Case Study I

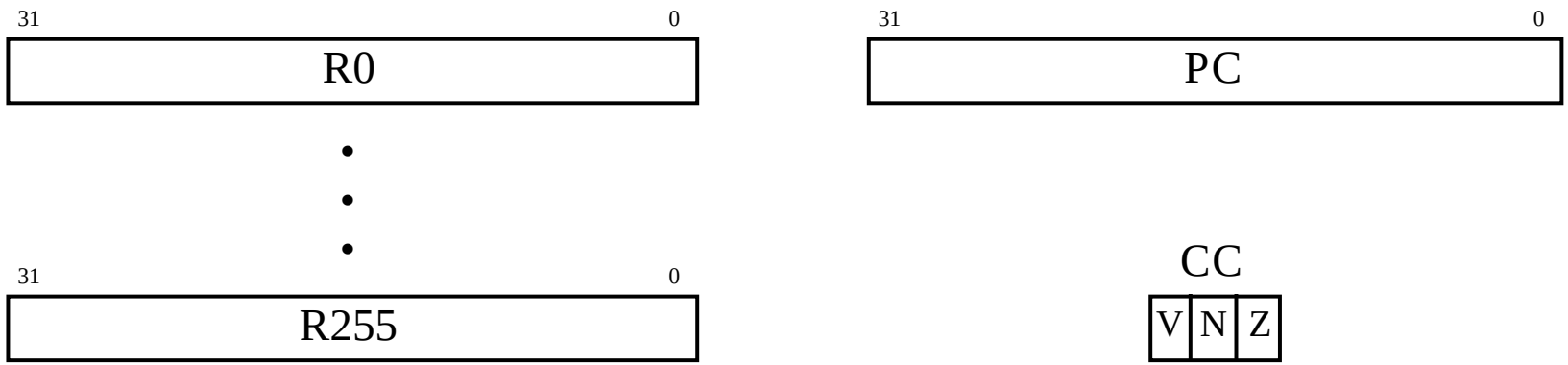
Prof. K. J. Hintz

Department of Electrical
and
Computer Engineering
George Mason University

DP-32 Processor, *e.g.*

- From Ashenden's *VHDL Cookbook*
- Complete Word Format Document on Web
  - VCBDP32.DOC
- Behavioral Description
- Structural Description in form of RTL
- Entity, Arch, Configuration, Pkgs, *etc*.

DP-32 Registers



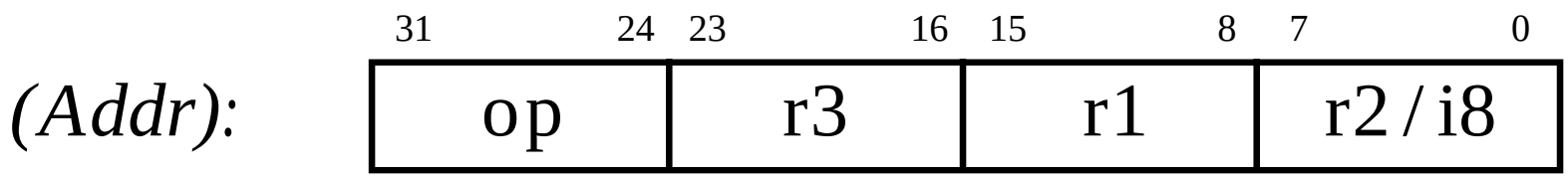
Arithmetic Instructions

- Many More Instructions

Instr	Name	Function	OpCode
ADD	Add	r3 ← r1+ r2	X"00"
SUB	Subtract	r3 ← r1 - r2	X"01"
MUL	Multiply	r3 ← r1 × r2	X"02"
DIV	Divide	r3 ← r1 ÷ r2	X"03"

