7. Sample Models: The DP32 Processor

This chapter contains an extended example, a description of a hypothetical processor called the DP32. The processor instruction set and bus architectures are first described, and then a behavioural description is given. A test bench model is constructed, and the model checked with a small test program. Next, the processor is decomposed into components at the register transfer level. A number of components are described, and a structural description of the processor is constructed using these components. The same test bench is used, but this time with the structural architecture.

7.1. Instruction Set Architecture

The DP32 is a 32-bit processor with a simple instruction set. It has a number of registers, shown in Figure 7-1. There are 256 general purpose registers (R0–R255), a program counter (PC) and a condition code register (CC). The general purpose registers are addressable by software, whereas the PC and CC registers are not.

On reset, the PC is initialised to zero, and all other registers are undefined. By convention, R0 is read-only and contains zero. This is not enforced by hardware, and the zero value must be loaded by software after reset.

The memory accessible to the DP32 consists of 32-bit words, addressed by a 32-bit word-address. Instructions are all multiples of 32-bit words, and are stored in this memory. The PC register contains the address of the next instruction to be executed. After each instruction word is fetched, the PC is incremented by one to point to the next word.

The three CC register bits are updated after each arithmetic or logical instruction. The Z (zero) bit is set if the result is zero. The N (negative) bit is set if the result of an arithmetic instruction is negative, and is undefined after logical instructions. The V(overflow) bit is set if the result of an arithmetic instruction exceeds the bounds of representable integers, and is

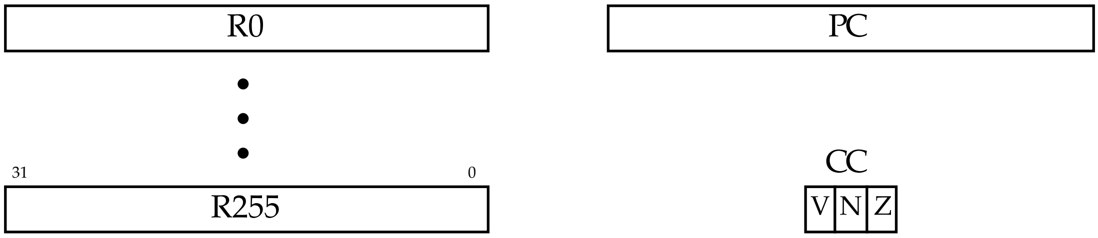

Figure 7-1. DP32 registers.

Instruction	Name	Function	opcode
Add	add	$r3 \leftarrow r1 + r2$	X"00"
Sub	subtract	$r3 \leftarrow r1 - r2$	X"01"
Mul	multiply	$r3 \leftarrow r1 \times r2$	X"02"
Div	divide	$r3 \leftarrow r1 \div r2$	X"03"
Addq	add quick	$r3 \leftarrow r1 + i8$	X"10"
Subq	subtract quick	$r3 \leftarrow r1 - i8$	X"11"
Mulq	multiply quick	$r3 \leftarrow r1 \times i8$	X"12"
Divq	divide quick	$r3 \leftarrow r1 \div i8$	X"13"
Land	logical and	$r3 \leftarrow r1$ & $r2$	X"04"
Lor	logical or	$r3 \leftarrow r1 \mid r2$	X"05"
Lxor	logical exclusive or	$r3 \leftarrow r1 \oplus r2$	X"06"
Lmask	logical mask	$r3 \leftarrow r1$ & ~$r2$	X"07"

Table 7-1. DP32 arithmetic and logic instructions.

undefined after logical instructions.

The DP32 instruction set is divided into a number of encoding formats. Firstly, arithmetic and logical instructions are all one 32-bit word long, formatted as follows:

	31 — 24	23 — 16	15 — 8	7 — 0
(Addr):	op	r3	r1	r2/i8

The op field is the op-code, r3 is the destination register address, r1 and r2 are source register addresses, and i8 is an immediate two-compliment integer operand. The arithmetic and logical instructions are listed in Table 7-1.

Memory load and store instructions have two formats, depending on whether a long or short displacement value is used. The format for a long displacement is:

	31 — 24	23 — 16	15 — 8	7 — 0
(Addr):	op	r3	r1	*ignored*
(Addr+1):	disp			

The format for a short displacement is:

	31 — 24	23 — 16	15 — 8	7 — 0
(Addr):	op	r3	r1	i8

The op field is the op-code, r3 specifies the register to be loaded or stored, r1 is used as an index register, disp is a long immediate displacement, and i8 is a short immediate displacement. The load and store instructions are listed in Table 7-2.

Instruction	Name	Function	opcode
Ld	load	r3 ← M[r1 + disp32]	X“20”
St	store	M[r1 + disp32] ← r3	X“21”
Ldq	load quick	r3 ← M[r1 + i8]	X“30”
Stq	store quick	M[r1 + i8] ← r3	X“31”

Table 7-2. DP32 load and store instructions.

Instruction	Name	Function	opcode
Br-*ivnz*	branch	if *cond* then PC ← PC + disp32	X“40”
Brq-*ivnz*	branch quick	if *cond* then PC ← PC + i8	X“51”
Bi-*ivnz*	branch indexed	if *cond* then PC ← r1 + disp32	X“41”
Biq-*ivnz*	branch indexed quick	if *cond* then PC ← r1 + i8	X“51”

Table 7-3. DP32 load and store instructions.

Finally, there are four branch instructions, listed in Table 7-3, each with a slightly different format. The format of the ordinary brach is:

	31–24	23–20	19–16	15–8	7–0
(Addr):	op	*xxxx*	ivnz	*xxxx*	*xxxx*
(Addr+1):	disp				

The format of a quick branch is:

	31–24	23–20	19–16	15–8	7–0
(Addr):	op	*xxxx*	ivnz	*xxxx*	i8

The format of an indexed branch

	31–24	23–20	19–16	15–8	7–0
(Addr):	op	*xxxx*	ivnz	r1	*xxxx*
(Addr+1):	disp				

The format of a quick indexed branch

	31–24	23–20	19–16	15–8	7–0
(Addr):	op	*xxxx*	ivnz	r1	i8

The op field is the op-code, disp is a long immediate displacement, i8 is a short immediate displacement, r1 is used as an index register, and ivnz is a the condition mask. The branch is taken if

$$cond \equiv ((V \ \& \ v) \mid (N \ \& \ n) \mid (Z \ \& \ z)) = i.$$

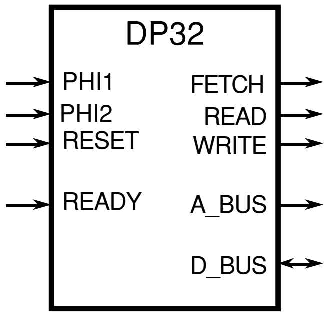

Figure 7-2. DP32 port diagram.

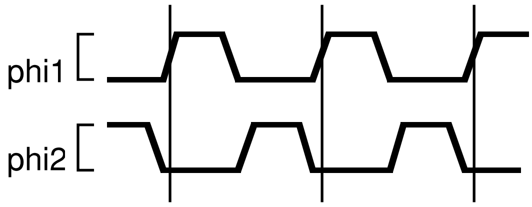

Figure 7-3. DP32 clock waveforms.

7.2. Bus Architecture

The DP32 processor communicates with its memory over synchronous 32-bit address and data buses. The external ports of the DP32 are shown in Figure 7-2.

The two clock inputs, phi1 and phi2, provide a two-phase non-overlapping clock for the processor. The clock waveforms are shown in Figure 7-3. Each cycle of the phi1 clock defines a bus state, one of Ti (idle), T1 or T2. Bus transactions consist of a T1 state followed by one or more T2 states, with Ti states between transactions.

The port a_bus is a 32-bit address bus, and d_bus is a 32-bit bidirection data bus. The read and write ports control bus read and write transactions. The fetch port is a status signal indicating that a bus read in progress is an instruction fetch. The ready input is used by a memory device to indicate that read data is available or write data has been accepted.

The timing for a bus read transaction is show in Figure 7-4. During an idle state, Ti, the processor places the memory address on the address bus to start the transaction. The next state is a T1 state. After the leading edge of the phi1 clock, the processor asserts the read control signal, indicating that the address is valid and the memory should start the read transaction. The processor also asserts the fetch signal if it is reading instructions. It always leaves the write signal negated during read transactions. During the T1 state and the following T2 state, the memory accesses the requested data, and places it on the data bus. If it has completed the data access by the end of the T2 state, it asserts ready. The processor accepts the data, and completes the transaction. On the other hand, if the memory has not yet supplied the data by the end of the T2 state, it leaves ready false. The processor then repeats T2 states until it detects ready true. By this means, a slow memory can extend the transaction until it has read the data. At the end of the transaction, the processor returns its control outputs to their default values, and the memory negates r eady and removes the data from the data bus. The processor continues with idle states until the next transaction is required.

The timing for a bus write transaction is show in Figure 7-5. Here also, the transaction starts with the processor placing the address on the address bus during a Ti state. After the leading edge of phi1 during the subsequent T1 state, the processor negates fetch and asserts write. The read signal remains false for the whole transaction. During the T1 state, the processor also makes the data to be written available on the data bus. The memory

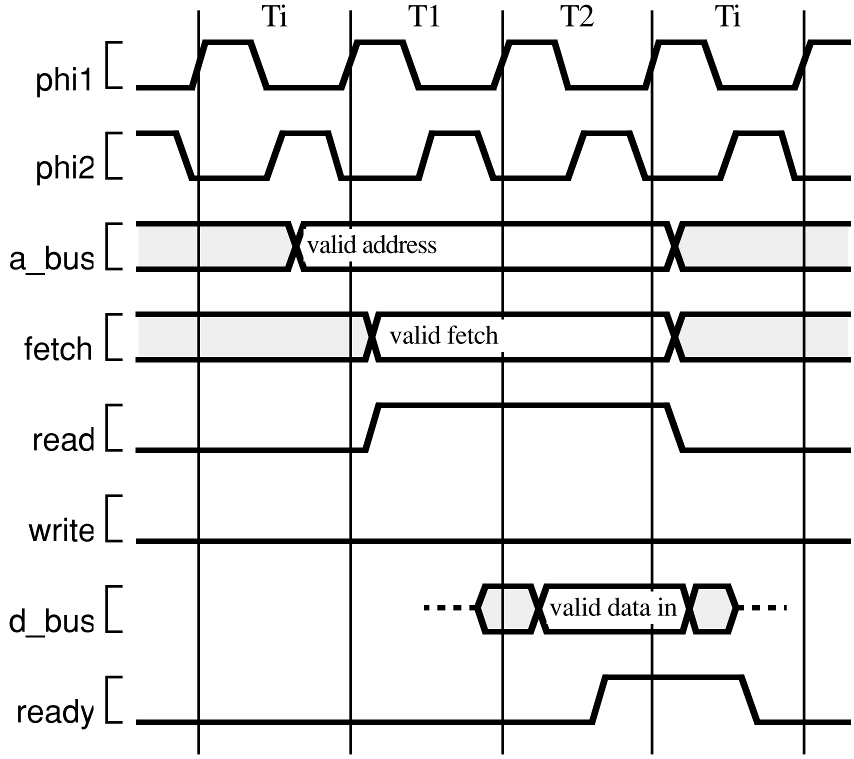

Figure 7-4. DP32 bus read transaction.

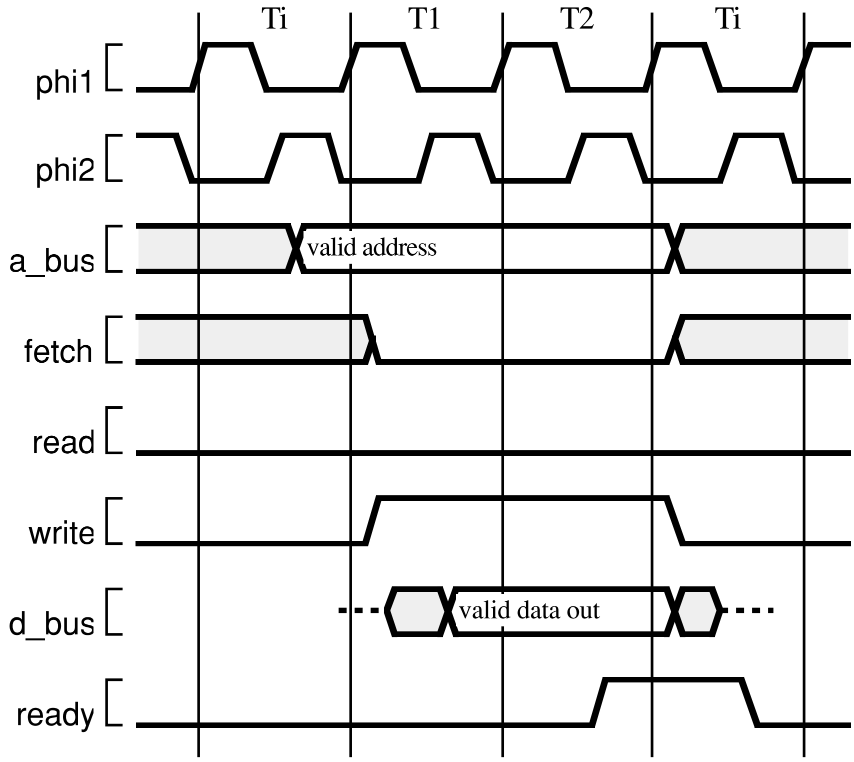

Figure 7-5. DP32 bus write transaction.

can accept this data during the T1 and subsequent T2 states. If it has completed the write by the end of the T2 state, it asserts ready. The processor then completes the transaction and continutes with Ti states, and the memory removes the data from the data bus and negates ready. If the memory has not had time to complete the write by the end of the T2 state, it leaves ready false. The processor will then repeat T2 states until it detects ready true.

7.3. Types and Entity

We start the description of the DP32 processor by defining a package containing the data types to be used in the model, and some useful operations on those types. The package declaration of dp32_types is listed in Figure 7-6.

```
package dp32_types is

    constant unit_delay : Time := 1 ns;

    type bool_to_bit_table is array (boolean) of bit;
    constant bool_to_bit : bool_to_bit_table;

    subtype bit_32 is bit_vector(31 downto 0);
    type bit_32_array is array (integer range <>) of bit_32;
    function resolve_bit_32 (driver : in bit_32_array) return bit_32;
    subtype bus_bit_32 is resolve_bit_32 bit_32;

    subtype bit_8 is bit_vector(7 downto 0);

    subtype CC_bits is bit_vector(2 downto 0);
    subtype cm_bits is bit_vector(3 downto 0);

    constant op_add :       bit_8 := X"00";
    constant op_sub :       bit_8 := X"01";
    constant op_mul :       bit_8 := X"02";
    constant op_div :       bit_8 := X"03";
    constant op_addq :      bit_8 := X"10";
    constant op_subq :      bit_8 := X"11";
    constant op_mulq :      bit_8 := X"12";
    constant op_divq :      bit_8 := X"13";
    constant op_land :      bit_8 := X"04";
    constant op_lor :       bit_8 := X"05";
    constant op_lxor :      bit_8 := X"06";
    constant op_lmask :     bit_8 := X"07";
    constant op_ld :        bit_8 := X"20";
    constant op_st :        bit_8 := X"21";
    constant op_ldq :       bit_8 := X"30";
    constant op_stq :       bit_8 := X"31";
    constant op_br :        bit_8 := X"40";
    constant op_brq :       bit_8 := X"50";
    constant op_bi :        bit_8 := X"41";
    constant op_biq :       bit_8 := X"51";

    function bits_to_int (bits : in bit_vector) return integer;
    function bits_to_natural (bits : in bit_vector) return natural;
    procedure int_to_bits (int : in integer; bits : out bit_vector);

end dp32_types;
```

Figure 7-6. Package declaration for dp32_types.

