
935
936 assign f_predPC =
937     ((f_icode == IJXX | f_icode == ICALL) ? f_valC : f_valP);
938
939 assign d_srcA =
940     ((D_icode == IRRMOVL | D_icode == IRMMOVL | D_icode == IOPL | D_icode
941     == IPUSHL) ? D_rA : (D_icode == IPOPL | D_icode == IRET) ? RESP :
942     RNONE);
943

```

```

944 assign d_srcB =
945     ((D_icode == IOPL | D_icode == IRMMOVL | D_icode == IMRMOVL) ? D_rB : (
946         D_icode == IPUSHL | D_icode == IPOPL | D_icode == ICALL | D_icode
947         == IRET) ? RESP : RNONE);
948
949 assign d_dstE =
950     ((D_icode == IRRMOVL | D_icode == IIRMOVL | D_icode == IOPL) ? D_rB : (
951         D_icode == IPUSHL | D_icode == IPOPL | D_icode == ICALL | D_icode
952         == IRET) ? RESP : RNONE);
953
954 assign d_dstM =
955     ((D_icode == IMRMOVL | D_icode == IPOPL) ? D_rA : RNONE);
956
957 assign d_valA =
958     ((D_icode == ICALL | D_icode == IJXX) ? D_valP : (d_srcA == e_dstE) ?
959         e_valE : (d_srcA == M_dstM) ? m_valM : (d_srcA == M_dstE) ? M_valE :
960         (d_srcA == W_dstM) ? W_valM : (d_srcA == W_dstE) ? W_valE : d_rvalA);
961
962 assign d_valB =
963     ((d_srcB == e_dstE) ? e_valE : (d_srcB == M_dstM) ? m_valM : (d_srcB
964         == M_dstE) ? M_valE : (d_srcB == W_dstM) ? W_valM : (d_srcB ==
965         W_dstE) ? W_valE : d_rvalB);
966
967 assign aluA =
968     ((E_icode == IRRMOVL | E_icode == IOPL) ? E_valA : (E_icode == IIRMOVL
969         | E_icode == IRMMOVL | E_icode == IMRMOVL) ? E_valC : (E_icode ==
970         ICALL | E_icode == IPUSHL) ? -4 : (E_icode == IRET | E_icode == IPOPL
971         ) ? 4 : 0);
972
973 assign aluB =
974     ((E_icode == IRMMOVL | E_icode == IMRMOVL | E_icode == IOPL | E_icode
975         == ICALL | E_icode == IPUSHL | E_icode == IRET | E_icode == IPOPL)
976         ? E_valB : (E_icode == IRRMOVL | E_icode == IIRMOVL) ? 0 : 0);
977
978 assign alufun =
979     ((E_icode == IOPL) ? E_ifun : ALUADD);
980
981 assign set_cc =
982     (((E_icode == IOPL) & ~(m_stat == SADR | m_stat == SINS | m_stat ==
983         SHLT)) & ~(W_stat == SADR | W_stat == SINS | W_stat == SHLT));
984
985 assign e_valA =
986     E_valA;
987
988 assign e_dstE =
989     (((E_icode == IRRMOVL) & ~e_Cnd) ? RNONE : E_dstE);
990
991 assign mem_addr =
992     ((M_icode == IRMMOVL | M_icode == IPUSHL | M_icode == ICALL | M_icode
993         == IMRMOVL) ? M_valE : (M_icode == IPOPL | M_icode == IRET) ?

```

```

994     M_valA : 0);
995
996 assign mem_read =
997     (M_icode == IMRMOVL | M_icode == IPOPL | M_icode == IRET);
998
999 assign mem_write =
1000     (M_icode == IRMMOVL | M_icode == IPUSHL | M_icode == ICALL);
1001
1002 assign m_stat =
1003     (dmem_error ? SADR : M_stat);
1004
1005 assign w_dstE =
1006     W_dstE;
1007
1008 assign w_valE =
1009     W_valE;
1010
1011 assign w_dstM =
1012     W_dstM;
1013
1014 assign w_valM =
1015     W_valM;
1016
1017 assign Stat =
1018     ((W_stat == SBUB) ? SAOK : W_stat);
1019
1020 assign F_bubble =
1021     0;
1022
1023 assign F_stall =
1024     (((E_icode == IMRMOVL | E_icode == IPOPL) & (E_dstM == d_srcA | E_dstM
1025         == d_srcB)) | (IRET == D_icode | IRET == E_icode | IRET ==
1026         M_icode));
1027
1028 assign D_stall =
1029     ((E_icode == IMRMOVL | E_icode == IPOPL) & (E_dstM == d_srcA | E_dstM
1030         == d_srcB));
1031
1032 assign D_bubble =
1033     (((E_icode == IJXX) & ~e_Cnd) | (~((E_icode == IMRMOVL | E_icode ==
1034         IPOPL) & (E_dstM == d_srcA | E_dstM == d_srcB)) & (IRET ==
1035         D_icode | IRET == E_icode | IRET == M_icode)));
1036
1037 assign E_stall =
1038     0;
1039
1040 assign E_bubble =
1041     (((E_icode == IJXX) & ~e_Cnd) | ((E_icode == IMRMOVL | E_icode == IPOPL
1042         ) & (E_dstM == d_srcA | E_dstM == d_srcB)));
1043

```

```
1044 assign M_stall =
1045     0;
1046
1047 assign M_bubble =
1048     ((m_stat == SADR | m_stat == SINS | m_stat == SHLT) | (W_stat == SADR
1049         | W_stat == SINS | W_stat == SHLT));
1050
1051 assign W_stall =
1052     (W_stat == SADR | W_stat == SINS | W_stat == SHLT);
1053
1054 assign W_bubble =
1055     0;
1056
1057 // -----
1058 // End of code generated by hcl2v
1059 // -----
1060 endmodule
1061
```

References

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

Index

CS:APP2e , 1

conditional expression (Verilog), 5

continuous assignment (Verilog), 4

Design Compiler, 2

Goel, Prabhu, 1

memory bank, Y86 processor, 8

module, Verilog, 4

Moorby, Philip, 1

multiplexor, 5

non-blocking assignment, Verilog, 7

reg, Verilog state-holding signal, 7

Synopsys, Inc., 2

wire, Verilog combinational signal, 2, 7, 14