Hardware Description Languages: Verilog

Jan Reineke

UNIVERSITÄT
DES
SAARLANDES

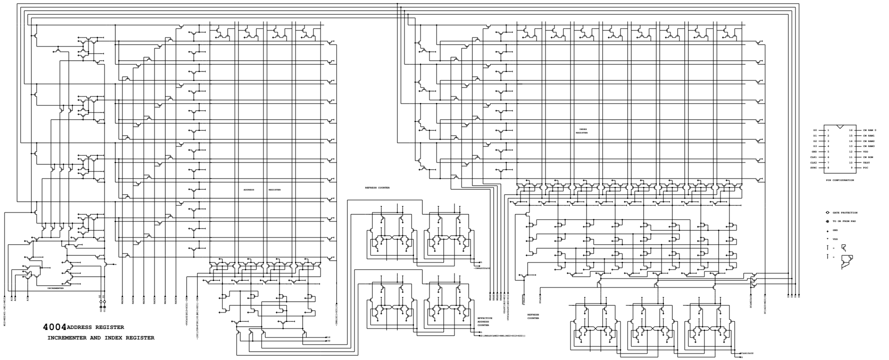
History: Hardware design in 1970

Step I: circuit design

- manual with paper and pencil
- at the level of individual transistors

History: Hardware design in 1970

Example: Intel 4004, 2.300 transistors, 108KHz, 10μm (excerpt)
first mass-produced single-chip microprocessor!



`http://www.4004.com/assets/redrawn-4004-schematics-2006-11-12.pdf`
Re-drawn schematics based on Revision G of the original Intel 4004 schematics by Intel Corporation (August 6, 1976). Schematic capture and design verification by Fred Huettig, Brian Silverman and Barry Silverman (February 2, 2006).

History: Hardware design in 1970

Step I: circuit design

- manual with paper and pencil
- at the level of individual transistors

Step II: Fabrication of photomasks for lithographic process

- manual labor
- error prone

History: Hardware design in 1970

Photo courtesy of the Intel Corporation, as published on
`http://www.computerhistory.org/revolution/digital-logic/12/287/1614`

Hardware design: Today

Hardware description languages

- Abstract from transistors
- Simulation
- Hardware synthesis
  using computer-aided design (CAD) programs
  - Field Programmable Gate Array (FPGA)
  - Application-Specific Integrated Circuit (ASIC)

Hardware design: Today

Hardware description languages

- Abstract from transistors
- Simulation
- Hardware synthesis
  using computer-aided design (CAD) programs
  - Field Programmable Gate Array (FPGA)
  - Application-Specific Integrated Circuit (ASIC)

In this course, we use . . .

Verilog, originally developed by Gateway Design Automation in 1984

Other languages: VHDL, SystemVerilog, Chisel

Previous concepts in the course

Previous concepts in the course

Boolean functions $\mathbb{B}^n \to \mathbb{B}^m$
Boolean expressions $a \wedge b$
Combinatorial circuits

Previous concepts in the course

Boolean functions $\mathbb{B}^n \rightarrow \mathbb{B}^m$

Boolean expressions $a \wedge b$

Combinatorial circuits

Moore/Mealy automata

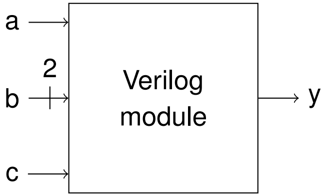
Memory elements (Flip-Flops, SRAM, . . .)