The constant unit_delay is used as the default delay time through-out the DP32 description. This approach is common when writing models to describe the function of a digital system, before developing a detailed timing model.

The constant bool_to_bit is a lookup table for converting between boolean conditions and the type bit. Examples of its use will be seen later. Note that it is a deferred constant, so its value will be given in the package body.

The next declarations define the basic 32-bit word used in the DP32 model. The function resolve_bit_32 is a resolution function used to determine the value on a 32-bit bus with multiple drivers. Such a bus is declared with the subtype bus_bit_32, a resolved type.

The subtype bit_8 is part of a 32-bit word used as an op-code or register address. CC_bits is the type for condition codes, and cm_bits is the type for the condition mask in a branch op-code.

The next set of constant declarations define the op-code bit patterns for valid op-codes. These symbolic names are used as a matter of good coding style, enabling the op-code values to be changed without having to modify the model code in numerous places.

Finally, a collection of conversion functions between bit-vector values and numeric values is defined. The bodies for these subprograms are hidden in the package body.

The body of the dp32_types package is listed in Figure 7-7. Firstly the value for the deferred constant bool_to_bit is given: false translates to '0' and true translates to '1'. An example of the use of this table is:

```
flag_bit <= bool_to_bit(flag_condition);
```

Next, the body of the resolution function for 32-bit buses is defined. The function takes as its parameter an unconstrained array of bit_32 values, and produces as a result the bit-wide logical-or of the values. Note that the function cannot assume that the length of the array will be greater than one. If no drivers are active on the bus, an empty array will be passed to the resolution function. In this case, the default value of all '0' bits (float_value) is used as the result.

```
package body dp32_types is

    constant bool_to_bit : bool_to_bit_table :=
        (false => '0', true => '1');

    function resolve_bit_32 (driver : in bit_32_array) return bit_32 is

        constant float_value : bit_32 := X"0000_0000";
        variable result : bit_32 := float_value;

    begin
        for i in driver'range loop
            result := result or driver(i);
        end loop;
        return result;
    end resolve_bit_32;
```

Figure 7-7. Package body for dp32_types.

The function bits_to_int converts a bit vector representing a twos-compliment signed integer into an integer type value. The local variable temp is declared to be a bit vector of the same size and index range as the parameter bits. The variable result is initialised to zero when the function is invoked, and subsequently used to accumulate the weighted bit values in

```
    function bits_to_int (bits : in bit_vector) return integer is

        variable temp : bit_vector(bits'range);
        variable result : integer := 0;

    begin
        if bits(bits'left) = '1' then          -- negative number
            temp := not bits;
        else
            temp := bits;
        end if;
        for index in bits'range loop           -- sign bit of temp = '0'
            result := result * 2 + bit'pos(temp(index));
        end loop;
        if bits(bits'left) = '1' then
            result := (-result) - 1;
        end if;
        return result;
    end bits_to_int;

    function bits_to_natural (bits : in bit_vector) return natural is

        variable result : natural := 0;

    begin
        for index in bits'range loop
            result := result * 2 + bit'pos(bits(index));
        end loop;
        return result;
    end bits_to_natural;

    procedure int_to_bits (int : in integer; bits : out bit_vector) is

        variable temp : integer;
        variable result : bit_vector(bits'range);

    begin
        if int < 0 then
            temp := -(int+1);
        else
            temp := int;
        end if;
        for index in bits'reverse_range loop
            result(index) := bit'val(temp rem 2);
            temp := temp / 2;
        end loop;
        if int < 0 then
            result := not result;
            result(bits'left) := '1';
        end if;
        bits := result;
    end int_to_bits;

end dp32_types;
```

Figure 7-7 (continued).

```
use work.dp32_types.all;
entity dp32 is
    generic (Tpd : Time := unit_delay);
    port (d_bus : inout bus_bit_32 bus;
            a_bus : out bit_32;
            read, write : out bit;
            fetch : out bit;
            ready : in bit;
            phi1, phi2 : in bit;
            reset : in bit);
end dp32;
```

Figure 7-8. Entity declaration for dp32.

the for loop. The function bits_to_natural performs a similar function to bits_to_int, but does not need to do any special processing for negative numbers. Finally, the function int_to_bits performs the inverse of bits_to_int.

The entity declaration of the DP32 processor is shown in Figure 7-8. The library unit is preceded by a use clause referencing all the items in the package dp32_types. The entity has a generic constant Tpd used to specify the propagation delays between input events and output signal changes. The default value is the unit delay specified in the dp32_types package. There are a number of ports corresponding to those shown in Figure 7-2. The reset, clocks, and bus control signals are represented by values of type bit. The address bus output is a simple bit-vector type, as the processor is the only module driving that bus. On the other hand, the data bus is a resolved bit-vector type, as it may be driven by both the processor and a memory module. The word **bus** in the port declaration indicates that all drivers for the data bus may be disconnected at the same time (ie, none of them is driving the bus).

7.4. Behavioural Description

In this section a behavioural model of the DP32 processor will be presented. This model can be used to run test programs in the DP32 instruction set by connecting it to a simulated memory model. The architecture body for the behavioural description is listed in Figure 7-9.

The declaration section for the architecture body contains the declaration for the DP32 register file type, and array of 32-bit words, indexed by a natural number constrained to be in the range 0 to 255.

The architecture body contains only one concurrent statement, namely an anonymous process which implements the behaviour as a sequential algorithm. This process declares a number of variables which represent the internal state of the processor: the register file (reg), the program counter (PC), and the current instruction register (current_instr). A number of working variables and aliases are also declared.

The procedure memory_read implements the behavioural model of a memory read transaction. The parameters are the memory address to read from, a flag indicating whether the read is an instruction fetch, and a result parameter returning the data read. The procedure refers to the entity ports, which are visible because they are declared in the parent of the procedure.

The memory_read model firstly drives the address and fetch bit ports, and then waits until the next leading edge of phi1, indicating the start of the next clock cycle. (The wait statement is sensitive to a change from '0' to '1' on phi1.) When that event occurs, the model checks the state of the reset input port, and if it is set, immediately returns without further action. If reset is clear, the model starts a T1 state by asserting the read bit port a propagation delay time after the clock edge. It then waits again until the next phi1 leading edge, indicating the start of the next clock cycle. Again, it checks reset and discontinues if reset is set. The model then starts a loop executing T2 states. It waits until phi2 changes from '1' to '0' (at the end of the cycle), and then checks reset again, returning if it is set. Otherwise it checks the ready bit input port, and if set, accepts the data from the data bus port and exits the loop. If ready is not set, the loop repeats, adding another T2 state to the transaction. After the loop, the model waits for the next clock edge indicating the start of the Ti state at the end of the transaction. After checking reset again, the model clears ready to complete the transaction, and returns to the parent process.

The procedure memory_write is similar, implementing the model for a memory write transaction. The parameters are simply the memory address to write to, and the data to write. The model similarly has reset checks after each wait point. One difference is that at the end of the transaction, there is a null signal assignment to the data bus port. This models the bahaviour of the processor disconnecting from the data bus, that is, at this point it stops driving the port.

```
use work.dp32_types.all;

architecture behaviour of dp32 is

    subtype reg_addr is natural range 0 to 255;
    type reg_array is array (reg_addr) of bit_32;

begin -- behaviour of dp32

    process

        variable reg : reg_array;
        variable PC : bit_32;
        variable current_instr : bit_32;
        variable op: bit_8;
        variable r3, r1, r2 : reg_addr;
        variable i8 : integer;
        alias cm_i : bit is current_instr(19);
        alias cm_V : bit is current_instr(18);
        alias cm_N : bit is current_instr(17);
        alias cm_Z : bit is current_instr(16);
        variable cc_V, cc_N, cc_Z : bit;
        variable temp_V, temp_N, temp_Z : bit;
        variable displacement, effective_addr : bit_32;
```

Figure 7-9. Behavioural architecture body for dp32.

```
procedure memory_read (addr : in bit_32;
                       fetch_cycle : in boolean;
                       result : out bit_32) is
begin
    -- start bus cycle with address output
    a_bus <= addr after Tpd;
    fetch <= bool_to_bit(fetch_cycle) after Tpd;
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    --
    -- T1 phase
    --
    read <= '1' after Tpd;
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    --
    -- T2 phase
    --
    loop
        wait until phi2 = '0';
        if reset = '1' then
            return;
        end if;
        -- end of T2
        if ready = '1' then
            result := d_bus;
            exit;
        end if;
    end loop;
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    --
    -- Ti phase at end of cycle
    --
    read <= '0' after Tpd;
end memory_read;
```

Figure 7-9 (continued).

```
procedure memory_write (addr : in bit_32;
                        data : in bit_32) is
begin
    -- start bus cycle with address output
    a_bus <= addr after Tpd;
    fetch <= '0' after Tpd;
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    --
    -- T1 phase
    --
    write <= '1' after Tpd;
    wait until phi2 = '1';
    d_bus <= data after Tpd;
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    --
    -- T2 phase
    --
    loop
        wait until phi2 = '0';
        if reset = '1' then
            return;
        end if;
        -- end of T2
        exit when ready = '1';
    end loop;
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    --
    -- Ti phase at end of cycle
    --
    write <= '0' after Tpd;
    d_bus <= null after Tpd;
end memory_write;
```

Figure 7-9 (continued).

The next four procedures, add, subtract, multiply and divide, implement the arithmetic operations on 32-bit words representing twos-complement signed integers. They each take two integer operands, and produce a 32-bit word result and the three condition code flags V (overflow), N (negative) and Z (zero). The result parameter is of mode **inout** because the test for negative and zero results read its value after it has been written. Each procedure is carefully coded to avoid causing an integer overflow on the host machine executing the model (assuming that machine uses 32-bit integers). The add and subtract procedures wrap around if overflow occurs, and multiply and divide return the largest or smallest integer.

Following these procedures is the body of the process which implements the DP32 behavioural model. This process is activated during the initialisation phase of a simulation. It consists of three sections which are repeated sequentially: reset processing, instruction fetch, and instruction execution.

```
procedure add (result : inout bit_32;
                    op1, op2 : in integer;
                    V, N, Z : out bit) is
begin
    if op2 > 0 and op1 > integer'high-op2 then       -- positive overflow
        int_to_bits(((integer'low+op1)+op2)-integer'high-1, result);
        V := '1';
    elsif op2 < 0 and op1 < integer'low-op2 then  -- negative overflow
        int_to_bits(((integer'high+op1)+op2)-integer'low+1, result);
        V := '1';
    else
        int_to_bits(op1 + op2, result);
        V := '0';
    end if;
    N := result(31);
    Z := bool_to_bit(result = X"0000_0000");
end add;

procedure subtract (result : inout bit_32;
                         op1, op2 : in integer;
                         V, N, Z : out bit) is
begin
    if op2 < 0 and op1 > integer'high+op2 then       -- positive overflow
        int_to_bits(((integer'low+op1)-op2)-integer'high-1, result);
        V := '1';
    elsif op2 > 0 and op1 < integer'low+op2 then  -- negative overflow
        int_to_bits(((integer'high+op1)-op2)-integer'low+1, result);
        V := '1';
    else
        int_to_bits(op1 - op2, result);
        V := '0';
    end if;
    N := result(31);
    Z := bool_to_bit(result = X"0000_0000");
end subtract;
```

Figure 7-9 (continued).

When the reset input is asserted, all of the control ports are returned to their initial states, the data bus driver is disconnected, and the PC register is cleared. The model then waits until reset is negated before proceeding. Throughout the rest of the model, the reset input is checked after each bus transaction. If the transaction was aborted by reset being asserted, no further action is taken in fetching or executing an instruction, and control falls through to the reset handling code.

The instruction fetch part is simply a call to the memory read procedure. The PC register is used to provide the address, the fetch flag is true, and the result is returned into the current instruction register. The PC register is then incremented by one using the arithmetic procedure previously defined.

The fetched instruction is next decoded into its component parts: the op-code, the source and destination register addresses and an immediate constant field. The op-code is then used as the selector for a case statement

```
procedure multiply (result : inout bit_32;
                    op1, op2 : in integer;
                    V, N, Z : out bit) is
begin
    if ((op1>0 and op2>0) or (op1<0 and op2<0))  -- result positive
            and (abs op1 > integer'high / abs op2) then     -- positive overflow
        int_to_bits(integer'high, result);
        V := '1';
    elsif ((op1>0 and op2<0) or (op1<0 and op2>0))  -- result negative
            and ((- abs op1) < integer'low / abs op2) then   -- negative overflow
        int_to_bits(integer'low, result);
        V := '1';
    else
        int_to_bits(op1 * op2, result);
        V := '0';
    end if;
    N := result(31);
    Z := bool_to_bit(result = X"0000_0000");
end multiply;

procedure divide (result : inout bit_32;
                  op1, op2 : in integer;
                  V, N, Z : out bit) is
begin
    if op2=0 then
        if op1>=0 then              -- positive overflow
            int_to_bits(integer'high, result);
        else
            int_to_bits(integer'low, result);
        end if;
        V := '1';
    else
        int_to_bits(op1 / op2, result);
        V := '0';
    end if;
    N := result(31);
    Z := bool_to_bit(result = X"0000_0000");
end divide;
```

Figure 7-9 (continued).

which codes the instruction execution. For the arithmetic instructions (including the quick forms), the arithmetic procedures previously defined are invoked. For the logical instructions, the register bit-vector values are used in VHDL logical expressions to determine the bit-vector result. The condition code Z flag is set if the result is a bit-vector of all '0' bits.

The model executes a load instruction by firstly reading the displacement from memory and incrementing the PC register. The displacement is added to the value of the index register to form the effective address. This is then used in a memory read to load the data into the result register. A quick load is executed similarly, except that no memory read is needed to fetch the displacement; the variable i8 decoded from the instruction is used. The store and quick store instructions parallel the load instructions, with the memory data read being replaced by a memory data write.

