Process

- The behavior of digital systems can be conceived as a set of independent, but communicating processes.
- A process can be thought of as an independent thread of control, which may be quite simple, involving only one repeated action, or very complex, resembling a software program.
- A process might be implemented as a sequential state machine, as a microcoded controller, or as an asynchronous clearing of a register.

Process

- The ***initial*** and ***always*** statements are the basic constructs for describing concurrency — concurrently active processes that will interact with each other.
- The always statement continuously repeats its statements, never exiting or stopping.
- The initial statement is similar to the always statement except that it is executed only once.
- Although it is possible to mix the description of behavior between the always and initial statements, it is more appropriate to maintain the separation.
  - Behavior of the hardware is described in the always.
  - Initialization for the simulation is specified in the initial.

Process

- At the start of the simulation, all of the initial and always statements are allowed to execute.
- When the simulator executes an event statement (@), a delay statement (#), or a wait statement where the expression is FALSE, the execution of the initial or always statement is suspended until the event occurs, the number of time units indicated in the delay has passed, or the wait statement expression becomes TRUE.
- Then, execution of statements in the initial or always statement continues.

If-Then-Else Statement

- A divide module using an iterative subtract and shift algorithm

```
`define DvLen 15 // Text macro definition
`define DdLen 31
`define QLen 15
`define HiDdMin 16

module divide (ddInput, dvInput, quotient, go, done);
input [`DdLen:0]        ddInput;
input [`DvLen:0]        dvInput;
input                   go;
```

If-Then-Else Statement

```
output [`QLen:0]        quotient;
output                  done;

reg [`QLen:0]           quotient;
reg                     done;

reg [`DdLen:0]          dividend;
reg [`DvLen:0]          divisor;
reg                     negDivisor, negDividend;

initial
        done = 0; // If TRUE, it signifies that the quotient is valid.
```

If-Then-Else Statement

```
always
begin
        wait (go); // If TRUE, it signifies that the dvInput and ddInput are valid.
        divisor = dvInput;
        dividend = ddInput;
        quotient = 0;
        if (divisor) // If not zero, follow the normal divide algorithm.
        begin
                negDivisor = divisor[`DvLen]; // Assumes 2's complement.
                if (negDivisor) divisor = - divisor; // Takes the absolute value.
                negDividend = dividend[`DdLen];
                if (negDividend) dividend = - dividend;
```

If-Then-Else Statement

```
repeat (`DvLen + 1) // Executes the begin-end block 16 times.
begin
        quotient = quotient << 1; // Shifts left one position.
        dividend = dividend << 1;
        dividend[`DdLen:`HiDdMin] = // Top part-select
                dividend[`DdLen:`HiDdMin] - divisor;
        if (!dividend[`DdLen]) // If dividend is positive
                quotient = quotient + 1;
        else // If dividend is negative
                dividend[`DdLen:`HiDdMin] =
                dividend[`DdLen:`HiDdMin] + divisor;
end
```

If-Then-Else Statement

```
                    if (negDivisor != negDividend) // If the signs are different
                        quotient = - quotient;
        end
        done = 1; // The quotient is now valid.
        wait (~go);
        done = 0;
end
endmodule
```

If-Then-Else Statement

- Text macros
  - The ***`define*** compiler directive provides a macro capability by defining a name and giving a constant textual value to it.
  - The name may then be used in the description.
  - On compilation, the text value will be substituted.

- Bit-select and part-select
  - Bit-select : identifier[expression]
  - Part-select : identifier[msb_expression:lsb_expression]
  - The indices of the bit-select and part-select may be positive or negative.

If-Then-Else Statement

- Relational operators
  - > (greater than), >= (greater than or equal), = = (equal), != (not equal)
  - In the case where x or z values are present, the comparisons are ambiguous and considered to be FALSE by the simulator.
  - 4’b110z = = 4’b110z : FALSE

- Case operators
  - = = = (equal), != = (not equal)
  - Can be used to specify that individual x or z bits are to take part in the comparison.
  - 4’b110z = = = 4’b110z : TRUE

If-Then-Else Statement

- Where does the *else* belong ?
  - Case 1 : The *else* is paired with the first *if*.

```
if (expressionA)
begin
        if (expressionB)
                a = a + b;
end
else
        q = r + s;
```

If-Then-Else Statement

- Where does the *else* belong ?
  - Case 2 : The *else* is paired with the second *if*.

```
if (expressionA)
        if (expressionB)
                a = a + b;
        else
                q = r + s;
```

  - When in doubt about where the *else* will be attached, use begin-end pairs to make it clear.