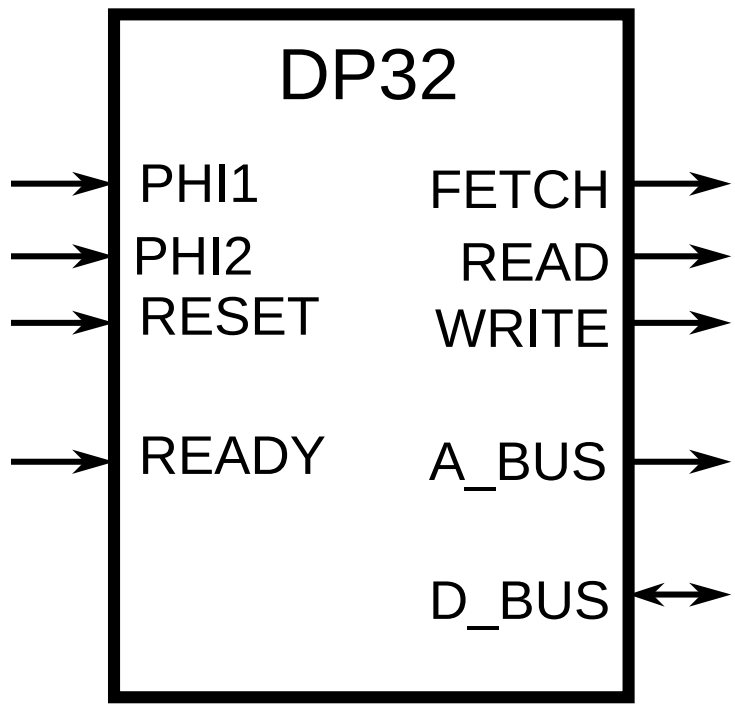
Typical 32-bit Instruction Format

- op: opcode
- r3: destination register
- r1: operand 1
- r2: operand 2 or,
  - i8: Immediate 2's complement operand