Execution of a branch instruction starts with a memory read to fetch the displacement, and an add to increment the PC register by one. The displacement is added to the value of the PC register to form the effective address. Next, the condition expression is evaluated, comparing the condition code bits with the condition mask in the instruction, to determine whether the branch is taken. If it is, the PC register takes on the effective address value. The branch indexed instruction is similar, with the index register value replacing the PC value to form the effective address. The quick branch forms are also similar, with the immediate constant being used for the displacement instead of a value fetched from memory.

```
begin
    --
    -- check for reset active
    --
    if reset = '1' then
        read <= '0' after Tpd;
        write <= '0' after Tpd;
        fetch <= '0' after Tpd;
        d_bus <= null after Tpd;
        PC := X"0000_0000";
        wait until reset = '0';
    end  if;
    --
    -- fetch next instruction
    --
    memory_read(PC, true, current_instr);
    if reset /= '1' then
        add(PC, bits_to_int(PC), 1, temp_V, temp_N, temp_Z);
        --
        -- decode & execute
        --
        op := current_instr(31 downto 24);
        r3 := bits_to_natural(current_instr(23 downto 16));
        r1 := bits_to_natural(current_instr(15 downto 8));
        r2 := bits_to_natural(current_instr(7 downto 0));
        i8 := bits_to_int(current_instr(7 downto 0));
```

Figure 7-9 (continued).

```
case op is
    when op_add =>
        add(reg(r3), bits_to_int(reg(r1)), bits_to_int(reg(r2)),
                cc_V, cc_N, cc_Z);
    when op_addq =>
        add(reg(r3), bits_to_int(reg(r1)), i8, cc_V, cc_N, cc_Z);
    when op_sub =>
        subtract(reg(r3), bits_to_int(reg(r1)), bits_to_int(reg(r2)),
                cc_V, cc_N, cc_Z);
    when op_subq =>
        subtract(reg(r3), bits_to_int(reg(r1)), i8, cc_V, cc_N, cc_Z);
    when op_mul =>
        multiply(reg(r3), bits_to_int(reg(r1)), bits_to_int(reg(r2)),
                cc_V, cc_N, cc_Z);
    when op_mulq =>
        multiply(reg(r3), bits_to_int(reg(r1)), i8, cc_V, cc_N, cc_Z);
    when op_div =>
        divide(reg(r3), bits_to_int(reg(r1)), bits_to_int(reg(r2)),
                cc_V, cc_N, cc_Z);
    when op_divq =>
        divide(reg(r3), bits_to_int(reg(r1)), i8, cc_V, cc_N, cc_Z);
    when op_land =>
        reg(r3) := reg(r1) and reg(r2);
        cc_Z := bool_to_bit(reg(r3) = X"0000_0000");
    when op_lor =>
        reg(r3) := reg(r1) or reg(r2);
        cc_Z := bool_to_bit(reg(r3) = X"0000_0000");
    when op_lxor =>
        reg(r3) := reg(r1) xor reg(r2);
        cc_Z := bool_to_bit(reg(r3) = X"0000_0000");
    when op_lmask =>
        reg(r3) := reg(r1) and not reg(r2);
        cc_Z := bool_to_bit(reg(r3) = X"0000_0000");
    when op_ld =>
        memory_read(PC, true, displacement);
        if reset /= '1' then
            add(PC, bits_to_int(PC), 1, temp_V, temp_N, temp_Z);
            add(effective_addr,
                    bits_to_int(reg(r1)), bits_to_int(displacement),
                    temp_V, temp_N, temp_Z);
            memory_read(effective_addr, false, reg(r3));
        end if;
    when op_ldq =>
        add(effective_addr,
                bits_to_int(reg(r1)), i8,
                temp_V, temp_N, temp_Z);
        memory_read(effective_addr, false, reg(r3));
    when op_st =>
        memory_read(PC, true, displacement);
        if reset /= '1' then
            add(PC, bits_to_int(PC), 1, temp_V, temp_N, temp_Z);
            add(effective_addr,
                    bits_to_int(reg(r1)), bits_to_int(displacement),
                    temp_V, temp_N, temp_Z);
            memory_write(effective_addr, reg(r3));
        end if;
```

Figure 7-9 (continued).

```
                when op_stq =>
                    add(effective_addr,
                            bits_to_int(reg(r1)), i8,
                            temp_V, temp_N, temp_Z);
                    memory_write(effective_addr, reg(r3));
                when op_br =>
                    memory_read(PC, true, displacement);
                    if reset /= '1' then
                        add(PC, bits_to_int(PC), 1, temp_V, temp_N, temp_Z);
                        add(effective_addr,
                                bits_to_int(PC), bits_to_int(displacement),
                                temp_V, temp_N, temp_Z);
                        if ((cm_V and cc_V) or (cm_N and cc_N) or (cm_Z and cc_Z))
                                = cm_i then
                            PC := effective_addr;
                        end if;
                    end if;
                when op_bi =>
                    memory_read(PC, true, displacement);
                    if reset /= '1' then
                        add(PC, bits_to_int(PC), 1, temp_V, temp_N, temp_Z);
                        add(effective_addr,
                                bits_to_int(reg(r1)), bits_to_int(displacement),
                                temp_V, temp_N, temp_Z);
                        if ((cm_V and cc_V) or (cm_N and cc_N) or (cm_Z and cc_Z))
                                = cm_i then
                            PC := effective_addr;
                        end if;
                    end if;
                when op_brq =>
                    add(effective_addr,
                            bits_to_int(PC), i8,
                            temp_V, temp_N, temp_Z);
                    if ((cm_V and cc_V) or (cm_N and cc_N) or (cm_Z and cc_Z))
                            = cm_i then
                        PC := effective_addr;
                    end if;
                when op_biq =>
                    add(effective_addr,
                            bits_to_int(reg(r1)), i8,
                            temp_V, temp_N, temp_Z);
                    if ((cm_V and cc_V) or (cm_N and cc_N) or (cm_Z and cc_Z))
                            = cm_i then
                        PC := effective_addr;
                    end if;
                when others =>
                    assert false report "illegal instruction" severity warning;
            end case;
        end if;     -- reset /= '1'
    end process;

end behaviour;
```

Figure 7-9 (continued).

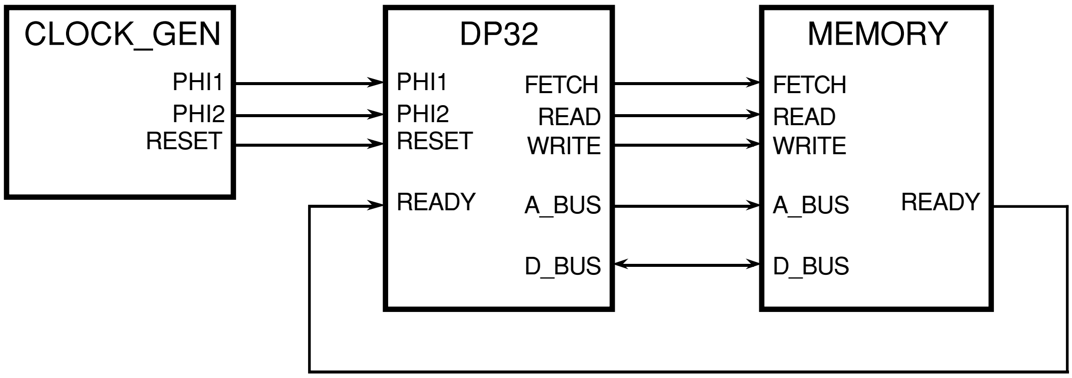

Figure 7-10. Test bench circuit for DP32.

```
use work.dp32_types.all;

entity clock_gen is
    generic (Tpw : Time;                    -- clock pulse width
             Tps : Time);                   -- pulse separation between phases
    port (phi1, phi2 : out bit;
          reset : out bit);
end clock_gen;

architecture behaviour of clock_gen is

    constant clock_period : Time := 2*(Tpw+Tps);

begin

    reset_driver :
        reset <= '1', '0' after 2*clock_period+Tpw;

    clock_driver : process
    begin
        phi1 <=  '1', '0' after Tpw;
        phi2 <= '1' after Tpw+Tps, '0' after Tpw+Tps+Tpw;
        wait for clock_period;
    end process clock_driver;

end behaviour;
```

Figure 7-11. Description of clock_gen driver.

7.5. Test Bench

One way of testing the behavioural model of the DP32 processor is to connect it in a test bench circuit, shown in Figure 7-10. The clock_gen component generates the two-phase clock and the reset signal to drive the processor. The memory stores a test program and data. We write behavioural models for these two components, and connect them in a structural description of the test bench.

Figure 7-11 lists the entity declaration and behavioural architecture of the clock generator. The clock_gen entity has two formal generic constants. Tpw is the pulse width for each of phi1 and phi2, that is, the time for which each clock is '1'. Tps is the pulse separation, that is, the time between one clock signal changing to '0' and the other clock signal changing to '1'.

Based on these values, the clock period is twice the sum of the pulse width and the separation.

The architecture of the clock generator consists of two concurrent statements, one to drive the `reset` signal and the other to drive the clock signals. The reset driver schedules a '1' value on `reset` when it is activated at simulation initialisation, followed by a '0' a little after two clock periods later. This concurrent statement is never subsequently reactivated, since its waveform list does not refer to any signals. The clock driver process, when activated, schedules a pulse on phi1 immediately, followed by a pulse on phi2, and then suspends for a clock period. When it resumes, it repeats, scheduling the next clock cycle.

The entity declaration and behavioural architecture of the memory module are shown in Figure 7-12. The architecture body consists of one process to implement the behaviour. The process contains an array variable to represent the storage of the memory. When the process is activated, it places the output ports in an initial state: the data bus disconnected and the ready bit negated. It then waits for either a read or write command. When one of these occurs, the address is sampled and converted from a bit-vector to a number. If it is within the address bounds of the memory, the command is acted upon.

For a write command, the ready bit is asserted after a delay representing the write access time of the memory, and then the model waits until the end of the write cycle. At that time, the value on the data bus from a propagation delay beforehand is sampled and written into the memory array. The use of this delayed value models the fact that memory devices actually store the data that was valid a setup-time before the triggering edge of the command bit.

For a read command, the data from the memory array is accessed and placed on the data bus after a delay. This delay represents the read access time of the memory. The ready bit is also asserted after the delay, indicating that the processor may continue. The memory then waits until the end of the read cycle.

At the end of a memory cycle, the process repeats, setting the data bus and ready bit drivers to their initial state, and waiting for the next command.

Figure 7-13 shows the entity declaration and structural architecture of the test bench circuit. The entity contains no ports, since there are no external connections to the test bench. The architecture body contains component declarations for the clock driver, the memory and the processor. The ports in these component declarations correspond exactly to those of the entity declarations. There are no formal generic constants, so the actuals for the generics in the entity declarations will be specified in a configuration. The architecture body next declares the signals which are used to connect the components together. These signals may be traced by a simulation monitor when the simulation is run. The concurrent statements of the architecture body consist of the three component instances.

```
use work.dp32_types.all;

entity memory is
    generic (Tpd : Time := unit_delay);
    port (d_bus : inout bus_bit_32 bus;
            a_bus : in bit_32;
            read, write : in bit;
            ready : out bit);
end memory;

architecture behaviour of memory is
begin

    process

        constant low_address : integer := 0;
        constant high_address : integer := 65535;
        type memory_array is
            array (integer range low_address to high_address) of bit_32;
        variable mem : memory_array;
        variable address : integer;

    begin
        --
        -- put d_bus and reply into initial state
        --
        d_bus <= null after Tpd;
        ready <= '0' after Tpd;
        --
        -- wait for a command
        --
        wait until (read = '1') or (write = '1');
        --
        -- dispatch read or write cycle
        --
        address := bits_to_int(a_bus);
        if address >= low_address and address <= high_address then
            -- address match for this memory
            if write = '1' then
                ready <= '1' after Tpd;
                wait until write = '0';              -- wait until end of write cycle
                mem(address) := d_bus'delayed(Tpd);  -- sample data from Tpd ago
            else -- read = '1'
                d_bus <= mem(address) after Tpd;     -- fetch data
                ready <= '1' after Tpd;
                wait until read = '0';               -- hold for read cycle
            end if;
        end if;
    end process;

end behaviour;
```

Figure 7-12. Description of memory module.

```
use work.dp32_types.all;

entity dp32_test is
end dp32_test;

architecture structure of dp32_test is

    component clock_gen
        port (phi1, phi2 : out bit;
                reset : out bit);
    end component;

    component dp32
        port (d_bus : inout bus_bit_32 bus;
                a_bus : out bit_32;
                read, write : out bit;
                fetch : out bit;
                ready : in bit;
                phi1, phi2 : in bit;
                reset : in bit);
    end component;

    component memory
        port (d_bus : inout bus_bit_32 bus;
                    a_bus : in bit_32;
                    read, write : in bit;
                    ready : out bit);
    end component;

    signal d_bus : bus_bit_32 bus;
    signal a_bus : bit_32;
    signal read, write : bit;
    signal fetch : bit;
    signal ready : bit;
    signal phi1, phi2 : bit;
    signal reset : bit;

begin

    cg : clock_gen
        port map (phi1 => phi1, phi2 => phi2, reset => reset);

    proc : dp32
        port map (d_bus => d_bus, a_bus => a_bus,
                    read => read, write => write, fetch => fetch,
                    ready => ready,
                    phi1 => phi1, phi2 => phi2, reset => reset);

    mem : memory
        port map (d_bus => d_bus, a_bus => a_bus,
                    read => read, write => write, ready => ready);

end structure;
```

Figure 7-13. Description of test bench circuit.

```
configuration dp32_behaviour_test of dp32_test is
    for structure
        for cg : clock_gen
            use entity work.clock_gen(behaviour)
                generic map (Tpw => 8 ns, Tps => 2 ns);
        end for;
        for mem : memory
            use entity work.memory(behaviour);
        end for;
        for proc : dp32
            use entity work.dp32(behaviour);
        end for;
    end for;
end dp32_behaviour_test;
```

Figure 7-14. Configuration of test bench using behaviour of DP32.

Lastly, a configuration for the test bench, using the behavioural description of the DP32 processor, is listed in Figure 7-14. The configuration specifies that each of the components in the structure architecture of the test bench should use the behaviour architecture of the corresponding entity. Actual generic constants are specified for the clock generator, giving a clock period of 20 ns. The default values for the generic constants of the other entities are used.

In order to run the test bench model, a simulation monitor is invoked and a test program loaded into the array variable in the memory model. The author used the Zycad System VHDL™ simulation system for this purpose. Figure 7-15 is an extract from the listing produced by an assembler created for the DP32 processor. The test program initializes R0 to zero (the assembler macro `i nitr0` generates an `l mask` instruction), and then loops incrementing a counter in memory. The values in parentheses are the instruction addresses, and the hexadecimal values in square brackets are the assembled instructions.

™ Zycad System VHDL is a trademark of Zycad Corporation.

```
                                include dp32.inc $

                                !!!   conventions:
                                !!!       r0 = 0
                                !!!       r1 scratch

                                begin
(      0) [07000000         ]     initr0
                                start:
(      1) [10020000         ]     addq(r2, r0, 0)    ! r2 := 0
                                loop:
(      2) [21020000 00000008]     sta(r2, counter)   ! counter := r2
(      4) [10020201         ]     addq(r2, r2, 1)    ! increment r2
(      5) [1101020A         ]     subq(r1, r2, 10)   ! if r2 = 10 then
(      6) [500900FA         ]     brzq(start)        !   restart
(      7) [500000FA         ]     braq(loop)         ! else next loop

                                counter:
(      8) [00000000         ]     data(0)
                                end
```

Figure 7-15. Assembler listing of a test program.

7.6. Register Transfer Architecture

The previous descriptions of the DP32 specified its behaviour without reference to the internal structure of the processor. Such a description is invaluable, as it allows the computer architect to evaluate the instruction set and compare it with alternatives before commiting expensive resources to detailed design and implementation.

Once this abstract architecture has been settled on, the next level of architecture can be designed. Figure 7-16 is a block diagram of a simple architecture to implement the DP32 instrcuction set. (Most control signals are not shown.) It consists mainly of a collection of registers and an arithmetic and logic unit (ALU), connected by a number of buses. There are also buffers for interfacing to the processor-memory bus, and a control unit for sequencing operation of the processor.