Sequential circuits

Outlook

Modules

Encapsulation of circuits



```
module name(
  input a,
  input [1:0] b,
  input c,
  output y
);

// functionality

endmodule
```

Assignments and Bitwise operators

- Assignment of signals via `assign` and `=`
- Bitwise operators correspond to gates
  And `&`, Or `|`, Negation `~`, Xor `^`

Example: Half adder

Assignments and Bitwise operators

- Assignment of signals via `assign` and `=`
- Bitwise operators correspond to gates
  And `&`, Or `|`, Negation `~`, Xor `^`

Example: Half adder

```
module halfadder(
  input a,
  input b,
  output s,
  output c
);


endmodule
```

Assignments and Bitwise operators

- Assignment of signals via `assign` and `=`
- Bitwise operators correspond to gates
  And `&`, Or `|`, Negation `~`, Xor `^`

Example: Half adder

```
module halfadder(
  input a,
  input b,
  output s,
  output c
);
  assign s = a ^ b;
  assign c = a & b;
endmodule
```

Constants

- General form $N'Bw$
  - N number of bits
  - B basis of the numeral system
  - w sequence of numerals (digits)

Constants

- General form $N'Bw$
  - N number of bits (*independently* (!) of the employed numeral system)
  - B basis of the numeral system
  - w sequence of numerals (digits)

Constants

- General form $N'Bw$
  - N number of bits (*independently* (!) of the employed numeral system)
  - B basis of the numeral system
  - w sequence of numerals (digits)

- Examples:
  - binary numbers `4'b1011`
  - long binary numbers `8'b1010_0011`
  - *without* number of bits `'b101`
  - decimal `4'd5`
  - hexadecimal `8'ha3`
  - signed decimal number `-3'sd5` (s for “signed”)

Constants

- General form $N'Bw$
  - N number of bits (*independently* (!) of the employed numeral system)
  - B basis of the numeral system
  - w sequence of numerals (digits)

- Examples:
  - binary numbers `4'b1011`
  - long binary numbers `8'b1010_0011`
  - *without* number of bits `'b101`
  - decimal `4'd5`
  - hexadecimal `8'ha3`
  - signed decimal number `-3'sd5` (s for “signed”)

- Don’t-care values `3'b1xx` (e.g. irrelevant signals in the MIPS decoder)
  - Simulation: simplifies debugging
  - Synthesis: more flexibility

Comparison and selection operators

- bit-by-bit (in)equality $==$, $!=$
- unsigned comparison of values $<$, $<=$, ...
- ternary operator `a ? b : c`

Example: 4-bit multiplexer

Comparison and selection operators

- bit-by-bit (in)equality ==, !=
- unsigned comparison of values <, <=, ...
- ternary operator `a ? b : c`

Example: 4-bit multiplexer

```
module multiplexer(
  input s,
  input [3:0] x0,
  input [3:0] x1,
  output [3:0] y
);

endmodule
```

Comparison and selection operators

- bit-by-bit (in)equality ==, !=
- unsigned comparison of values <, <=, ...
- ternary operator `a ? b : c`

Example: 4-bit multiplexer

```
module multiplexer(
  input s,
  input [3:0] x0,
  input [3:0] x1,
  output [3:0] y
);
  assign y = (s == 1'b0) ? x0 : x1;
endmodule
```

Hierarchical circuits

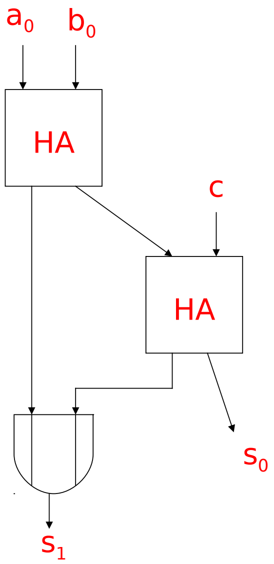
UNIVERSITÄT DES SAARLANDES

Instantiating modules

```
modulename instancename(
   .param1name(arg1name),
   .param2name(arg2name),
   ...
);
```

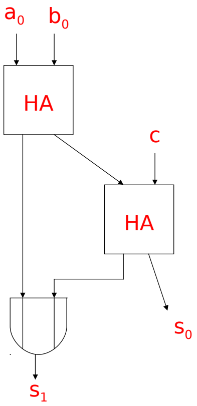
Temporary signals via `wire`

Example: Full adder



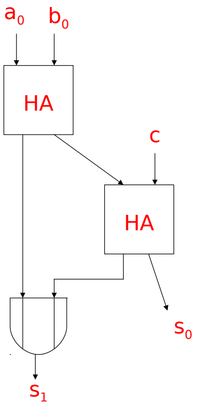
Example: Full adder

```
module fulladder(
  input a0,
  input b0,
  input c,
  output s0,
  output s1
);



endmodule
```