If-Then-Else Statement

- The conditional operator (?:)
  - Can be used in place of the *if* statement when one of two values is to be selected for assignment.

```
if (negDivisor != negDividend) quotient = - quotient;
```

↓

```
qutoient = (negDivisor != negDiviend) ? - quotient : quotient;
```

  - The conditional operator may appear in an expression that is either part of a procedural or continuous assignment statement.
  - The if-then-else construct is a statement that may appear only in the body of an initial or always statement, or in a task or function.

Loops

- The ***repeat*** loop
  - The general form:

    repeat (expression) statement
  - The value of loop count is determined once at the beginning of the loop.
  - The loop is executed the given number of times.
  - It is not possible to exit the loop execution by changing the loop count variable.
  - Example:

```
repeat (16)
begin
        // Statements
end
```

Loops

- The ***for*** loop
  - The general form:

    for (reg_assignment; expression; reg_assignment) statement
  - The first assignment is executed once at the beginning of the loop.
  - The expression is evaluated before the body of the loop to check the end.
  - The second assignment is executed after the body of the loop and before the next check for the end of the loop.
  - Example:

```
for (i = 16; i; i = i - 1)
begin
        // Statements
end
```

Loops

- The ***while*** loop
  - The general form:

    while (expression) statement
  - The expression is evaluated and if it is TRUE, the statement is executed.
  - We enter and stay in the loop while the expression is TRUE.
  - Example:

```
i = 16;
while (i)
begin
        // Statements
        i = i - 1;
end
```

Loops

- The ***while*** loop
  - Example:

```
module sureDeath (inputA); // This will not work !
input       inputA;


always
begin
            while (inputA)
                        ; // Wait for external variable.
            // Other statements
end
endmodule
```

Loops

- The ***while*** loop
  - The while statement expression in this example is dependent on the value of *inputA* and the while statement is null.
  - Hence, this while statement appears to have the effect of doing nothing until the value of *inputA* is TRUE.
  - However, since we are waiting for an external value to change, the correct statement to use in this example is the *wait*.

- The ***forever*** loop
  - The general form:

    forever statement
  - The forever loop loops forever.

Loops

- The ***forever*** loop
  - Example:

```
module microprocessor;
always
begin
        powerOnInitializations;
        forever
        begin
                fetchAndExecuteInstructions;
        end
end
endmodule
```

Loops

- The ***disable*** statement
  - Disables or terminates any named begin-end block.
  - Example:

```
begin : break
        for (i = 0; i < n; i = i + 1)
        begin : continue
                if (a = = 0)
                        disable continue; // Proceed with i = i + 1.
                if (a = = b)
                        disable break; // Exit for loop.
        end
end
```

Multi-Way Branching

- The Mark-1 processor with the ***if-else-if*** statement

```
module mark1;
reg [31:0]      m [0:8191];     // 8192 x 32 bit memory
reg [12:0]      pc;             // 13-bit program counter
reg [31:0]      acc;            // 32-bit accumulator
reg [15:0]      ir;             // 16-bit instruction register

always
begin
        ir = m[pc]; // Fetch an instruction.
```

Multi-Way Branching

```
        // Decoding and executing
        if (ir[15:13] = = 3'b000) pc = m[ir[12:0]];
        else if (ir[15:13] = = 3'b001) pc = pc + m[ir[12:0]];
        else if (ir[15:13] = = 3'b010) acc = -m[ir[12:0]];
        else if (ir[15:13] = = 3'b011) m[ir[12:0]] = acc;
        else if ((ir[15:13] = = 3'b101) || (ir[15:13] = = 3'b100))
                acc = acc - m[ir[12:0]];
        else if (ir[15:13] = = 3'b110)
                if (acc < 0) pc = pc + 1;
        #1 pc = pc + 1; // Increment program counter and time.
end
endmodule
```