The software addressable registers are implemented using a three-port register file. Ports 1 and 2 supply source operands onto the op1 and op2 buses respectively. The address for port 2 is normally taken from the r2 field of the current instruction, but a multiplexor is included to allow the r3 field to be used when a store instruction is executed. The op1 and op2 buses

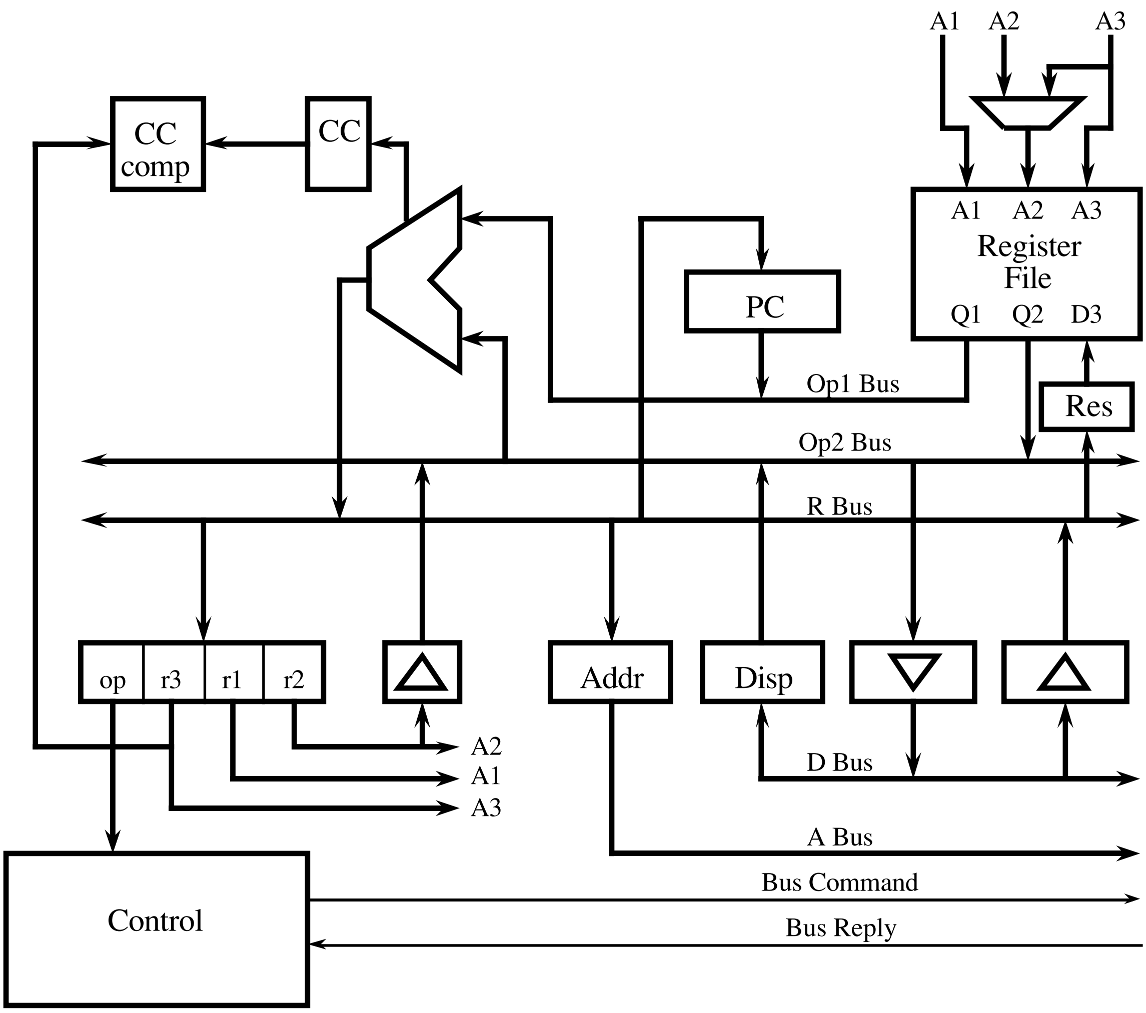

Figure 7-16. DP32 data paths block diagram.

are connected to the ALU inputs, and the ALU output drives the result bus. The result can be latched for writing back to the register file using port 3. The program counter (PC) register also supplies the op1 bus, and can be loaded from the result bus. The ALU condition flags are latched into the condition code (CC) register, and from there can be compared with the condition mask from the current instruction. The memory bus interface includes an address latch to drive the address bus, a data output buffer driven from the op2 bus, a data input buffer driving the result bus, and a displacement latch driving the op2 bus. An instruction fetched from memory is stored in current instruction register. The r1, r2 and r3 fields are used as register file addresses. The r2 field is also used as an immediate constant and may be sign extended onto the op2 bus. Four bits from the r3 field are used as the condition mask, and the opcode field is used by the control unit.

In this section, descriptions will be given for each of the sub-modules in this architecture, and then they will be used in a structural architecture body of the DP32 entity.

7.6.1. Multiplexor

An entity declaration and architecture body for a 2-input multiplexor is listed in Figure 7-17. The entity has a select input bit, two bit-vector inputs i0 and i1, and a bit-vector output y. The size of the bit-vector ports is determined by the generic constant width, which must be specified when the entity is used in a structural description. The architecture body contains a concurrent selected signal assignment, which uses the value of the select input to determine which of the two bit-vector inputs is passed through to the output. The assignment is sensitive to all of the input signals, so when any of them changes, the assignment will be resumed.

7.6.2. Transparent Latch

An entity declaration and architecture body for a latch is listed in Figure 7-18. The entity has an enable input bit, a bit-vector input d, and a bit-vector output q. The size of the bit-vector ports is determined by the generic constant width, which must be specified when the entity is used in a structural description. The architecture body contains a process which is

```
use work.dp32_types.all;

entity mux2 is
    generic (width : positive;
                   Tpd : Time := unit_delay);
    port (i0, i1 : in bit_vector(width-1 downto 0);
             y : out bit_vector(width-1 downto 0);
             sel : in bit);
end mux2;

architecture behaviour of mux2 is
begin
    with sel select
        y <=   i0 after Tpd when '0',
                   i1 after Tpd when '1';
end behaviour;
```

Figure 7-17. Description of 2-input multiplexor.

```
use work.dp32_types.all;

entity latch is
    generic (width : positive;
             Tpd : Time := unit_delay);
    port (d : in bit_vector(width-1 downto 0);
          q : out bit_vector(width-1 downto 0);
          en : in bit);
end latch;

architecture behaviour of latch is

begin

    process (d, en)
    begin
        if en = '1' then
            q <= d after Tpd;
        end if;
    end process;

end behaviour;
```

Figure 7-18. Description of a transparent latch.

sensitive to the d and en inputs. The behaviour of the latch is such that when en is '1', changes on d are transmitted through to q. However, when en changes to '0', any new value on d is ignored, and the current value on q is maintained. In the model shown in Figure 7-18, the latch storage is provided by the output port, in that if no new value is assigned to it, the current value does not change.

7.6.3. Buffer

An entity declaration and architecture body for a buffer is listed in Figure 7-19. The entity has an enable input bit en, a bit-vector input a, and a resolved bit-vector bus output b. It is not possible to make this entity generic with respect to input and output port width, because of a limitation imposed by the VHDL language semantics. The output port needs to be a resolved signal, so a bus resolution function is specified in the definition of the port type. This function takes a parameter which is an unconstrained array. In order to make the buffer port width generic, we would need to specify a bus resolution function which took as a parameter an unconstrained array of bit-vector elements whose length is not known. VHDL does not allow the element type of an unconstrained array to be an unconstrained array, so this approach is not possible. For this reason, we define a buffer entity with fixed port widths of 32 bits.

The behaviour of the buffer is implemented by a process sensitive to the en and a inputs. If en is '1', the a input is transmitted through to the b output. If en is '0', the driver for b is disconnected, and the value on a is ignored.

```
use work.dp32_types.all;

entity buffer_32 is
    generic (Tpd : Time := unit_delay);
    port (a : in bit_32;
            b : out bus_bit_32 bus;
            en : in bit);
end buffer_32;

architecture behaviour of buffer_32 is

begin

    b_driver: process (en, a)
    begin
        if en = '1' then
            b <= a after Tpd;
        else
            b <= null after Tpd;
        end if;
    end process b_driver;

end behaviour;
```

Figure 7-19. Description of a buffer.

```
use work.dp32_types.all;

entity signext_8_32 is
    generic (Tpd : Time := unit_delay);
    port (a : in bit_8;
            b : out bus_bit_32 bus;
            en : in bit);
end signext_8_32;

architecture behaviour of signext_8_32 is

begin

    b_driver: process (en, a)
    begin
        if en = '1' then
            b(7 downto 0) <= a after Tpd;
            if a(7) = '1' then
                b(31 downto 8) <= X"FFFF_FF" after Tpd;
            else
                b(31 downto 8) <= X"0000_00" after Tpd;
            end if;
        else
            b <= null after Tpd;
        end if;
    end process b_driver;

end behaviour;
```

Figure 7-20. Description of the sign extending buffer.

7.6.4. Sign Extending Buffer

The sign-extending buffer shown in Figure 7-20 is almost identical to the plain buffer, except that it has an 8-bit input. This input is treated as a twos-complement signed integer, and the output is the same integer, but extended to 32 bits. The extension is achieved by replicating the sign bit into bits 8 to 31 of the output.

7.6.5. Latching Buffer

Figure 7-21 lists an entity declaration an architecture body for a latching buffer. This model is a combination of those for the plain latch and buffer. When latch_en is '1', changes on d are stored in the latch, and may be transmitted through to q. However, when latch_en changes to '0', any new value on d is ignored, and the currently stored value is maintained. The out_en input controls whether the stored value is tranmitted to the output. Unlike the plain latch, explicit storage must be provided (in the form of the variable latched_value), since the output driver may be disconnected when a new value is to be stored.

7.6.6. Program Counter Register

The entity declaration and architecture body of the PC register are listed in Figure 7-22. The PC register is a master/slave type register, which can be reset to all zeros by asserting the reset input. When reset is negated, the latch operates normally. With latch_en at '1', the value of the d input is stored in the variable master_PC, but the output (if enabled) is driven from the previously stored value in slave_PC. Then when latch_en changes from

```
use work.dp32_types.all;

entity latch_buffer_32 is
    generic (Tpd : Time := unit_delay);
    port (d : in bit_32;
            q : out bus_bit_32 bus;
            latch_en : in bit;
            out_en : in bit);
end latch_buffer_32;

architecture behaviour of latch_buffer_32 is

begin

    process (d, latch_en, out_en)
        variable latched_value : bit_32;
    begin
        if latch_en = '1' then
            latched_value := d;
        end if;
        if out_en = '1' then
            q <= latched_value after Tpd;
        else
            q <= null after Tpd;
        end if;
    end process;

end behaviour;
```

Figure 7-21. Description of a latching buffer.

```
use work.dp32_types.all;

entity PC_reg is
    generic (Tpd : Time := unit_delay);
    port (d : in bit_32;
            q : out bus_bit_32 bus;
            latch_en : in bit;
            out_en : in bit;
            reset : in bit);
end PC_reg;

architecture behaviour of PC_reg is

begin

    process (d, latch_en, out_en, reset)
        variable master_PC, slave_PC : bit_32;
    begin
        if reset = '1' then
            slave_PC := X"0000_0000";
        elsif latch_en = '1' then
            master_PC := d;
        else
            slave_PC := master_PC;
        end if;
        if out_en = '1' then
            q <= slave_PC after Tpd;
        else
            q <= null after Tpd;
        end if;
    end process;

end behaviour;
```

Figure 7-22. Description of the PC register.

'1' to '0', the slave value is update from the master value, and any subsequent changes in the d input are ignored. This behaviour means that the PC register output can be used to derive a new value, and the new value written back at the same time. If an ordinary transparent latch were used, a race condition would be created, since the new value would be transmitted through to the output in place of the old value, affecting the calculation of the new value.

7.6.7. Register File

Figure 7-23 lists the description of the 3-port register file, with two read ports and one write port. Each port has an address input (a1, a2 and a3) and an enable input (en1, en2 and en3). The read ports have data bus outputs (q1 and q2), and the write port has a data input (d3). The number bits in the port addresses is determined by the generic constant depth. The behaviour of the entity is implemented by the process reg_file. It declares a numeric type used to index the register file, and an array for the register file storage. When any of the inputs change, firstly the write port enable is checked, and if asserted, the addressed register is updated. Then each of the read port enables is checked. If asserted, the addressed data is fetched and driven onto the corresponding data output bus. If the port is disabled, the data output bus driver is disconnected.

```
use work.dp32_types.all;

entity reg_file_32_rrw is
    generic (depth : positive;              -- number of address bits
             Tpd : Time := unit_delay;
             Tac : Time := unit_delay);
    port (a1 : in bit_vector(depth-1 downto 0);
          q1 : out bus_bit_32 bus;
          en1 : in bit;
          a2 : in bit_vector(depth-1 downto 0);
          q2 : out bus_bit_32 bus;
          en2 : in bit;
          a3 : in bit_vector(depth-1 downto 0);
          d3 : in bit_32;
          en3 : in bit);
end reg_file_32_rrw;

architecture behaviour of reg_file_32_rrw is

begin

    reg_file: process (a1, en1, a2, en2, a3, d3, en3)

        subtype reg_addr is natural range 0 to depth-1;
        type register_array is array (reg_addr) of bit_32;

        variable registers : register_array;

    begin
        if en3 = '1' then
            registers(bits_to_natural(a3)) := d3;
        end if;
        if en1 = '1' then
            q1 <= registers(bits_to_natural(a1)) after Tac;
        else
            q1 <= null after Tpd;
        end if;
        if en2 = '1' then
            q2 <= registers(bits_to_natural(a2)) after Tac;
        else
            q2 <= null after Tpd;
        end if;
    end process reg_file;

end behaviour;
```

Figure 7-23. Description of the 3-port register file.

7.6.8. Arithmetic & Logic Unit

The description of the ALU is listed in Figure 7-24. The package ALU_32_types defines an enumerated type for specifying the ALU function. This must be placed in a package, since it is required for both the ALU description and for entities that make use of the ALU. There is no corresponding package body, since the type is fully defined in the package specification.

The ALU entity declaration uses the ALU_32_types package as well as the general dp32_types package. It has two operand input ports, a result output and condition code output ports, and a command input port. This last port is an example of a port which is of an enumerated type, since at this stage of design, no encoding is known or specified for the ALU function command.