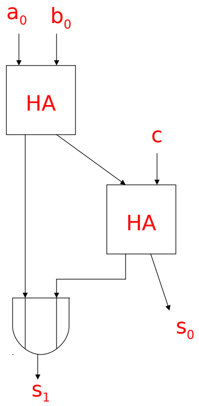


Example: Full adder

```
module fulladder(
  input a0,
  input b0,
  input c,
  output s0,
  output s1
);
  wire ha0c, ha0s, ha1c;




endmodule
```

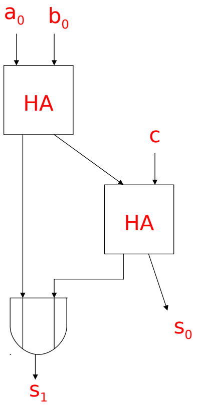


Example: Full adder



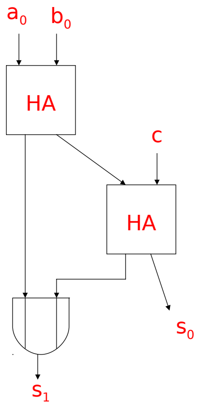
```
module fulladder(
  input a0,
  input b0,
  input c,
  output s0,
  output s1
);
  wire ha0c, ha0s, ha1c;
  halfadder ha0(.a(a0), .b(b0), .c(ha0c), .s(ha0s));


endmodule
```

Example: Full adder



```
module fulladder(
  input a0,
  input b0,
  input c,
  output s0,
  output s1
);
  wire ha0c, ha0s, ha1c;
  halfadder ha0(.a(a0), .b(b0), .c(ha0c), .s(ha0s));
  halfadder ha1(.a(ha0s), .b(c), .c(ha1c), .s(s0));

endmodule
```

Example: Full adder



```
module fulladder(
  input a0,
  input b0,
  input c,
  output s0,
  output s1
);
  wire ha0c, ha0s, ha1c;
  halfadder ha0(.a(a0), .b(b0), .c(ha0c), .s(ha0s));
  halfadder ha1(.a(ha0s), .b(c), .c(ha1c), .s(s0));
  assign s1 = ha0c | ha1c;
endmodule
```

More operators

- Part-selects of `a[7:0]`
  `a[2]` or `a[1:0]` or `a[3 +: 4]` (= `a[6:3]`)

More operators

- Part-selects of `a[7:0]`
  `a[2]` or `a[1:0]` or `a[3 +: 4]` (= `a[6:3]`)
- If e.g. `a[7:0] = 10011011`
  Then `a[2] = 0`, `a[1:0] = 11`, and `a[6:3] = 0011`.

More operators

- Shift operation, logical (fills with zero) `<<`, `>>`
  E.g. `1010 >> 1` = `0101`

More operators

- Shift operation, logical (fills with zero) `<<`, `>>`
  E.g. `1010 >> 1` = `0101`
- Shift operation, arithmetic (keep sign) `<<<`, `>>>`
  E.g. `1010 >>> 1` = `1101`

More operators

- Shift operation, logical (fills with zero) <<, >>
  E.g. `1010 >> 1` = `0101`
- Shift operation, arithmetic (keep sign) <<<, >>>
  E.g. `1010 >>> 1` = `1101`
- In practice, <<< and << behave the same.

More operators

- Reduction `&a` of `a[7:0]`
  is equivalent to `a[7] & a[6] & ... & a[0]`

More operators

- Reduction `&a` of `a[7:0]`
  is equivalent to `a[7] & a[6] & ... & a[0]`
- arithmetic operators (unsigned) `+`, `*`, ...