Multi-Way Branching

- The Mark-1 processor with the ***case*** statement

```
module mark1Case;
reg [31:0]      m [0:8191];     // 8192 x 32 bit memory
reg [12:0]      pc;             // 13-bit program counter
reg [31:0]      acc;            // 32-bit accumulator
reg [15:0]      ir;             // 16-bit instruction register

always
begin
        ir = m[pc]; // Fetch an instruction.
```

Multi-Way Branching

```
        case (ir[15:13]) // Decoding and executing
                3'b000 : pc = m[ir[12:0]];
                3'b001 : pc = pc + m[ir[12:0]];
                3'b010 : acc = -m[ir[12:0]];
                3'b011 : m[ir[12:0]] = acc;
                3'b100,
                3'b101 : acc = acc - m[ir[12:0]];
                3'b110 : if (acc < 0) pc = pc + 1;
        endcase
        #1 pc = pc + 1; // Increment program counter and time.
end
endmodule
```

Multi-Way Branching

- Comparison of *case* and *if-else-if*
  - Since all of the expressions were compared with one controlling expression, the case is sometimes more compact.
  - The conditional expressions in the if-else-if construct are more general.
  - The comparison for the case is done using 4-valued logic and will succeed only when each bit matches exactly w.r.t. the values 0, 1, x, and z.
  - In contrast, if statement expressions involving x or z values may result in x or z value which will be interpreted as FALSE (unless case equality is used).

Multi-Way Branching

- The case statement example with x and z values

```
reg   ready;

// Other statements
case (ready)
        1'bz        : $display ("ready is high impedance");
        1'bx        : $display ("ready is unknown");
        default     : $display ("ready is %b", ready);
endcase
```

Multi-Way Branching

- The ***casez*** and ***casex*** statements
  - Casez allows for z values to be treated as don't care values.
  - Casex allows for both z and x values to be treated as don't care values.
  - Example:

```
module decode;
reg [7:0]  r, mask;

always
begin
        // Other statements
        mask = 8'bx0x0x0x0;
        casex (r ^ mask)
```

Multi-Way Branching

```
                    8'b001100xx : statement1;
                    8'b1100xx00 : statement2;
                    8'b00xx0011 : statement3;
                    8'bxx001100 : statement4;
            endcase
end
endmodule
```

r = 8'b01100110

→ r ^ mask = 8`bx1x0x1x0

→ Match 8`b1100xx00 and 8`bxx001100.

→ Statement2 will be executed because it was found first.

Functions and Tasks

- Modules versus functions and tasks
  - Functions and tasks are the constructs analogous to software functions and procedures that allow for the behavioral description of a module to be broken into more-manageable parts.
  - Modules break a design up into more manageable parts.
  - The use of modules implies that there are structural boundaries being described.

Functions and Tasks

- Why functions and tasks are useful ?
  - Allow often-used behavioral sequences to be written once and called when needed.
  - Allow for a cleaner writing style — instead of long sequences of behavioral statements, the sequences can be broken into more readable pieces, regardless of whether they are called one or many times.
  - Allow for data to be hidden from other parts of the design.

Functions and Tasks

- Comparison of functions and tasks
  - Enabling (calling)
    - Because a task call is a separate procedural statement, it cannot be called from a continuous assignment statement.
    - Because a function call is an operand in an expression, it is called from within the expression and returns a value used in the expression.
    - Functions may be called from within procedural and continuous assignment statements.
  - Inputs and outputs
    - A task can have zero or more arguments of any type.
    - A function has at least one input, but does not have inouts or outputs.

Functions and Tasks

- Comparison of functions and tasks
  - Timing and event controls (#, @, and wait)
    - A task can contain timing and event control statements.
    - Functions may not contain these statements.
  - Enabling other tasks and functions
    - A task may enable other tasks and functions.
    - A function can enable other functions but not other tasks.
  - Values returned
    - A task does not return a value, but values written into its inout or output ports are copied back at the end of the task execution.
    - A function returns a single value, and the value to be returned is assigned to the function identifier within the function.