The ALU behaviour is implemented by the process ALU_function, sensitive to changes on the operand and command input ports. If the command to be performed is an arithmetic operation, the model firstly converts the operands to integers. This is followed by a case statement dispatching on the command. For the disable command, no operation is performed, and for the pass1 command, the result is operand1 unchanged. The result for logic commands is derived by applying the corresponding VHDL logical operations to the bit-vector operands. For arithmetic commands the result is computed the same was as it was in the behavioural model of the DP32 presented in Section 7.4. Also, the overflow condition code bit (cc_V), which is only defined for arithmetic operations, is assigned here. Finally, the result and remaining condition code bits are assigned. The result output is only driven if the command is not disable, otherwise it is disconnected.

```
package ALU_32_types is

    type ALU_command is (disable, pass1, incr1,
                         add, subtract, multiply, divide,
                         log_and, log_or, log_xor, log_mask);

end ALU_32_types;


use work.dp32_types.all, work.ALU_32_types.all;

entity ALU_32 is
    generic (Tpd : Time := unit_delay);
    port (operand1 : in bit_32;
          operand2 : in bit_32;
          result : out bus_bit_32 bus;
          cond_code : out CC_bits;
          command : in ALU_command);
end ALU_32;
```

Figure 7-24. Description of the Arithmetic and Logic Unit.

```
architecture behaviour of ALU_32 is

    alias cc_V : bit is cond_code(2);
    alias cc_N : bit is cond_code(1);
    alias cc_Z : bit is cond_code(0);

begin

    ALU_function: process (operand1, operand2, command)

        variable a, b : integer;
        variable temp_result : bit_32;

    begin
        case command is
            when add | subtract | multiply | divide =>
                a := bits_to_int(operand1);
                b := bits_to_int(operand2);
            when incr1 =>
                a := bits_to_int(operand1);
                b := 1;
            when others =>
                null;
        end case;
        case command is
            when disable =>
                null;
            when pass1 =>
                temp_result := operand1;
            when log_and =>
                 temp_result := operand1 and operand2;
            when log_or =>
                temp_result := operand1 or operand2;
            when log_xor =>
                temp_result := operand1 xor operand2;
            when log_mask =>
                temp_result := operand1 and not operand2;
            when add | incr1 =>
                if b > 0 and a > integer'high-b then      -- positive overflow
                    int_to_bits(((integer'low+a)+b)-integer'high-1, temp_result);
                    cc_V <= '1' after Tpd;
                elsif b < 0 and a < integer'low-b then   -- negative overflow
                     int_to_bits(((integer'high+a)+b)-integer'low+1, temp_result);
                    cc_V <= '1' after Tpd;
                else
                    int_to_bits(a + b, temp_result);
                    cc_V <= '0' after Tpd;
                end if;
            when subtract =>
                if b < 0 and a > integer'high+b then      -- positive overflow
                    int_to_bits(((integer'low+a)-b)-integer'high-1, temp_result);
                    cc_V <= '1' after Tpd;
                elsif b > 0 and a < integer'low+b then  -- negative overflow
                    int_to_bits(((integer'high+a)-b)-integer'low+1, temp_result);
                    cc_V <= '1' after Tpd;
                else
                    int_to_bits(a - b, temp_result);
                    cc_V <= '0' after Tpd;
                end if;
```

Figure 7-24 (continued).

```
            when multiply =>
                if ((a>0 and b>0) or (a<0 and b<0))     -- result positive
                        and (abs a > integer'high / abs b) then
                    -- positive overflow
                    int_to_bits(integer'high, temp_result);
                    cc_V <= '1' after Tpd;
                elsif ((a>0 and b<0) or (a<0 and b>0))     -- result negative
                        and ((- abs a) < integer'low / abs b) then
                    -- negative overflow
                    int_to_bits(integer'low, temp_result);
                    cc_V <= '1' after Tpd;
                else
                    int_to_bits(a * b, temp_result);
                    cc_V <= '0' after Tpd;
                end if;
            when divide =>
                if b=0 then
                    if a>=0 then                        -- positive overflow
                        int_to_bits(integer'high, temp_result);
                    else
                        int_to_bits(integer'low, temp_result);
                    end if;
                    cc_V <= '1' after Tpd;
                else
                    int_to_bits(a / b, temp_result);
                    cc_V <= '0' after Tpd;
                end if;
        end case;
        if command /= disable then
            result <= temp_result after Tpd;
        else
            result <= null after Tpd;
        end if;
        cc_Z <= bool_to_bit(temp_result = X"00000000") after Tpd;
        cc_N <= bool_to_bit(temp_result(31) = '1') after Tpd;
    end process ALU_function;

end behaviour;
```

Figure 7-24 (continued).

```
use work.dp32_types.all;

entity cond_code_comparator is
    generic (Tpd : Time := unit_delay);
    port (cc : in CC_bits;
          cm : in cm_bits;
          result : out bit);
end cond_code_comparator;

architecture behaviour of cond_code_comparator is
    alias cc_V : bit is cc(2);
    alias cc_N : bit is cc(1);
    alias cc_Z : bit is cc(0);
    alias cm_i : bit is cm(3);
    alias cm_V : bit is cm(2);
    alias cm_N : bit is cm(1);
    alias cm_Z : bit is cm(0);
begin
    result <= bool_to_bit(((cm_V and cc_V)
                          or (cm_N and cc_N)
                          or (cm_Z and cc_Z)) = cm_i) after Tpd;
end behaviour;
```

Figure 7-25. Description of the condition code comparator.

7.6.9. Condition Code Comparator

The description of the condition code comparator is listed in Figure 7-25. The cc input port contains the three condition code bits V, N and Z, and the cm input contains the four condition mask bits derived from a DP32 instruction. Aliases for each of these bits are declared in the architecture body. The behaviour is implemented by a single concurrent signal assignment statement, which is sensitive to all of the input bits. Whenever any of the bits changes value, the assignment will be resumed and a new result bit computed.

7.6.10. Structural Architecture of the DP32

In this section, a structural architecture body for the DP32 processor, corresponding to Figure 7-16, will be described. See Figure 7-26 for a listing of the architecture body.

```
use work.dp32_types.all, work.ALU_32_types.all;

architecture RTL of dp32 is

    component reg_file_32_rrw
        generic (depth : positive);
        port (a1 : in bit_vector(depth-1 downto 0);
                q1 : out bus_bit_32 bus;
                en1 : in bit;
                a2 : in bit_vector(depth-1 downto 0);
                q2 : out bus_bit_32 bus;
                en2 : in bit;
                a3 : in bit_vector(depth-1 downto 0);
                d3 : in bit_32;
                en3 : in bit);
    end component;

    component mux2
        generic (width : positive);
        port (i0, i1 : in bit_vector(width-1 downto 0);
                y : out bit_vector(width-1 downto 0);
                sel : in bit);
    end component;

    component PC_reg
        port (d : in bit_32;
                q : out bus_bit_32 bus;
                latch_en : in bit;
                out_en : in bit;
                reset : in bit);
    end component;

    component ALU_32
        port (operand1 : in bit_32;
                operand2 : in bit_32;
                result : out bus_bit_32 bus;
                cond_code : out CC_bits;
                command : in ALU_command);
    end component;

    component cond_code_comparator
        port (cc : in CC_bits;
                cm : in cm_bits;
                result : out bit);
    end component;

    component buffer_32
        port (a : in bit_32;
                b : out bus_bit_32 bus;
                en : in bit);
    end component;

    component latch
        generic (width : positive);
        port (d : in bit_vector(width-1 downto 0);
                q : out bit_vector(width-1 downto 0);
                en : in bit);
    end component;
```

Figure 7-26. Structural description of the DP32 processor.

```
    component latch_buffer_32
        port (d : in bit_32;
                q : out bus_bit_32 bus;
                latch_en : in bit;
                out_en : in bit);
    end component;

    component signext_8_32
        port (a : in bit_8;
                b : out bus_bit_32 bus;
                en : in bit);
    end component;

    signal op1_bus : bus_bit_32;
    signal op2_bus : bus_bit_32;
    signal r_bus : bus_bit_32;

    signal ALU_CC : CC_bits;
    signal CC : CC_bits;

    signal current_instr : bit_32;
    alias instr_a1 : bit_8 is current_instr(15 downto 8);
    alias instr_a2 : bit_8 is current_instr(7 downto 0);
    alias instr_a3 : bit_8 is current_instr(23 downto 16);
    alias instr_op : bit_8 is current_instr(31 downto 24);
    alias instr_cm : cm_bits is current_instr(19 downto 16);

    signal reg_a2 : bit_8;
    signal reg_result : bit_32;

    signal addr_latch_en : bit;
    signal disp_latch_en : bit;
    signal disp_out_en : bit;
    signal d2_en : bit;
    signal dr_en : bit;
    signal instr_latch_en : bit;
    signal immed_signext_en : bit;
    signal ALU_op : ALU_command;
    signal CC_latch_en : bit;
    signal CC_comp_result : bit;
    signal PC_latch_en : bit;
    signal PC_out_en : bit;
    signal reg_port1_en : bit;
    signal reg_port2_en : bit;
    signal reg_port3_en : bit;
    signal reg_port2_mux_sel : bit;
    signal reg_res_latch_en : bit;

begin -- architecture RTL of dp32

    reg_file : reg_file_32_RRW
        generic map (depth => 8)
        port map (a1 => instr_a1, q1 => op1_bus, en1 => reg_port1_en,
                    a2 => reg_a2, q2 => op2_bus, en2 => reg_port2_en,
                    a3 => instr_a3, d3 => reg_result, en3 => reg_port3_en);

    reg_port2_mux : mux2
        generic map (width => 8)
        port map (i0 => instr_a2, i1 => instr_a3, y => reg_a2,
                    sel => reg_port2_mux_sel);
```

Figure 7-26 (continued).

The architecture refers to the items declared in the packages dp32_types and ALU_32_types, so a use clause for these packages is included. The declaration section of the architecture contains a number of component declarations, corresponding to the entity declarations listed in Sections 7.6.1 to 7.6.9. Instances of these components are subsequently used to construct the processor architecture.

Next, a number of signals are declared, corresponding to the buses illustrated in Figure 7-16. These are followed by further signal declarations for control signals not shown in the figure. The control signals are used to connect the data path component instances with the control unit implemented in the block called controller.

```
reg_res_latch : latch
    generic map (width => 32)
    port map (d => r_bus, q => reg_result, en => reg_res_latch_en);

PC : PC_reg
    port map (d => r_bus, q => op1_bus,
              latch_en => PC_latch_en, out_en => PC_out_en,
              reset => reset);

ALU : ALU_32
    port map (operand1 => op1_bus, operand2 => op2_bus,
              result => r_bus, cond_code => ALU_CC,
              command => ALU_op);

CC_reg : latch
    generic map (width => 3)
    port map (d => ALU_CC, q => CC, en   => CC_latch_en);

CC_comp : cond_code_comparator
    port map (cc => CC, cm => instr_cm, result => CC_comp_result);

dr_buffer : buffer_32
    port map (a => d_bus, b => r_bus, en => dr_en);

d2_buffer : buffer_32
    port map (a => op2_bus, b => d_bus, en => d2_en);

disp_latch : latch_buffer_32
    port map (d => d_bus, q => op2_bus,
              latch_en => disp_latch_en, out_en => disp_out_en);

addr_latch : latch
    generic map (width => 32)
    port map (d => r_bus, q => a_bus, en => addr_latch_en);

instr_latch : latch
    generic map (width => 32)
    port map (d => r_bus, q => current_instr, en => instr_latch_en);

immed_signext : signext_8_32
    port map (a => instr_a2, b => op2_bus, en => immed_signext_en);
```

Figure 7-26 (continued).

```
controller : block
    port (phi1, phi2 : in bit;
            reset : in bit;
            opcode : in bit_8;
            read, write, fetch : out bit;
            ready : in bit;
            addr_latch_en : out bit;
            disp_latch_en : out bit;
            disp_out_en : out bit;
            d2_en : out bit;
            dr_en : out bit;
            instr_latch_en : out bit;
            immed_signext_en : out bit;
            ALU_op : out ALU_command;
            CC_latch_en : out bit;
            CC_comp_result : in bit;
            PC_latch_en : out bit;
            PC_out_en : out bit;
            reg_port1_en : out bit;
            reg_port2_en : out bit;
            reg_port3_en : out bit;
            reg_port2_mux_sel : out bit;
            reg_res_latch_en : out bit);
    port map (phi1 => phi1, phi2 => phi2,
                reset => reset,
                opcode => instr_op,
                read => read, write => write, fetch => fetch,
                ready => ready,
                addr_latch_en => addr_latch_en,
                disp_latch_en => disp_latch_en,
                disp_out_en => disp_out_en,
                d2_en => d2_en,
                dr_en => dr_en,
                instr_latch_en => instr_latch_en,
                immed_signext_en => immed_signext_en,
                ALU_op => ALU_op,
                CC_latch_en => CC_latch_en,
                CC_comp_result => CC_comp_result,
                PC_latch_en => PC_latch_en, PC_out_en => PC_out_en,
                reg_port1_en => reg_port1_en,
                reg_port2_en => reg_port2_en,
                reg_port3_en => reg_port3_en,
                reg_port2_mux_sel => reg_port2_mux_sel,
                reg_res_latch_en => reg_res_latch_en);
```

Figure 7-26 (continued).

The control unit is a state machine, whose behaviour is described by a single process called state_machine. The controller sequences through the states listed in the declaration of the type controller_state to fetch, decode and execute instructions. The variable state holds the controller state for the current clock cycle, and next_state is set to determine the state for the next clock cycle. Write_back_pending is a flag used to schedule a register write operation for the next clock cycle. The constant ALU_op_select is a lookup table used to determine the ALU function from the instruction op-code.

Figure 7-26 (continued).

```
begin -- block controller

    state_machine: process

        type controller_state is
            (resetting, fetch_0, fetch_1, fetch_2, decode,
                disp_fetch_0, disp_fetch_1, disp_fetch_2,
                execute_0, execute_1, execute_2);

        variable state, next_state : controller_state;
        variable write_back_pending : boolean;

        type ALU_op_select_table is
            array (natural range 0 to 255) of ALU_command;

        constant ALU_op_select : ALU_op_select_table :=
            (16#00# => add,
                16#01# => subtract,
                16#02# => multiply,
                16#03# => divide,
                16#10# => add,
                16#11# => subtract,
                16#12# => multiply,
                16#13# => divide,
                16#04# => log_and,
                16#05# => log_or,
                16#06# => log_xor,
                16#07# => log_mask,
                others => disable);
```

The body of the state machine process starts by waiting for the leading edge of the phi1 clock, indicating the start of a clock cycle. When this occurs, the reset signal is checked, and if it is asserted the controller state is set to resetting and all control outputs are negated. On the other hand, if reset is negated, the controller state is updated to the previously computed next state.

```
begin -- process state_machine
    --
    -- start of clock cycle
    --
    wait until phi1 = '1';
    --
    -- check for reset
    --
    if reset = '1' then
        state := resetting;
        --
        -- reset external bus signals
        --
        read <= '0' after Tpd;
        fetch <= '0' after Tpd;
        write <= '0' after Tpd;
        --
        -- reset dp32 internal control signals
        --
        addr_latch_en <= '0' after Tpd;
        disp_latch_en <= '0' after Tpd;
        disp_out_en <= '0' after Tpd;
        d2_en <= '0' after Tpd;
        dr_en <= '0' after Tpd;
        instr_latch_en <= '0' after Tpd;
        immed_signext_en <= '0' after Tpd;
        ALU_op <= disable after Tpd;
        CC_latch_en <= '0' after Tpd;
        PC_latch_en <= '0' after Tpd;
        PC_out_en <= '0' after Tpd;
        reg_port1_en <= '0' after Tpd;
        reg_port2_en <= '0' after Tpd;
        reg_port3_en <= '0' after Tpd;
        reg_port2_mux_sel <= '0' after Tpd;
        reg_res_latch_en <= '0' after Tpd;
        --
        -- clear write-back flag
        --
        write_back_pending := false;
        --
    else -- reset = '0'
        state := next_state;
    end if;
```

Figure 7-26 (continued).

The remainder of the state machine body is a case statement using the current state to determine the action to be performed for this clock cycle. If the processor is being reset (in the resetting state), it waits until the trailing edge of phi2 at the end of the clock cycle, and checks the reset signal again. If reset has been negated, the processor can start fetching instructions, so the next state is set to fetch_0, otherwise it is is set to resetting again.

```
--
-- dispatch action for current state
--
case state is
    when resetting =>
        --
        -- check for reset going inactive at end of clock cycle
        --
        wait until phi2 = '0';
        if reset = '0' then
            next_state := fetch_0;
        else
            next_state := resetting;
        end if;
        --
    when fetch_0 =>
        --
        -- clean up after previous execute cycles
        --
        reg_port1_en <= '0' after Tpd;
        reg_port2_mux_sel <= '0' after Tpd;
        reg_port2_en <= '0' after Tpd;
        immed_signext_en <= '0' after Tpd;
        disp_out_en <= '0' after Tpd;
        dr_en <= '0' after Tpd;
        read <= '0' after Tpd;
        d2_en <= '0' after Tpd;
        write <= '0' after Tpd;
        --
        -- handle pending register write-back
        --
        if write_back_pending then
            reg_port3_en <= '1' after Tpd;
        end if;
        --
        -- enable PC via ALU to address latch
        --
        PC_out_en <= '1' after Tpd;          -- enable PC onto op1_bus
        ALU_op <= pass1 after Tpd;           -- pass PC to r_bus
        --
        wait until phi2 = '1';
        addr_latch_en <= '1' after Tpd;      -- latch instr address
        wait until phi2 = '0';
        addr_latch_en <= '0' after Tpd;
        --
        next_state := fetch_1;
        --
```

Figure 7-26 (continued).