More operators

- Reduction `&a` of `a[7:0]`
  is equivalent to `a[7] & a[6] & ... & a[0]`
- arithmetic operators (unsigned) `+`, `*`, ...
- Concatenation of `a` and `b` into `c[1:0]`
  `assign c[1:0] = {a, b};`

More operators

- Reduction `&a` of `a[7:0]`
  is equivalent to `a[7] & a[6] & ... & a[0]`
- arithmetic operators (unsigned) `+`, `*`, ...
- Concatenation of `a` and `b` into `c[1:0]`
  `assign c[1:0] = {a, b};`
- Duplication of bits `c[31:0] = {16{2'b10}}`
  = `c[31:0] = 10101010101010101010101010101010`

Outlook

Memory elements

Two types of memories:

- level-triggered latches
- edge-triggered flip-flops

Memory elements

Two types of memories:

- level-triggered latches
- edge-triggered flip-flops

Both types of memory are *in principle* supported by Verilog, but not by all synthesis tools

$\Rightarrow$ We are only using edge-triggered flip-flops

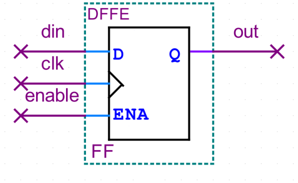
Verilog syntax for flip-flops

- Declaration of variable that can be used as a 1-bit memory:

  ```
  reg q;
  ```
- `always` block for assignment of `reg` variables.
  Execution depends on variables in *sensitivity list*.

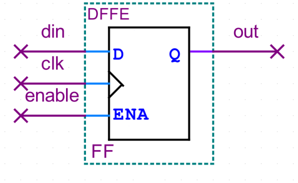
  ```
  always @(sensitivity list)
  begin
    ...
  end
  ```
- Edge detection `posedge a` and `negedge a`
- (Non-blocking) assignment `<=`
- Common structuring elements within an `always` block: `if`, `case`

Example I: D-Flip-Flop with enable input



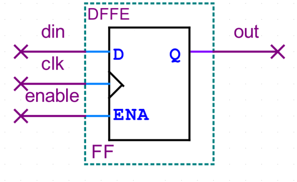
```
module dffe(
  input clk,
  input din,
  input enable,
  output out
);









endmodule
```

Example I: D-Flip-Flop with enable input



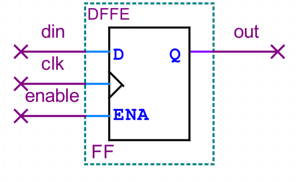
```
module dffe(
  input clk,
  input din,
  input enable,
  output out
);
  reg q;




endmodule
```

Example I: D-Flip-Flop with enable input



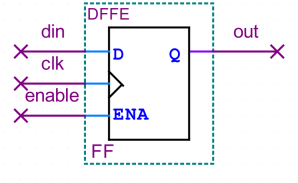
```
module dffe(
  input clk,
  input din,
  input enable,
  output out
);
  reg q;
  always @(posedge clk)
  begin


  end

endmodule
```

Example I: D-Flip-Flop with enable input



```
module dffe(
  input clk,
  input din,
  input enable,
  output out
);
  reg q;
  always @(posedge clk)
  begin
    if (enable)
      q <= din;
  end

endmodule
```

Example I: D-Flip-Flop with enable input



```
module dffe(
  input clk,
  input din,
  input enable,
  output out
);
  reg q;
  always @(posedge clk)
  begin
    if (enable)
      q <= din;
  end
  assign out = q;
endmodule
```

Example II: Cyclic 4-bit counter

```
module counter (
  input clock,
  input reset,
  output [3:0] count
);
  reg [3:0] q;
  always @(posedge clock)
  begin
    if (reset)
      q <= 4'b0;
    else
      q <= q + 1'b1;
  end
  assign count = q;
endmodule
```

