```
            @(posedge rxc_)
            receivebuf={rcd, receivebuf[7:1]};
        end
        get_and_check_parity;   //receive and check parity bit, if any
        mark_char_received; //set rxrdy line, if enalbed
    end
endtask

always @(external_syndet_watche)
    @(posedge rxc_)
        status[6]=1;
            /****INTERNAL SYNCHRONOUS MODE RECEIVE ***/
            /*   Hunt for the sync char(s)            */
            /*   (if in synchronous internal sync detect mode) */
            /* Syndet is set high when the sync(s) are found */

always @ (hunt_sysnc1_e)//search for 1st sync char in the data stream
begin :sync_hunt_blk
    while(!(((receivebuf ^ sync1) & syncmask) === 8'b0000_0000))
    begin
        @(posedge rxc_)
        receivebuf = {rcd, receivebuf[7:1]};
    end
    if ( modreg[7]==0)  // if double sync mod
        ->hunt_sync2_e; //check for 2nd sync char directly agter 1
    else
        -> sync_hunted_e;   // if single sync mode , sync hunt is complete
end
always @ (hunt_sync2_e) // find the second synchronous character
begin : double_sync_hunt_blk
    repeat(databits)
    begin
        @(posedge rxc_)
        receivebuf={rcd,receivebuf[7:1]};
    end
    if((receivebuf ^ sync2)& syncmask===8'b0000_0000)
        ->sync_hunted_e; // if sync2 followed syn1,sync hunt is complete
    else
        ->hunt_sync1_e; //else hunt for sync1 again

    // Note : the data stream [sync1 sync1 sync2] will have sync detected.
    // Suppose sync1=11001100:
    // Then [1100 1100 1100 sync2]will NOT be detected .
    // In general : never let a suffix of sync1 also be a prefix of sync1.
end
```

```
always @ (sync_hunted_e)
begin :parity_sync_hunt_blk
    get_and_check_parity;
    status[6]=1;//set syndet status bit (sync chars detected )
end

task syn_receive_internal;
forever
begin
    repeat(databits)//no longer in hunt mode so read entire chars and
    begin             // then look for syncs (instead of on bit boundaries)
        @(posedge rxc_)
        receivebuf={rcd,receivebuf[7:1]};
    end
    case (sync_to_receive)
    2:          // if looking for 2nd sync char ...
    begin
        if(((receivebuf ^ sync2) & syncmask)===0)
        begin   //... and 2nd sync char is found
            sync_to_receive =1; //then look ofr 1st sync (or data)
            status[6]=1;    // and mark sync detected
        end
        else if (((receivebuf ^ sync1) & syncmask)===0)
        begin   //... and 1st sync char is found
            sync_to_receive = 2;//then look for 2nd sync char
        end
        end
    1:
    begin
        if ((( receivebuf ^ sync1) & syncmask) ===0) // ... and 1st sync is found
        begin
            if(modreg[7]==0)//if doulbe sync mode
                sync_to_receive =2; // look for 2nd sync to foll
            else
                status[6]=1;    //else look for 1st or data and mark sync detected
        end
        else;   //and data was found , do nothing
    end
    endcase
    get_and_check_parity;   // receive and check parity bit, if any
    mark_char_received;
end
endtask
```

```
//*********************************************************
task syn_receive_external;
forever
begin
// have not found the original programs
end
endtask

task get_and_check_parity;
begin
    receivebuf=receivebuf >> (8-databits);
    if(modreg[4] == 1)
    begin
        @(posedge rxc_)
        if (( ^receivebuf ^ modreg[5] ^ rcd) != 1)
            parity_error;
    end
end
endtask

task mark_char_received;
begin
    if(command[2]==1)   // if receiving is enabled
    begin
    rxrdy=1;//set receive read status bit
    status[1]=1;//if previous data was not read
    if(rdatain == 1)
        overrun_error; // overrun error
    rdata=receivebuf;   //latch the data
    rdatain=1;  //mark data as not having been read
    end
if(dflags[2])
    $display("I8251A  (%h)  at  %d  :  receive  data  :  %b",  instance_id,
$time,receivebuf);
end
endtask


/************* ASYNCHRONOUS MODE RECEIVER ***************/
/* CHECK FOR BREAK DETECTION (RCD LOW THROUGH 2 */
/* RECEIVE SEQUENCES IN THE ASYNCHRONOUS MODE .*/

always @ (break_detect_e)
begin :break_detect_blk
```

```
    #1 /* to be sure break_delay_clk is waiting on break_delay_e
        after it triggered break_detect_e */
    if (rcd==0)
    begin
        ->break_delay_e; // start + databits +parity +stop bit
        breakcount_period = 1 +databits + modreg[4] + (tstoptotal!=0);
        // the number of rxc periods needed for 2 receive sequence
        breakcount_period  = 2* breakcount_period*baudmx;
        //if rcd stays low through 2 consecutive
        // (start ,data,prity ,stop ) sequences ...