The processor fetches an instruction from memory by sequencing through the states fetch_0, fetch_1 and fetch_2 on successive clock cycles. Figure 7-27 shows the timing of control signals for an instruction fetch. The fetch_0 processor cycle corresponds to a Ti cycle on the memory bus. During this cycle, the PC register output is enabled onto the op1 bus, and the ALU function set to pass1. The ALU passes the PC value through to the result bus, and it is latched into the memory address register during the second half of the cycle. The PC value is thus set up on the memory address bus. The fetch_1 cycle corresponds to a memory bus T1 cycle. The controller starts the memory transaction by asserting fetch and read. At the same time, it changes the ALU function code to incr1, causing the ALU to place

```
when fetch_1 =>
    --
    -- clear pending register write-back
    --
    if write_back_pending then
        reg_port3_en <= '0' after Tpd;
        write_back_pending := false;
    end if;
    --
    -- increment PC & start bus read
    --
    ALU_op <= incr1 after Tpd;          -- increment PC onto r_bus
    fetch <= '1' after Tpd;
    read <= '1' after Tpd;
    --
    wait until phi2 = '1';
    PC_latch_en <= '1' after Tpd;       -- latch incremented PC
    wait until phi2 = '0';
    PC_latch_en <= '0' after Tpd;
    --
    next_state := fetch_2;
    --
when fetch_2 =>
    --
    -- cleanup after previous fetch_1
    --
    PC_out_en <= '0' after Tpd;         -- disable PC from op1_bus
    ALU_op <= disable after Tpd;        -- disable ALU from r_bus
                  --
    -- latch current instruction
    --
    dr_en <= '1' after Tpd;           -- enable fetched instr onto r_bus
    --
    wait until phi2 = '1';
    instr_latch_en <= '1' after Tpd;  -- latch fetched instr from r_bus
    wait until phi2 = '0';
    instr_latch_en <= '0' after Tpd;
    --
    if ready = '1' then
        next_state := decode;
    else
        next_state := fetch_2;          -- extend bus read
    end if;
```

Figure 7-26 (continued).

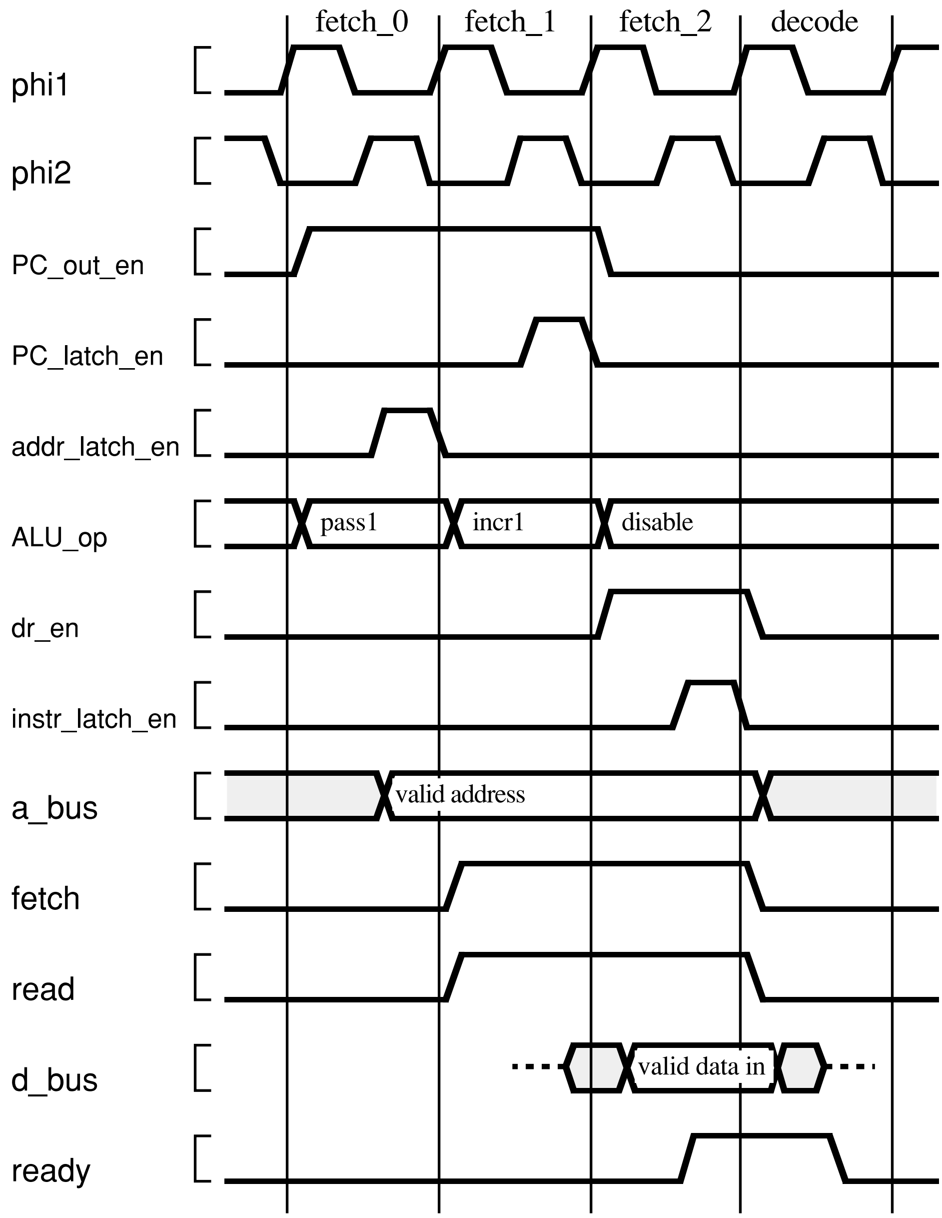

Figure 7-27. Timing for DP32 instruction fetch.

the incremented PC value on the result bus. This is then latched back into the PC register during the second half of the cycle. The fetch_2 processor cycle corresponds to the memory bus T2 cycle, during which data is returned to the processor from the memory. The controller disables the PC from the op1 bus and the ALU from the result bus, and enables the data input buffer to accept memory data onto the result bus. This data is latched into the current instruction register during the second half of the cycle. If ready is false, the processor repeats the F2 cycle, otherwise it completes the bus transaction and moves to the decode state, corresponding to a bus Ti cycle.

Returning to the VHDL description, we see that the fetch_0 branch of the case statement implements the first cycle of an instruction fetch. Firstly, any signals left asserted from previous cycle are negated again. Next, any register write scheduled from the previously executed instruction is handled. (This will be described fully below.) Then the PC register output is enabled and the ALU function set, as described above. The process then waits until the leading edge of phi2, by which time the PC should be valid on the result bus. It pulses the address latch enable signal by asserting it, waiting until the trailing edge of phi2, then negating the signal. Finally, the next state variable is set to fetch_1, so that when the process resumes in the next cycle, it will move to this state.

When the process is in state fetch_1, it starts the cycle by terminating any register write back that may have been pending. It then changes the ALU function code to increment the PC value, and starts the bus transaction. In the second half of the cycle, when phi2 is asserted, the PC latch enable is asserted to store the incremented PC value. The next state is then set to

```
when decode =>
    --
    -- terminate bus read from previous fetch_2
    --
    fetch <= '0' after Tpd;
    read <= '0' after Tpd;
    dr_en <= '0' after Tpd;         -- disable fetched instr from r_bus
    --
    -- delay to allow decode logic to settle
    --
    wait until phi2 = '0';
    --
    -- next state based on opcode of currect instruction
    --
    case opcode is
        when op_add | op_sub | op_mul | op_div
                | op_addq | op_subq | op_mulq | op_divq
                | op_land | op_lor | op_lxor | op_lmask
                | op_ldq | op_stq =>
            next_state := execute_0;
        when op_ld | op_st =>
            next_state := disp_fetch_0;      -- fetch offset
        when op_br | op_bi =>
            if CC_comp_result = '1' then     -- if branch taken
                next_state := disp_fetch_0;  --    fetch displacement
            else                             -- else
                next_state := execute_0;     --    increment PC
                                             --    past displacement
            end if;
        when op_brq | op_biq =>
            if CC_comp_result = '1' then     -- if branch taken
                next_state := execute_0;     --    add immed
                                             --       displacement to PC
            else                             -- else
                next_state := fetch_0;       --    no action needed
            end if;
        when others =>
            assert false report "illegal instruction" severity warning;
            next_state := fetch_0;           -- ignore and carry on
    end case; -- op
    --
```

Figure 7-26 (continued).

fetch_2.

The last cycle of the instruction fetch is state fetch_2. The controller disables the PC register and ALU outputs, and enables the buffer between the memory data bus and the result bus. During the second half of the cycle, it asserts the instruction register latch enable. At the end of the cycle, when phi2 has returned to '0', the ready input is checked. If it is asserted, the state machine can continue to the d ecode state in the next cycle, otherwise the fetch_2 state must be repeated.

In the decode state, the controller terminates the previous bus transaction and disables the bus input buffer. It then delays for the rest of the cycle, modeling the time required for decode logic to analyse the current instruction and for the condition code comparator to stabilize. The op-code part of the instruction is then examined to determine the next state. For arithmetic, logical and quick load/store instructions, the next state is execute_0, in which the instruction is interpreted. For load/store instructions with a long displacement, a bus transaction must be performed to read the displacement, so the next state is disp_fetch_0. For branch instructions with a long displacement, the fetch is only required if the branch is to be taken, in which case the next state is disp_fetch_0. Otherwise the next state is execute_0, in which the PC will be incremented past the displacement stored in memory. For branch quick instructions, the displacement is encoded in the instruction. If the branch is taken, the next state is execute_0 to update the PC. Otherwise no further action is needed to interpret the instruction, so the next state is fetch_0. If any other op-code is detected, an assertion is used to report the illegal instruction. The instruction is ignored and execution continues with the next instruction, so the next state is fetch_0.

```
        when disp_fetch_0 =>
            --
            -- enable PC via ALU to address latch
            --
            PC_out_en <= '1' after Tpd;          -- enable PC onto op1_bus
            ALU_op <= pass1 after Tpd;           -- pass PC to r_bus
            --
            wait until phi2 = '1';
            addr_latch_en <= '1' after Tpd;      -- latch displacement address
            wait until phi2 = '0';
            addr_latch_en <= '0' after Tpd;
            --
            next_state := disp_fetch_1;
            --
        when disp_fetch_1 =>
            --
            -- increment PC & start bus read
            --
            ALU_op <= incr1 after Tpd;           -- increment PC onto r_bus
            fetch <= '1' after Tpd;
            read <= '1' after Tpd;
            --
            wait until phi2 = '1';
            PC_latch_en <= '1' after Tpd;        -- latch incremented PC
            wait until phi2 = '0';
            PC_latch_en <= '0' after Tpd;
            --
            next_state := disp_fetch_2;
            --
        when disp_fetch_2 =>
            --
            -- cleanup after previous disp_fetch_1
            --
            PC_out_en <= '0' after Tpd;          -- disable PC from op1_bus
            ALU_op <= disable after Tpd;         -- disable ALU from r_bus
            --
            -- latch displacement
            --
            wait until phi2 = '1';
            disp_latch_en <= '1' after Tpd;  -- latch fetched disp from r_bus
            wait until phi2 = '0';
            disp_latch_en <= '0' after Tpd;
            --
            if ready = '1' then
                next_state := execute_0;
            else
                next_state := disp_fetch_2;      -- extend bus read
            end if;
```

Figure 7-26 (continued).

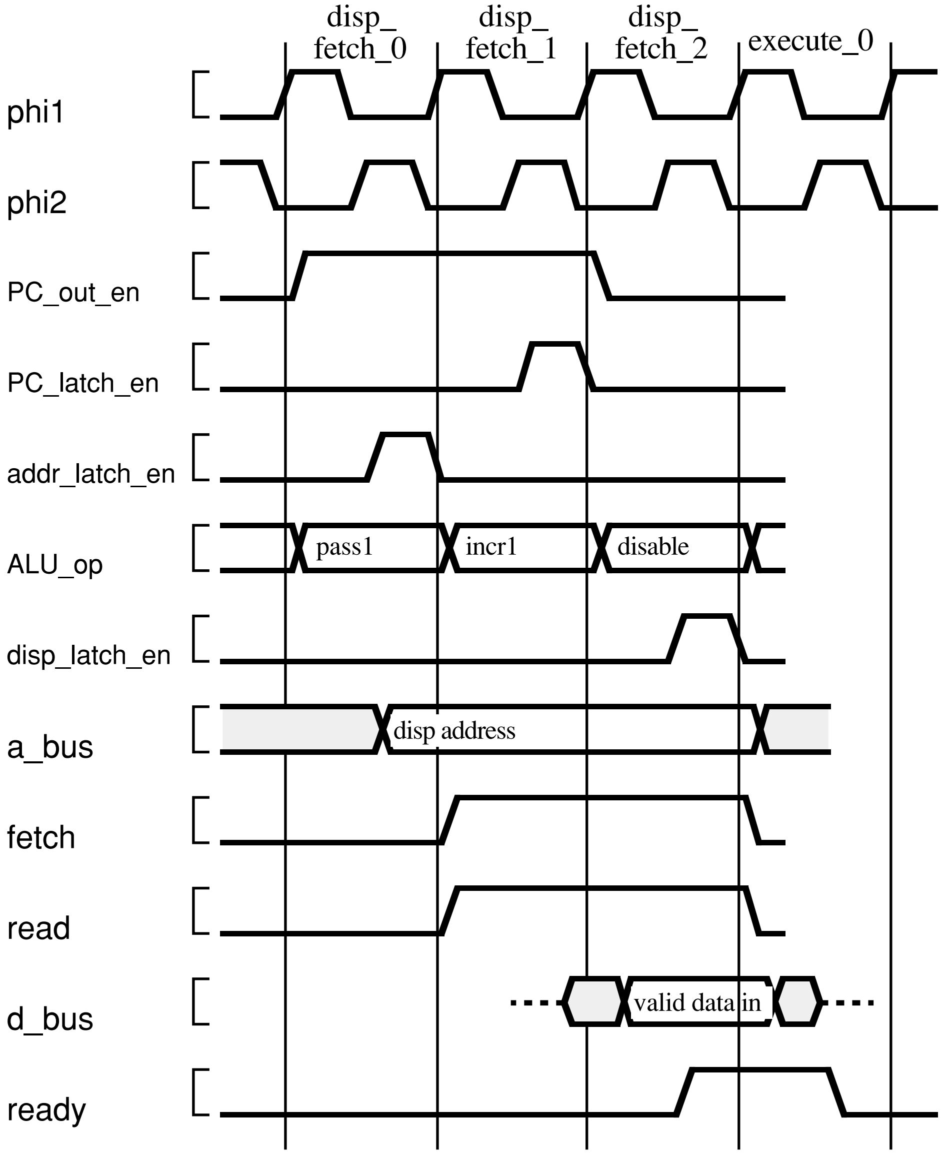

Figure 7-28. Timing for DP32 displacement fetch.