Example II: Cyclic *n*-bit counter

```
module counter #(parameter N = 4) (
  input clock,
  input reset,
  output [N-1:0] count
);
  reg [N-1:0] q;
  always @(posedge clock)
  begin
    if (reset)
      q <= 'b0;
    else
      q <= q + 1'b1;
  end
  assign count = q;
endmodule
```

Outlook: `always` for combinatorial circuits

```
wire x, y;
assign x = a ? b : c;
assign y = a ? b : d ? c : e;
```

Outlook: `always` for combinatorial circuits

```
wire x, y;
assign x = a ? b : c;
assign y = a ? b : d ? c : e;
```

```
reg x, y;
always @*
begin
  if (a) begin
    x = b;
    y = b;
  end else begin
    x = c;
    if (d)
      y = c;
    else
      y = e;
  end
end
```

⇒ `reg` and `always` can also be used for combinatorial circuits

Blocking and non-blocking assignments

Blocking assignment

```
reg x, y;
always @(posedge clk)
begin
  x = y;
  y = x;
end
```

Assignments are blocking,
i.e., they are executed **sequentially**
$\Rightarrow$ *x* and *y* are equal

Non-blocking assignment

```
reg x, y;
always @(posedge clk)
begin
  x <= y;
  y <= x;
end
```

Assignments are non-blocking, i.e.,
are conceptually executed in parallel
$\Rightarrow$ *x* and *y* are exchanged

Rules for clean code

1. For memory elements use `always @(pos/negedge clk)`, non-blocking assignments `<=` and `reg`
2. For simple combinatorial circuits `assign`, blocking assignments = and `wire`
3. For complex combinatorial circuits `always @*`, blocking assignments = and `reg`
4. Never assign a signal multiple times (`wire` or `reg`), i.e., no two *active* assignments at the same time

Rules for clean code

1. For memory elements use `always @(pos/negedge clk)`,
   non-blocking assignments `<=` and `reg`
2. For simple combinatorial circuits `assign`,
   blocking assignments `=` and `wire`
3. For complex combinatorial circuits `always @*`,
   blocking assignments `=` and `reg`
4. Never assign a signal multiple times (`wire` or `reg`),
   i.e., no two *active* assignments at the same time

Follow these rules

$\Rightarrow$ Otherwise: *strange* often, subtle errors

Outlook

Test bench

A test bench is a Verilog module that

1. instantiates a module-under-test *M*,
2. generates input stimuli for *M*, and
3. *optionally* tests the correctness of the outputs of *M* automatically.

Test bench: Input stimuli

Use of *delays*:

`statement 1; #5; statement 2;` delays the sequential execution

Test bench: Input stimuli

Use of *delays*:

`statement 1; #5; statement 2;` delays the sequential execution

- Once

```
initial
 begin
  reset <= 1;
  #22;
  reset <= 0;
 end
```

Test bench: Input stimuli

Use of *delays*:

`statement 1; #5; statement 2;` delays the sequential execution

- Once

```
initial
 begin
  reset <= 1;
  #22;
  reset <= 0;
 end
```

- Periodically

```
always
 begin
  clk <= 1; #5; clk <= 0; #5;
 end
```

Test bench: Input stimuli

Use of *delays*:

`statement 1; #5; statement 2;` delays the sequential execution

- Once

```
initial
 begin
  reset <= 1;
  #22;
  reset <= 0;
 end
```

- Multiple times with `repeat(x)`

- Periodically

```
always
 begin
  clk <= 1; #5; clk <= 0; #5;
 end
```

Additional functions

Additional functions to control the simulation, non-synthesizable, thus mainly used in test benches

- `$display` output similar to `printf` in C
- `$finish` stops the simulation
- `$readmemb`/`$readmemh` initializes memory elements
- `$dumpfile`/`$dumpvars` outputs variables as waveforms

Tools

Simulation

Icarus: open-source simulator for Verilog
`http://iverilog.icarus.com/`

GTKWave: open-source visualization and analysis tool for waveforms
`http://gtkwave.sourceforge.net/`