Functions and Tasks

- The Mark-1 processor with a multiply instruction

```
module mark1Mult;
reg [31:0]      m [0:8191];     // 8192 x 32 bit memory
reg [12:0]      pc;             // 13-bit program counter
reg [31:0]      acc;            // 32-bit accumulator
reg [15:0]      ir;             // 16-bit instruction register

always
begin : executeInstructions
        ir = m[pc]; // Fetch an instruction.
        case (ir[15:13]) // Decoding and executing
```

Functions and Tasks

```
                3'b000 : pc = m[ir[12:0]];
                3'b001 : pc = pc + m[ir[12:0]];
                3'b010 : acc = -m[ir[12:0]];
                3'b011 : m[ir[12:0]] = acc;
                3'b100,
                3'b101 : acc = acc - m[ir[12:0]];
                3'b110 : if (acc < 0) pc = pc + 1;
                3'b111 : acc = acc * m[ir[12:0]]; // Multiply instruction added.
        endcase
        #1 pc = pc + 1; // Increment program counter and time.
end
endmodule
```

Functions and Tasks

- The Mark-1 processor with a ***task***

```
module mark1Task;
reg [31:0]      m [0:8191];     // 8192 x 32 bit memory
reg [12:0]      pc;             // 13-bit program counter
reg [31:0]      acc;            // 32-bit accumulator
reg [15:0]      ir;             // 16-bit instruction register

always
begin : executeInstructions
        ir = m[pc]; // Fetch an instruction.
```

Functions and Tasks

```
case (ir[15:13]) // Decoding and executing
        3'b000 : pc = m[ir[12:0]];
        3'b001 : pc = pc + m[ir[12:0]];
        3'b010 : acc = -m[ir[12:0]];
        3'b011 : m[ir[12:0]] = acc;
        3'b100,
        3'b101 : acc = acc - m[ir[12:0]];
        3'b110 : if (acc < 0) pc = pc + 1;
        3'b111 : multiply (acc, m[ir[12:0]]); // Task call
endcase
#1 pc = pc + 1; // Increment program counter and time.
end
```

Functions and Tasks

```
task multiply;
inout [31:0]      a;
input [31:0]      b;

begin : serialMult
        reg [15:0]          mcnd, mpy;          // Multiplicand and multiplier
        reg [31:0]          prod;               // Product

        mpy = b[15:0];  // Part-select on b to load low-order 16 bits into mpy
        mcnd = a[15:0]; // Part-select on a to load low-order 16 bits into mcnd
        prod = 0;
```

Functions and Tasks

```
        repeat (16)
        begin
                if (mpy[0]) // If the low-order bit of mpy is one, concatenate.
                        prod = prod + {mcnd, 16`h0000};
                prod = prod >> 1; // prod shifted right one position
                mpy = mpy >> 1;  // mpy shifted right one position
        end
        a = prod;
end
endtask
endmodule
```

Functions and Tasks

- Parameters in tasks
  - The input, output, and inout names declared in tasks are local variables, whose scope is the task-endtask block.
  - When a task is called, the internal variables declared as inputs or inouts receive copies of the values named as the calling site.
  - When the execution of the task is done, all of the variables declared as inouts or outputs are copied back to the variables listed at the call site.
  - In this example:
    - When *multiply* is called, *acc* is copied into *a*, the value read from memory is loaded into *b*, and the task proceeds.
    - When the task is ready to return, *prod* is loaded into *a*.
    - On return, *a* is then copied back into *acc*.