The sequence for fetching a displacement from memory is similar to that for fetching the instruction word. The only difference is that instead of the read word being enabled onto the result bus and latched into the instruction register, the word is simply latched from the memory data bus into the displacement latch. The timing for a displacement fetch is shown in Figure 7-28. The sequence consists of the processor states disp_fetch_0, disp_fetch_1 and one or more repetitions of disp_fetch_2, corresponding to bus states Ti, T1 and T2 respectively. This sequence is always followed by the first execute state, corresponding to the bus Ti state at the end of the bus transaction. In the VHDL description, the case branches for disp_fetch_0, disp_fetch_1 and disp_fetch_2 implement this behaviour.

```
when execute_0 =>
    --
    -- terminate bus read from previous disp_fetch_2
    --
    fetch <= '0' after Tpd;
    read <= '0' after Tpd;
    --
    case opcode is
        when op_add | op_sub | op_mul | op_div
                | op_addq | op_subq | op_mulq | op_divq
                | op_land | op_lor | op_lxor | op_lmask =>
            -- enable r1 onto op1_bus
            reg_port1_en <= '1' after Tpd;
            if opcode = op_addq or opcode = op_subq
                    or opcode = op_mulq or opcode = op_divq then
                -- enable i8 onto op2_bus
                immed_signext_en <= '1' after Tpd;
            else
                -- select a2 as port2 address
                reg_port2_mux_sel <= '0' after Tpd;
                -- enable r2 onto op2_bus
                reg_port2_en <= '1' after Tpd;
            end if;
            -- select ALU operation
            ALU_op <= ALU_op_select(bits_to_int(opcode)) after Tpd;
            --
            wait until phi2 = '1';
            -- latch cond codes from ALU
            CC_latch_en <= '1' after Tpd;
            -- latch result for reg write
            reg_res_latch_en <= '1' after Tpd;
            wait until phi2 = '0';
            CC_latch_en <= '0' after Tpd;
            reg_res_latch_en <= '0' after Tpd;
            --
            next_state := fetch_0;          -- execution complete
            write_back_pending := true; -- register write_back required
            --
        when op_ld | op_st | op_ldq | op_stq =>
            -- enable r1 to op1_bus
            reg_port1_en <= '1' after Tpd;
            if opcode = op_ld or opcode = op_st then
                -- enable displacement to op2_bus
                disp_out_en <= '1' after Tpd;
            else
                -- enable i8 to op2_bus
                immed_signext_en <= '1' after Tpd;
            end if;
            ALU_op <= add after Tpd;  -- effective address to r_bus
            --
            wait until phi2 = '1';
            addr_latch_en <= '1' after Tpd; -- latch effective address
            wait until phi2 = '0';
            addr_latch_en <= '0' after Tpd;
            --
            next_state := execute_1;
            --
```

Figure 7-26 (continued).

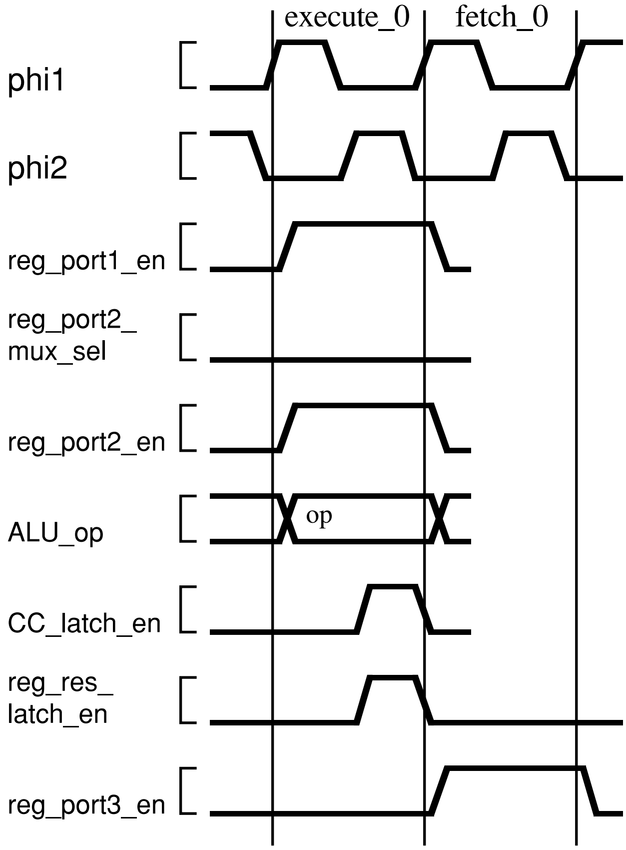

Figure 7-29. Execution of register/register operations.

Execution of instructions starts in state execute_0. The first action is to negate the bus control signals that may have been active from a previous displacement fetch sequence. Subsequent action depends on the instruction being executed, so a nested case statement is used, with the op-code as the selection expression.

Arithmetic and logic instructions only require one cycle to exectute. The processor timing for the case where both operands are in registers is shown in Figure 7-29. The address for register port 1 is derived from the r1 field of the current instruction, and this port output is enabled onto the op1 bus. The multiplexor for the address for register port 2 is set to select field r2 of the current instruction, and this port output is enabled onto the op2 bus. The ALU function code is set according to the op-code of the current instruction, and the ALU output is placed on the result bus. During the second half of the cycle, when the ALU result and condition codes are stable, the register result latch and condition code latch are enabled, capturing the results of the operation. In the next cycle, the register read ports and the latches are are disabled, and the register write port is enabled to write the result back into the destination register. This write back operation overlaps the first cycle of the next instruction fetch. The result register address, derived from the r3 field of the current instruction, is not overwritten until the end of the next instruction fetch, so the write back is performed to the correct register.

The timing for arithmetic and logical instructions where the second operand is an immediate constant is shown in Figure 7-30. The difference is that register port 2 is not enabled; instead, the sign extension buffer is enabled. This converts the 8-bit signed i8 field of the current instruction to a 32-bit signed integer on the op2 bus.

Looking again at the exectute_0 branch of the state machine, the nested case statement contains a branch for arithmetic and logical instructions. It firstly enables port 1 of the register file, and then enables either port 2 or the sign extension buffer, depending on the op-code. The lookup table ALU_op_select is indexed by the op-code to determine the ALU function code. The process then waits until the leading edge of phi2, and asserts the register result and condition code latch enables while phi2 is '1'. At the end of the cycle, the next state is set to fetch_0, and the write back pending flag is set. During the subsequent instruction fetch, this flag is checked (in the fetch_0 branch of the outer case statement). The register port 3 write enable control signal is asserted during the fetch_0 state, and then at the beginning of the fetch_1 state it is negated and the flag cleared.

```
            when op_br | op_bi | op_brq | op_biq =>
                if CC_comp_result = '1' then
                    if opcode = op_br then
                        PC_out_en <= '1' after Tpd;
                        disp_out_en <= '1' after Tpd;
                    elsif opcode = op_bi then
                        reg_port1_en <= '1' after Tpd;
                        disp_out_en <= '1' after Tpd;
                    elsif opcode = op_brq then
                        PC_out_en <= '1' after Tpd;
                        immed_signext_en <= '1' after Tpd;
                    else -- opcode = op_biq
                        reg_port1_en <= '1' after Tpd;
                        immed_signext_en <= '1' after Tpd;
                    end if;
                    ALU_op <= add after Tpd;
                else
                    assert opcode = op_br or opcode = op_bi
                        report "reached state execute_0 "
                            & "when brq or biq not taken"
                        severity error;
                    PC_out_en <= '1' after Tpd;
                    ALU_op <= incr1 after Tpd;
                end if;
                --
                wait until phi2 = '1';
                PC_latch_en <= '1' after Tpd;   -- latch incremented PC
                wait until phi2 = '0';
                PC_latch_en <= '0' after Tpd;
                --
                next_state := fetch_0;
                --
            when others =>
                null;
        end case; -- op
        --
```

Figure 7-26 (continued).

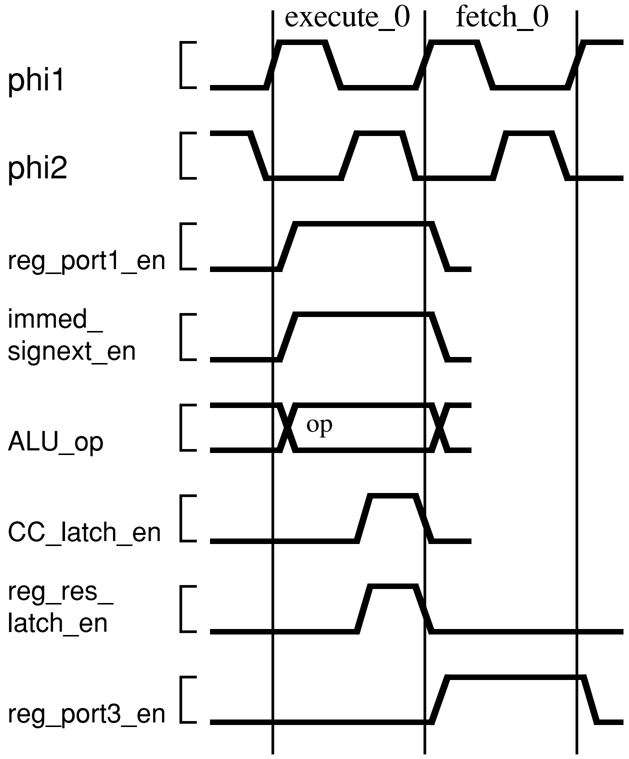

Figure 7-30. Execution of register/immed operations.

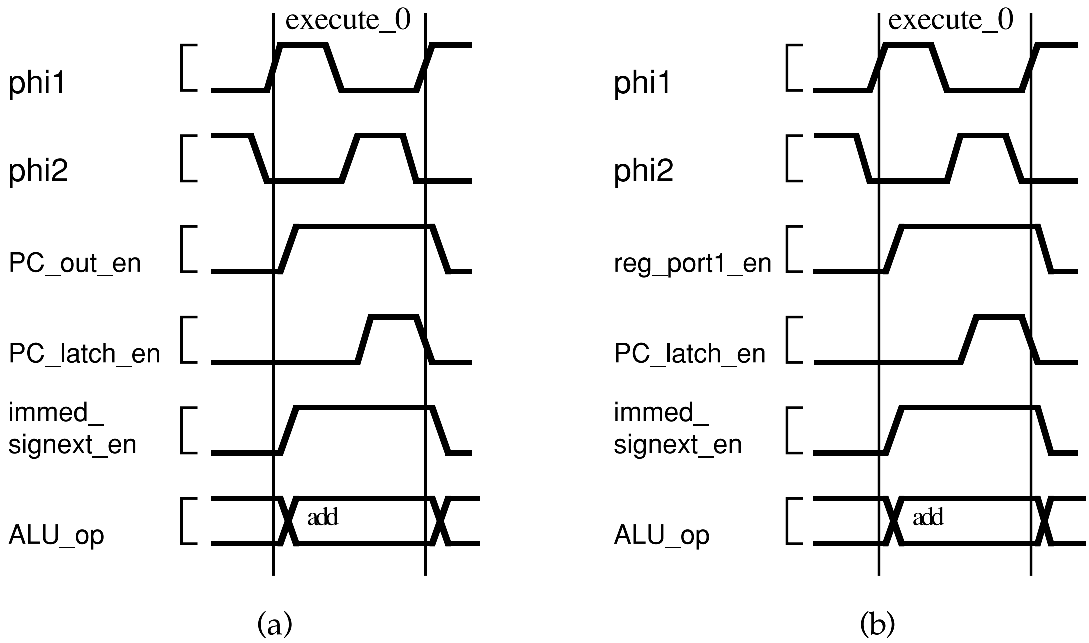

Figure 7-31. Execution of quick branch with branch taken.

```
                when execute_1 =>
                    --
                    -- opcode is load or store instruction.
                    -- cleanup after previous execute_0
                    --
                    reg_port1_en <= '0' after Tpd;
                    if opcode = op_ld or opcode = op_st then
                        -- disable displacement from op2_bus
                        disp_out_en <= '0' after Tpd;
                    else
                        -- disable i8 from op2_bus
                        immed_signext_en <= '0' after Tpd;
                    end if;
                    ALU_op <= add after Tpd;        -- disable ALU from r_bus
                    --
                    -- start bus cycle
                    --
                    if opcode = op_ld or opcode = op_ldq then
                        fetch <= '0' after Tpd;             -- start bus read
                        read <= '1' after Tpd;
                    else -- opcode = op_st or opcode = op_stq
                        reg_port2_mux_sel <= '1' after Tpd;     -- address a3 to port2
                        reg_port2_en <= '1' after Tpd;          -- reg port2 to op2_bus
                        d2_en <= '1' after Tpd;       -- enable op2_bus to d_bus buffer
                        write <= '1' after Tpd;       -- start bus write
                    end if;
                    --
                    next_state := execute_2;
                    --
                when execute_2 =>
                    --
                    -- opcode is load or store instruction.
                    -- for load, enable read data onto r_bus
                    --
                    if opcode = op_ld or opcode = op_ldq then
                        dr_en <= '1' after Tpd;       -- enable data to r_bus
                        wait until phi2 = '1';
                        -- latch data in reg result latch
                        reg_res_latch_en <= '1' after Tpd;
                        wait until phi2 = '0';
                        reg_res_latch_en <= '0' after Tpd;
                        write_back_pending := true;         -- write-back pending
                    end if;
                    --
                    next_state := fetch_0;
                    --
            end case; -- state
        end process state_machine;

    end block controller;

end RTL;
```

Figure 7-26 (continued).

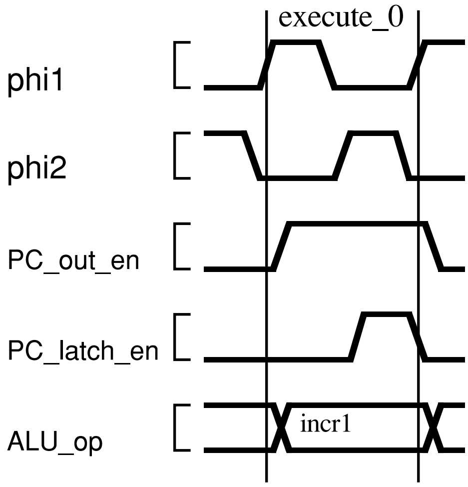

Figure 7-32. Execution of branch with branch not taken.

We now move on to the execution of branch instructions. We saw previously that for quick branches, when the branch is not taken execution completes after the decode state. When the branch is taken a single execute cycle is required to update the PC with the effective address. The timing for this case is shown in Figure 7-31. Figure 7-31(a) shows an ordinary quick branch, in which the PC is enabled onto the op1 bus. Figure 7-31(b) shows an indexed quick branch, in which the index register, read from register file port 1 is enabled onto the op1 bus. The sign extension buffer is enabled to place the immediate displacement on the op2 bus, and the ALU function code is set to add the two values, forming the effective address of the branch on the result bus. This is latched back into the PC register during the second half of the execution cycle.