        repeat(breakcount_period)
            @(posedge rxc_);
        status[6]=1;// ... then set break detect (status[6]) high
    end
end

always @(break_delay_e)
begin : break_delay_blk
    @(posedge rcd ) //but if rcd goes high during that time
    begin :break_delay_blk
        disable break_detect_blk;
        status[6] = 0;  //... then set the break detect low
        @(negedge rcd ) //and when rcd goes low again ...
        ->break_detect_e;   // ... start the break detection again
    end
end

/******** ASYNCHRONOUS MODE RECEIVE TASK *****************/
task asyn_receive;
forever
    @(negedge rcd) // the receive line went to zero, maybe a start bit
    begin
        rbaudcnt = baudmx /2;
        if (baudmx == 1)
            rbaudcnt=1;
        repeat(rbaudcnt) @(posedge rxc_); // after half a bit ...
        if(rcd == 0)//if it is still a start bit
        begin
            rbaudcnt = baudmx;
            repeat(databits) // receive the data bits
            begin
                repeat(rbaudcnt ) @(posedge rxc_);
                #1 receivebuf={rcd,receivebuf[7:1]};
            end
            repeat (rbaudcnt) @(posedge rxc_);
```

```
                //shift the data to the low part
            receivebuf = receivebuf >> (8-databits);
            if(modreg[4]==1)///if parity is enabled
            begin
                if ((^receivebuf ^ modreg[5]^rcd)!=1)
                    parity_error;   //check for a parity error
                repeat(rbaudcnt) @(posedge rxc_);
            end

            #1 if (rcd == 0 )     // if middle of stop bit is 0
                frame_error;// frame error (should be 1)

            mark_char_received;
        end
    end
endtask
endmodule
```

[例 2]. “商业化”的虚拟模块之二: Intel 8085a 微处理器的行为描述模块

```
/*****************************************************************************
                  Intel 8085a 微处理器仿真模块的Verilog源代码
          注意：作者不能保证本模块的完整和精确，使用本模块者如遇问题一切责任自负
*****************************************************************************/

module intel_8085a
        (clock, x2, resetff, sodff, sid, trap, rst7p5, rst6p5, rst5p5,
         intr, intaff, ad, a, s0, aleff, writeout, readout, s1,iomout,
         ready, nreset, clockff, hldaff, hold);

    reg [8:1]      dflags;
    initial         dflags = 'b011;
    // diag flags:
    // 1 = trace instructions
    // 2 = trace IN and OUT instructions
    // 3 = trace instruction count

    output
        resetff, sodff, intaff, s0, aleff,
        writeout, readout, s1, iomout, clockff, hldaff;

    inout[7:0] ad, a;
```

```
input
        clock, x2, sid, trap,
        rst7p5, rst6p5, rst5p5,
        intr, ready, nreset, hold;

reg[15:0]
    pc,        // program counter
    sp,        // stack pointer
    addr;      // address output

reg[8:0]
    intmask;   // interrupt mask and status

reg[7:0]
    acc,       // accumulator
    regb,      // general
    regc,      // general
    regd,      // general
    rege,      // general
    regh,      // general
    regl,      // general
    ir,        // instruction
    data;      // data output

reg
    aleff,     // address latch enable
    s0ff,      // status line 0
    s1ff,      // status line 1
    hldaff,    // hold acknowledge
    holdff,    // internal hold
    intaff,    // interrupt acknowledge
    trapff,    // trap interrupt request
    trapi,     // trap execution for RIM instruction
    inte,      // previous state of interrupt enable flag
    int,       // interrupt acknowledge in progress
    validint,  // interrupt pending
    haltff,    // halt request
    resetff,   // reset output
    clockff,   // clock output
    sodff,     // serial output data
    read,      // read request signal
    write,     // write request signal
    iomff,     // i/o memory select
    acontrol,  // address output control
    dcontrol,  // data output control
    s,         // data source control
    cs,        // sign condition code
    cz,        // zero condition code
    cac,       // aux carry condition code
    cp,        // parity condition code
    cc;        // carry condition code

wire
    s0 = s0ff & ~haltff,
    s1 = s1ff & ~haltff;

tri[7:0]
    ad = dcontrol ? (s ? data : addr[7:0]) : 'bz,
    a = acontrol ? addr[15:8] : 'bz;

tri
    readout = acontrol ? read : 'bz,
    writeout = acontrol ? write : 'bz,
    iomout = acontrol ? iomff : 'bz;

event
    ec1, // clock 1 event
    ec2; // clock 2 event

// internal clock generation
always begin
    @(posedge clock) -> ec1;
    @(posedge clock) -> ec2;
end

integer instruction; // instruction count
initial instruction = 0;

always begin:run_processor
    #1 reset_sequence;
    fork
        execute_instructions;             // Instructions executed
        wait(!nreset)                     // in parallel with reset
            @ec2 disable run_processor; // control. Reset will
    join                                  // disable run_processor
end                                       // and all tasks and
                                          // functions enabled from
                                          // it when nreset set to 0.