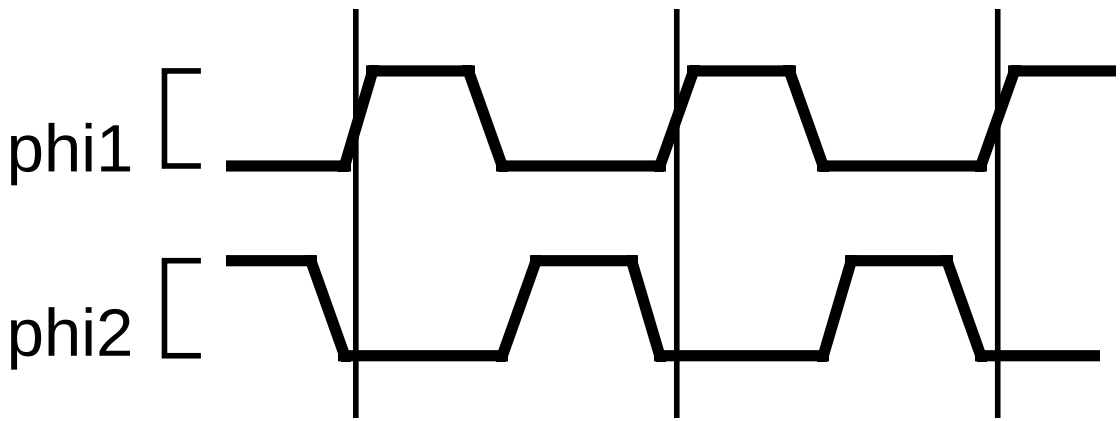


Port Diagram of DP32

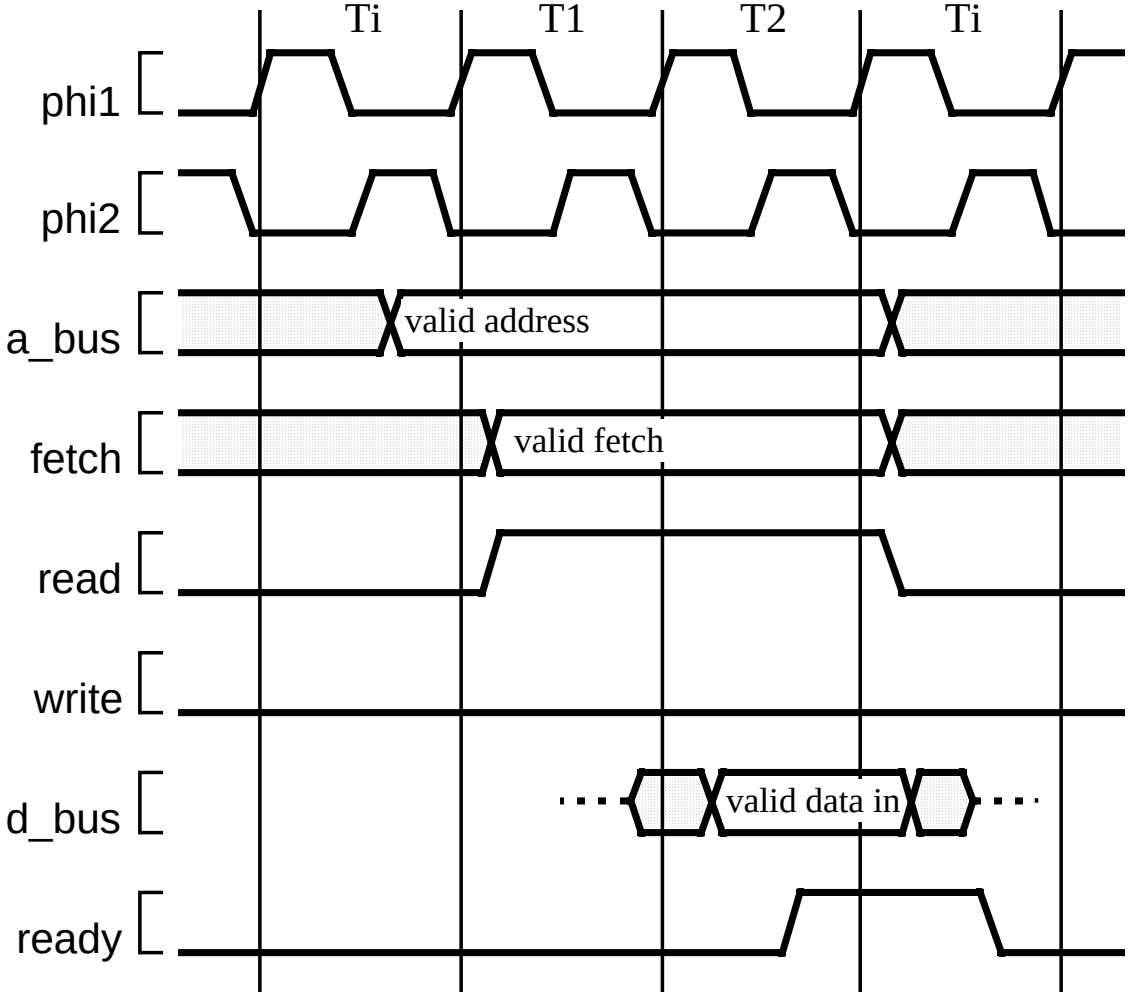
- 32-bit Address and Data Busses
- D_Bus is Bidirectional
- Fetch Means Instruction Being Read



Two-phase, Non-overlapping Clock



Bus Read Timing



DP32 Package

```
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 ;
```

DP32 Package

```
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 ;
```

DP32 Package

```
  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 ) ;
--condition code mask
  subtype cm_bits is
     bit_vector ( 3 downto 0 ) ;
```

DP32 Package

```
  constant op_add : bit_8 := X"00" ;
  constant op_sub : bit_8 := X"01" ;
  constant op_mul : bit_8 := X"02" ;

--rest of opcodes here
```

DP32 Package

```
--defined in package body
function bits_to_int
    ( bits : in bit_vector )
    return integer ;
function bits_to_natural
    ( bits : in bit_vector )
    return natural ;
```