For branches with a long displacement, a single execution cycle is

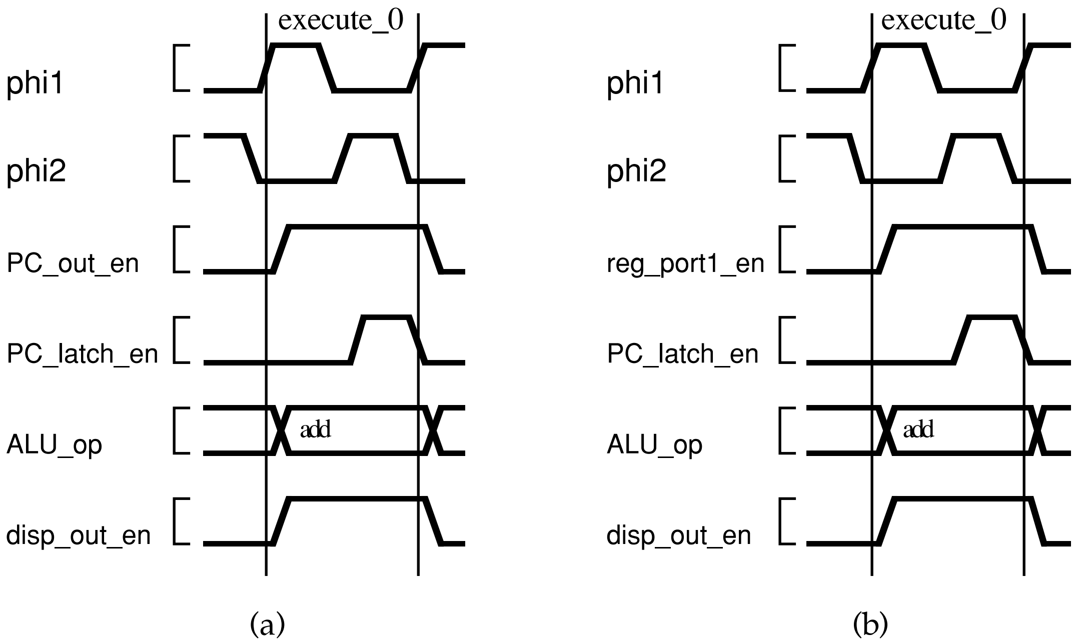

Figure 7-33. Execution of branch with branch taken.

always required. If the branch is not taken, the PC must be incremented to point to the instruction after the displacment. The timing for this is shown in Figure 7-32. The PC is enabled onto the op1 bus, and the ALU function is set to incr1. This increments the value and places it on the result bus. Then during the second half of the cycle, the new value is latched back into the PC register.

For long displacement branches where the branch is taken, the PC must be updated with the effective address. Figure 7-33(a) shows the timing for an ordinary branch, in which the PC is enabled onto the op1 bus. Figure 7-33(b) shows the timing for an indexed branch, in which the index register is enabled from register port 1 onto the op1 bus. The displacement register output is enabled onto the op2 bus, and the ALU function is set to add, to add the displacement to the base address, forming the effective address on the result bus. This is latched back into the PC register during the second half of the cycle.

The VHDL description implements the execution of a branch instruction as part of the nested case statement for the execute_0 state. The process checks the result bit from the condition code comparator. If it is set, the branch is taken, so the base address and displacement are enabled (depending on the type of branch), and the ALU function code set to add. Otherwise, if the condition code comparator result is clear, the branch is not taken. This should only be the case for long branches, since quick branches should never get to the execute_0 state. An assertion statement is used to verify this condition. For long branches which are not taken, the PC is enabled onto the op1 bus and the ALU function code set to incr1 to increment the value past the displacement in memory. The PC latch enable signal is then pulsed when phi2 changes to '1'. Finally, the next state is set to fetch_0, so the processor will continue with the next instruction.

The remaining instructions to be considered are the load and store instructions. These all take three cycles to execute, since a bus transaction is required to transfer the data to or from the memory. For long displacement loads and stores, the displacement has been previously fetched into the displacement register. For the quick forms, the immediate displacement in the instruction word is used.

Figure 7-34 shows the timing for execution of load and quick load instructions. The base address register is read from register file port 1 and enabled onto the op1 bus. For long displacement loads, the previously fetched displacement is enabled onto the op2 bus, and for quick loads, the sign extended immediate displacement is enabled onto the op2 bus. The ALU function code is set to add, to form the effective address on the result bus, and this is latched into the memory bus address register during the second half of the first execute cycle. During the next two cycles the controller performs a memory read transaction, with the fetch signal held negated. The data from the data bus is enabled onto the result bus through the connecting buffer, and latched into the register result latch. This value is then written back to the register file during the first cycle of the subsequent instruction fetch.

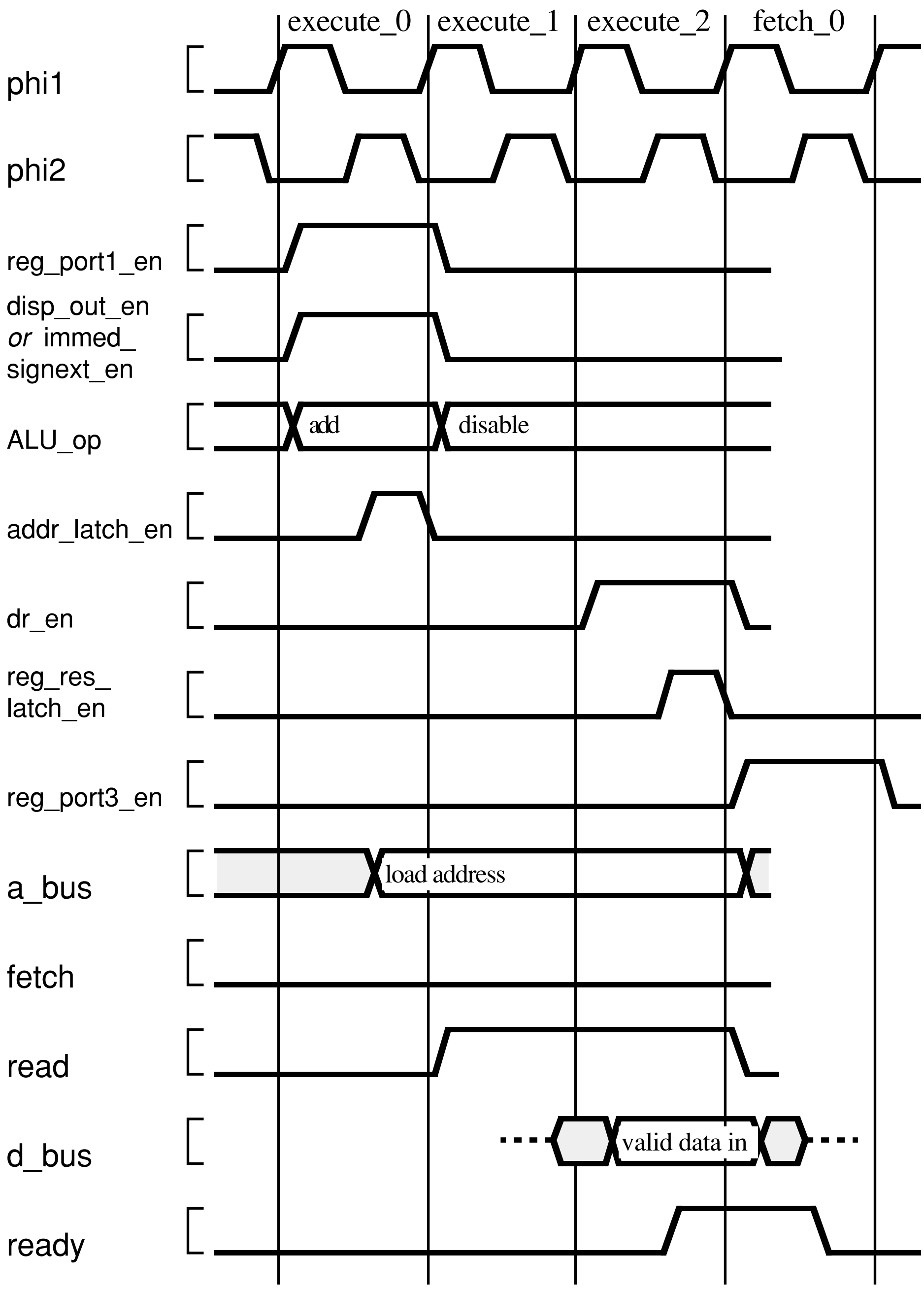

Figure 7-34. Execution of load instructions.

The timing for execution of store and quick store instructions is shown in Figure 7-35. As with load instructions, the base address and displacement are added, and the effective address is latched in the memory bus address register. During the next two cycles the controller performs a bus write transaction. The multiplexor for the register file port 2 address is set to select the r3 field of the instruction, which specifies the register to be stored, and the port 2 output is enabled onto the op2 bus. The buffer between the op2 bus and the memory data bus is enabled to transmit the data to the memory. Execution of the instruction completes at the end of the bus transaction.

Returning to the VHDL description, the first cycle of execution of load and store instructions is included as a branch of the nested case in the execute_0 state. The base address register output port is enabled, and either the displacement latch output or the sign extension buffer is enabled, depending on the instruction type. The ALU function code is set to add the two to form the effective address. The process then waits until phi2 changes to '1', indicating the second half of the cycle, and pulses the address latch enable. The next state is then set to execute_1 to continue execution of the instruction.

In state execute_1, the process firstly removes the base address, displacement and effective address from the DP32 internal buses, then starts a memory bus transaction. For load instructions, the fetch signal is negated and the read signal is asserted. For store instructions, the source register value is enabled onto the op2 bus, the memory data bus output buffer is enabled, and the write signal is aserted. The next state variable is then set to execute_2 for the next cycle.

In state execute_2, for load instructions, the memory data bus input buffer is enabled to transmit the data onto the result bus. The process then waits until phi2 is '1', in the second half of the cycle, and pulses the enable for the register result latch. The write back pending flag is then set to schedule the destination register write during the next instruction fetch cycle. For both load and store instructions, the next state is fetch_0. All control signals set during the execute_1 state will be returned to their negated values in the fetch_0 state.

The test bench described in Section 7.5 can be used to test the register transfer architecture of the DP32. This is done using an alternate configuration, replacing the behavioural architecture in the test bench with the register transfer architecture. Figure 7-36 shows such a configuration. The entity bindings for the clock generator and memory are the same, using the behavioural architectures, but the processor component instance uses the rtl architecture of the dp32 entity. This binding indication is followed by a configuration for that architecture, binding the entities described in Sections 7.6.1–7.6.9 to the component instances contained in the architecture. The newly configured description can be simulated using the same test programs as before, and the results compared to verify that they implement the same behaviour.

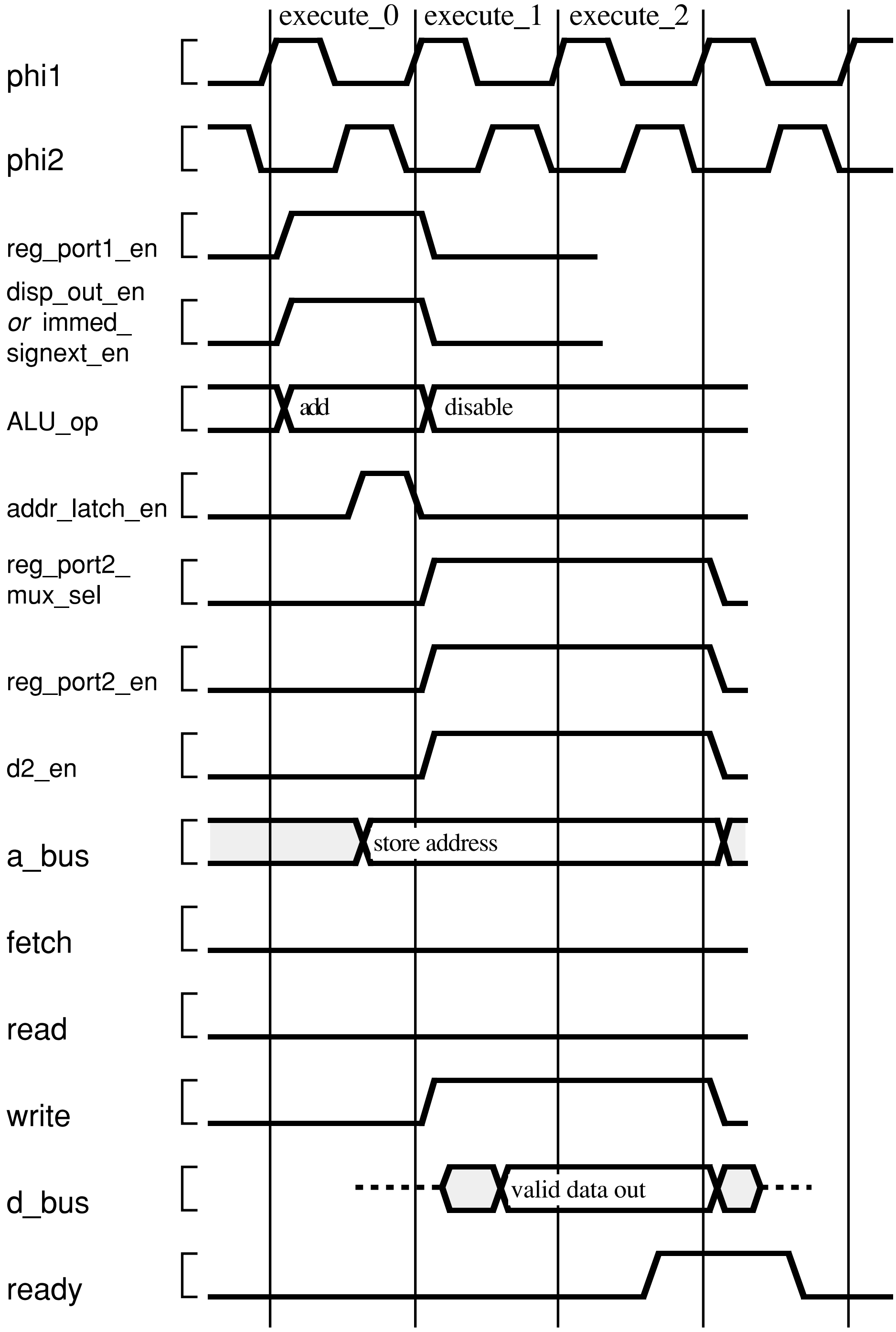

Figure 7-35. Execution of store instructions.

```
use work.dp32_types.all;

configuration dp32_rtl_test of dp32_test is

    for structure
        for cg : clock_gen
            use entity work.clock_gen(behaviour)
                generic map (Tpw => 8 ns, Tps => 2 ns);
        end for;
        for mem : memory
            use entity work.memory(behaviour);
        end for;
        for proc : dp32
            use entity work.dp32(rtl);
            for rtl
                for all : reg_file_32_rrw
                    use entity work.reg_file_32_rrw(behaviour);
                end for;
                for all : mux2
                    use entity work.mux2(behaviour);
                end for;
                for all : latch
                    use entity work.latch(behaviour);
                end for;
                for all : PC_reg
                    use entity work.PC_reg(behaviour);
                end for;
                for all : ALU_32
                    use entity work.ALU_32(behaviour);
                end for;
                for all : cond_code_comparator
                    use entity work.cond_code_comparator(behaviour);
                end for;
                for all : buffer_32
                    use entity work.buffer_32(behaviour);
                end for;
                for all : latch_buffer_32
                    use entity work.latch_buffer_32(behaviour);
                end for;
                for all : signext_8_32
                    use entity work.signext_8_32(behaviour);
                end for;
            end for;
        end for;
    end for;

end dp32_rtl_test;
```

Figure 7-36. Configuration using register transfer architecture of DP32.