```

```
task reset_sequence;
begin
    wait(!nreset)
    fork
        begin
            $display("Performing 8085(%m) reset sequence");
            read = 1;
            write = 1;
            resetff = 1;
            dcontrol = 0;
            @ec1 // synchronized with clock 1 event
                pc = 0;
                ir = 0;
                intmask[3:0] = 7;
                intaff = 1;
                acontrol = 0;
                aleff = 0;
                intmask[7:5] = 0;
                sodff = 0;
                trapff = 0;
                trapi = 0;
                iomff = 0;
                haltff = 0;
                holdff = 0;
                hldaff = 0;
                validint = 0;
                int = 0;
            disable check_reset;
        end
        begin:check_reset
            wait(nreset)              // Check, in parallel with the
                disable run_processor; // reset sequence, that nreset
        end                           // remains at 0.
    join
    wait(nreset) @ec1 @ec2 resetff = 0;
end
endtask


/* fetch and execute instructions */
task execute_instructions;
forever begin
    instruction = instruction + 1;
    if(dflags[3])
        $display("executing instruction %d", instruction);
```

```
@ec1 // clock cycle 1
    addr = pc;
    s = 0;
    iomff = 0;
    read = 1;
    write = 1;
    acontrol = 1;
    dcontrol = 1;
    aleff = 1;
    if(haltff) begin
        haltff = 1;
        s0ff = 0;
        s1ff = 0;
        haltreq;
    end
    else begin
        s0ff = 1;
        s1ff = 1;
    end
@ec2
    aleff = 0;

@ec1 // clock cycle 2
    read = 0;
    dcontrol = 0;
@ec2
    ready_hold;

@ec2 // clock cycle 3
    read = 1;
    data = ad;
    ir = ad;

@ec1 // clock cycle 4
    if(do6cycles(ir)) begin
        // do a 6-cycle instruction fetch
        @ec1 @ec2 // conditional clock cycle 5
            if(hold) begin
                holdff =1 ;
                acontrol = 0;
                dcontrol = 0;
                @ec2 hldaff = 1;
            end
            else begin
```

```
                    holdff = 0;
                    hldaff = 0;
                end

            @ec1; // conditional clock cycle 6
        end

        if(holdff) holdit;
        checkint;
        do_instruction;

        while(hold) @ec2 begin
            acontrol = 0;
            dcontrol = 0;
        end
        holdff = 0;
        hldaff = 0;
        if(validint) interrupt;
end
endtask


function do6cycles;
input[7:0] ireg;
begin
    do6cycles = 0;
    case(ireg[2:0])
        0, 4, 5, 7: if(ireg[7:6] == 3) do6cycles = 1;
        1: if((ireg[3] == 1) && (ireg[7:5] == 7)) do6cycles = 1;
        3: if(ireg[7:6] == 0) do6cycles = 1;
    endcase
end
endfunction


task checkint;
begin
    if(rst6p5)
        if((intmask[3] == 1) && (intmask[1] == 0)) intmask[6] = 1;
    else
        intmask[6] = 0;

    if(rst5p5)
        if((intmask[3] == 1) && (intmask[0] == 0)) intmask[5] = 1;
    else
```