DP32 Package

```
   procedure int_to_bits
      ( int  : in integer ;
        bits : out bit_vector );
end dp32_types ;
```

Package Body

```
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" ;
```

Package Body

```
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 ;
```

Package Body

```
function bits_to_int
 ( bits : in bit_vector )
  return integer is
--declarations
  variable temp : bit_vector( bits'range ) ;
  variable result : integer := 0 ;
```

Package Body

```
begin
 if bits(bits'left) = ‘1’
 then      -- negative number
  temp := not bits ;
 else
  temp := bits ;
 end if ;
```

Package Body

```
   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 ;
```

Package Body

```
function bits_to_natural
   (bits : in bit_vector)
return natural is
variable result : natural := 0 ;
begin
```

Package Body

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

Package Body

```
procedure int_to_bits
  ( int  : in integer;
    bits : out bit_vector) is
 variable temp : integer ;
 variable result : bit_vector(bits'range);
 begin
```

Package Body

```
if int < 0 then
 temp := -(int+1) ;
else
 temp := int ;
end if ;
```

Package Body

```
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 ;
```

Package Body

```
    bits := result ;
  end int_to_bits ;
end dp32_types ;
```

DP32 Entity

```
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 ;
```

DP32 Behavioral Arch

```
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;
```

DP32 Behavioral Arch

```
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;
```

DP32 Behavioral Arch

```
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;
```

DP32 Behavioral Arch

```
-- T1 phase

read <= '1' after Tpd ;
wait until phi1 = '1';
if reset = '1' then return ;
end if;
```

DP32 Behavioral Arch

```
  -- 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 ;
```

DP32 Behavioral Arch

```
  wait until phi1 = '1' ;
  if reset = '1' then return ;
  end if ;
--
-- Ti phase at end of cycle
--
  read <= '0' after Tpd ;
  end memory_read ;
```

DP32 Behavioral Arch

```
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 ;
```

DP32 Behavioral Arch

```
--
-- 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;
```

DP32 Behavioral Arch

```
--T2 phase
--
loop
wait until phi2 = '0';
if reset = '1' then return ;
end if ;
-- end of T2
exit when ready = '1' ;
end loop ;
```

DP32 Behavioral Arch

```
  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 ;
```

DP32 Behavioral Arch

```
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' ;
```

DP32 Behavioral Arch

```
 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 ;
```

DP32 Behavioral Subtract

```
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';
```

DP32 Behavioral Arch

```
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;
```

Other Operators

- Multiply...
- Divide...

DP32 Behav. Kernel--reset

```
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;
```

DP32 Behavioral Kernel

```
-- 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)) ;
```

DP32 Behavioral Kernel

```
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) );
```

DP32 Behav. Partial Decode

```
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_sub =>
  subtract( reg(r3), bits_to_int(reg(r1)),
            bits_to_int(reg(r2)),
            cc_V, cc_N, cc_Z);
```

DP32 Behav. Partial Decode

```
when op_ld => -- load operation
 memory_read( PC, true, displacement );
 if reset /= '1'
 then
  add( PC,
       bits_to_int(PC),
       1,
       temp_V,
       temp_N,
       temp_Z );
```

DP32 Behav. Partial Decode

```
    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 ;
```

DP32 Behav. Partial Decode

```
when op_st => -- store operation
 memory_read( PC,
              true,
              displacement);
 if reset /= '1' then
  add( PC,
       bits_to_int(PC),
       1,
       temp_V, temp_N, temp_Z );
```