Functions and Tasks

- The Mark-1 processor with a ***function***

```
module mark1Fun;
reg [31:0]      m [0:8191];     // 8192 x 32 bit memory
reg [12:0]      pc;             // 13-bit program counter
reg [31:0]      acc;            // 32-bit accumulator
reg [15:0]      ir;             // 16-bit instruction register

always
begin : executeInstructions
        ir = m[pc]; // Fetch an instruction.
```

Functions and Tasks

```
        case (ir[15:13]) // Decoding and executing
                3'b000 : pc = m[ir[12:0]];
                3'b001 : pc = pc + m[ir[12:0]];
                3'b010 : acc = -m[ir[12:0]];
                3'b011 : m[ir[12:0]] = acc;
                3'b100,
                3'b101 : acc = acc - m[ir[12:0]];
                3'b110 : if (acc < 0) pc = pc + 1;
                3'b111 : acc = multiply (acc, m[ir[12:0]]); // Function call
        endcase
        #1 pc = pc + 1; // Increment program counter and time.
    end
```

Functions and Tasks

```
function [31:0] multiply;
input [31:0]      a;
input [31:0]      b;

begin : serialMult
        reg [15:0]          mcnd, mpy;          // Multiplicand and multiplier

        mpy = b[15:0];  // Part-select on b to load low-order 16 bits into mpy
        mcnd = a[15:0]; // Part-select on a to load low-order 16 bits into mcnd
        mulitply = 0;
```

Functions and Tasks

```
        repeat (16)
        begin
                if (mpy[0]) // If the low-order bit of mpy is one, concatenate.
                        multiply = mulitply + {mcnd, 16`h0000};
                multiply = multiply >> 1; // multiply shifted right one position
                mpy = mpy >> 1;  // mpy shifted right one position
        end
end
endfunction
endmodule
```

Functions and Tasks

- The Mark-1 processor with a separate module

```
module mark1Mod;
reg [31:0]      m [0:8191];     // 8192 x 32 bit memory
reg [12:0]      pc;             // 13-bit program counter
reg [31:0]      acc;            // 32-bit accumulator
reg [15:0]      ir;             // 16-bit instruction register

reg [31:0]      mcnd;
reg             go;
wire [31:0]     prod;
wire            done;
```

Functions and Tasks

```
multiply        mul (prod, acc, mcnd, go, done);  // Module instantiation


always
begin : executeInstructions
        go = 0;
        ir = m[pc]; // Fetch an instruction.
        case (ir[15:13]) // Decoding and executing
                3'b000 : pc = m[ir[12:0]];
                3'b001 : pc = pc + m[ir[12:0]];
                3'b010 : acc = -m[ir[12:0]];
                3'b011 : m[ir[12:0]] = acc;
```

Functions and Tasks

```
                3'b100,
                3'b101 : acc = acc - m[ir[12:0]];
                3'b110 : if (acc < 0) pc = pc + 1;
                3'b111 : // A handshaking protocol using go and done
                begin
                        wait (~done) mcnd = m[ir[12:0]]; go = 1;
                        wait (done); acc = prod;
                end
        endcase
        #1 pc = pc + 1; // Increment program counter and time.
end
endmodule
```

Functions and Tasks

```
module multiply (prod, mpy, mcnd, go, done);
output [31:0]    prod;
input [31:0]     mpy, mcnd;
input            go;
output           done;

reg [31:0]       prod;
reg              done;
reg [15:0]       myMpy;

always
begin
```

Functions and Tasks

```
        done = 0; wait (go);
        myMpy = mpy[15:0];
        prod = 0;
        repeat (16)
        begin
                if (myMpy[0]) prod = prod + {mcnd, 16`h0000};
                prod = prod >> 1;
                myMpy = myMpy >> 1;
        end
        done = 1; wait (~go);
end
endmodule
```

Rules of Scope

- Verilog allows for identifiers to be defined within four entities:
  - Modules
  - Tasks
  - Functions
  - Named blocks

- Range of description (local scope) over which the identifier is known:
  - Module-endmodule pairs
  - Task-endtask pairs
  - Function-endfunction pairs
  - Begin:name-end pairs

Rules of Scope

- Forward referencing
  - Forward referenced:
    - Identifiers for modules, tasks, functions, and named begin-end blocks are allowed to be forward referencing and thus may be used before they have been defined.
  - Not forward referenced:
    - Forward referencing is not allowed with register and net accesses.
    - They must be defined before they are used.
    - Typically, they are defined at the start of the local scope.
    - An exception is that output nets of gate primitives can be declared implicitly.