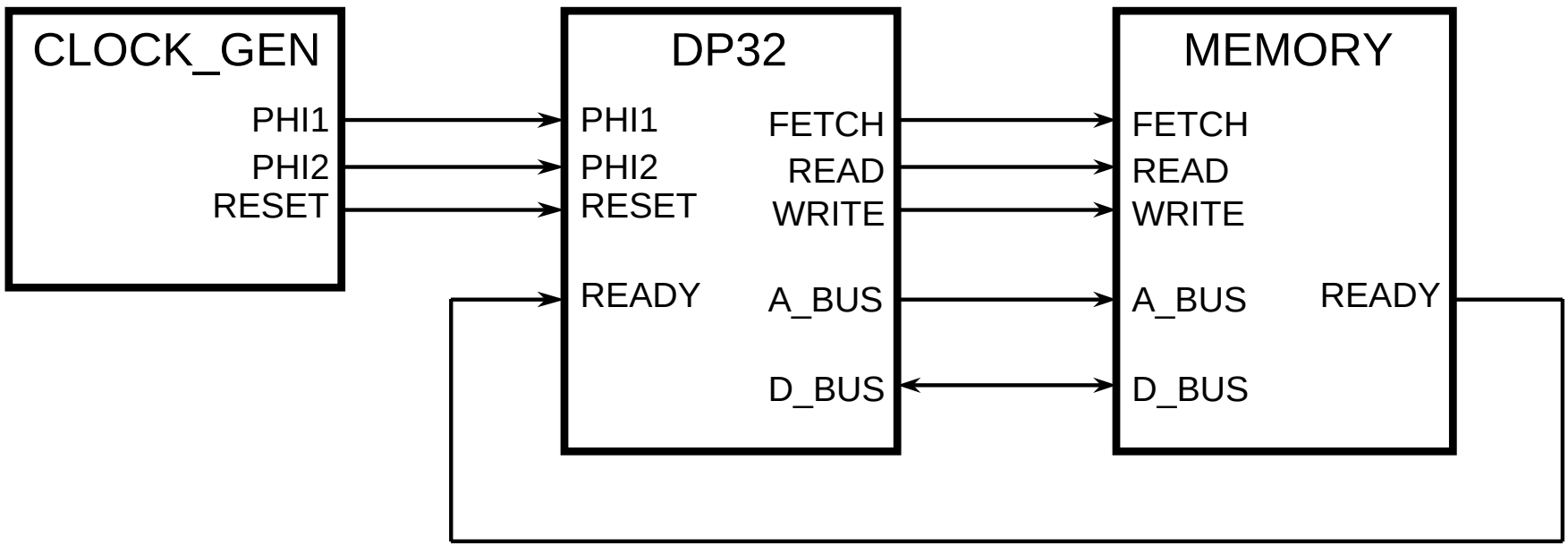
DP32 Behav. Partial Decode

```
    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 ;
```

DP32 Behav. Partial Decode

```
      when others =>
       assert false report "illegal
         instruction" severity warning ;
    end case ;
  end if ;-- reset /= '1'
 end process ;
end behaviour ;
```

DP32 Test Bench



DP32 Test Bench Entity

```
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 ;
```

DP32 Test Bench Arch.

```
architecture behaviour of clock_gen is
 constant clock_period :
  Time := 2*(Tpw+Tps);
begin
 reset_driver :
     reset <= '1',
              '0' after 2*clock_period + Tpw ;
```

DP32 Test Bench Arch.

```
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 ;
```

DP32 Memory Entity

```
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 ;
```

DP32 Memory Arch.

```
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 ;
```

DP32 Memory Arch

```
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');
```

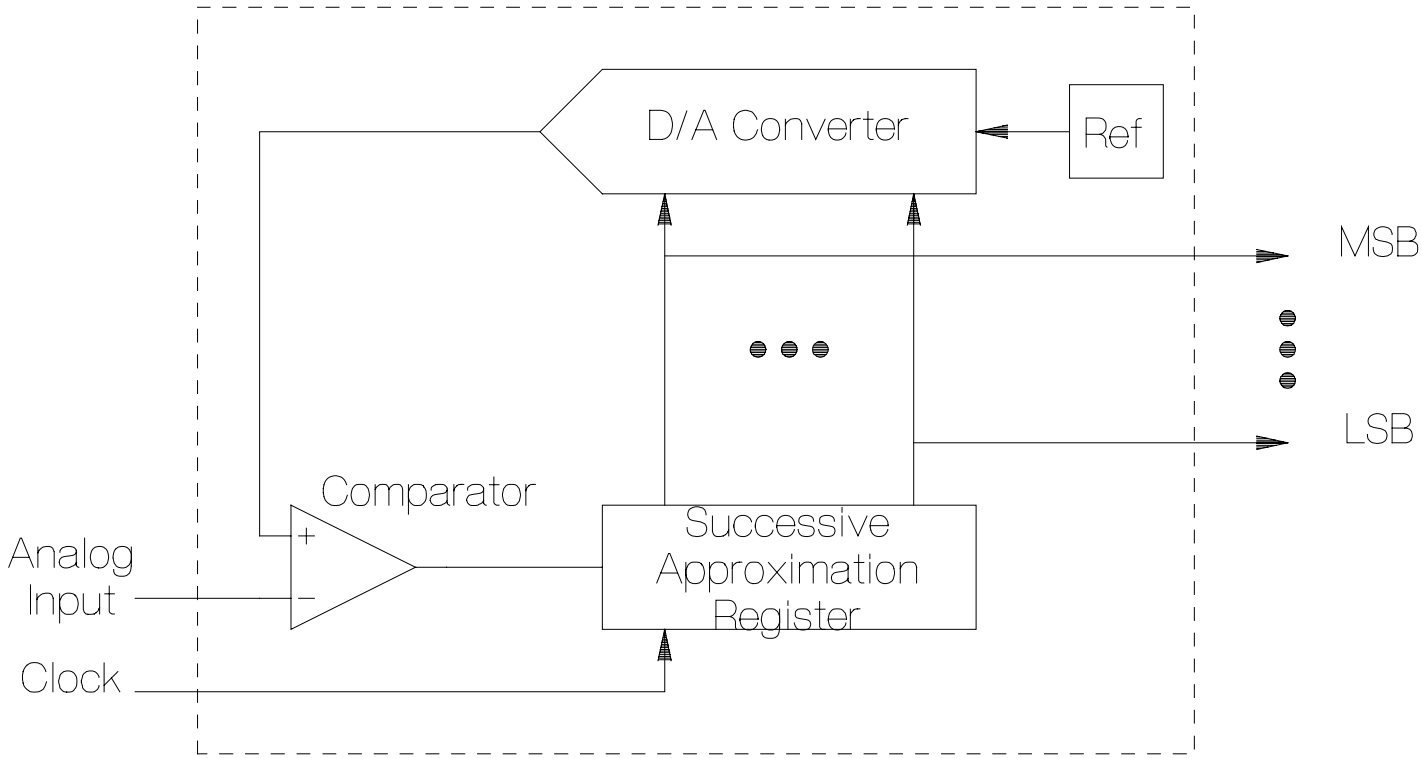
DP32 Memory Arch

```
-- 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) ;
```

DP32 Memory Arch

```
-- 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 ;
```

Successive Approximation A/D



S-A Algorithm

- Digital Approximations of Analog Value Are Supplied to D/A Converter
- Unknown Analog Value Is Compared in Analog Comparator With Analog Output of D/A Converter
- If Input Is Greater Than Approximation, Keep Approximation and Try Next Lowest Resolution

Assuming 8-bit conversion

First approximation

1000_0000 = $V_{ref}/2$

If analog input is greater,

then try 2nd approximation of 1100_0000

else

try 2nd approximation of 1010_0000

Approximations

- At each clock cycle, try a new digital approximation

$$V_n = V_{n-1} \pm \frac{V_{ref}}{2^{n+1}}$$

$$V_0 = \frac{V_{ref}}{2}$$

Negative-Valued Inputs

- Alternatives
  - Could assume bipolar D/A converter
  - Could add offset to unknown analog voltage equal to 1/2 of reference voltage
- Need to Convert From Binary Magnitude to 2'S Complement Output
- Alternatives Other Than These Are Acceptable

End of Lecture