Tools
Simulation

Icarus: open-source simulator for Verilog
`http://iverilog.icarus.com/`

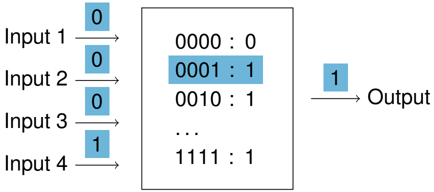
GTKWave: open-source visualization and analysis tool for waveforms
`http://gtkwave.sourceforge.net/`

Demo

Outlook

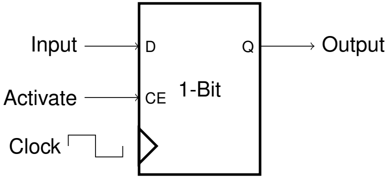
Structure of an FPGA — Logic blocks

- Look-up table (4LUT)

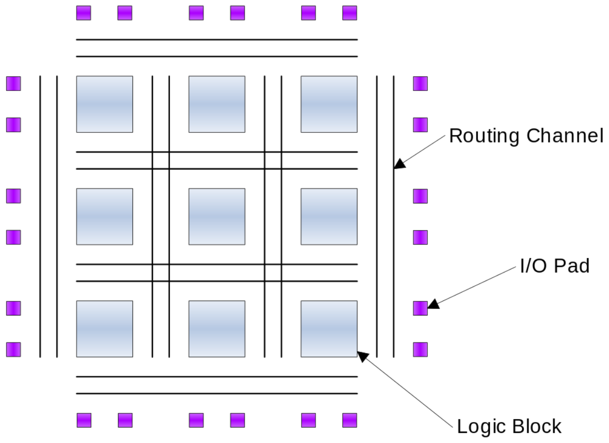


Can be used to implement an arbitrary Boolean function from $\mathbb{B}^4 \rightarrow \mathbb{B}$.

- Register



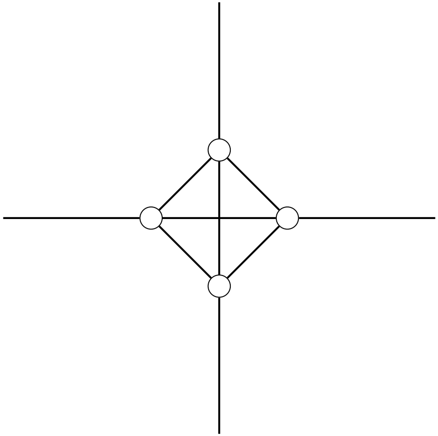
Structure of a Field Programmable Gate Array[1]



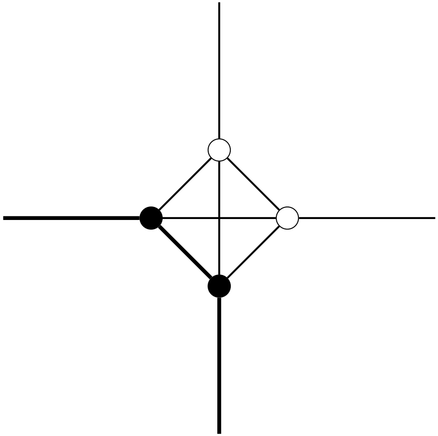
In addition: Dedicated components:
Memory blocks (8192 Bits), multipliziers (9 Bits), etc.

[1] `http://commons.wikimedia.org/wiki/File:Fpga_structure.svg`

Structure of an FPGA — Network



Structure of an FPGA — Network



Compilation

1. Analysis and elaboration
   - Syntax and semantics
   - Find dedicated components
2. Synthesis
   - Translation into logical blocks
3. Fitter
   - Place-and-Route
4. Assembler
5. Timing analysis
   - Computation of critical paths and maximal frequency

Example: Pong

https://commons.wikimedia.org/wiki/File:APF_TV_Fun_(with_paddle_model).jpg
Original from: http://www.flickr.com/photos/adampsyche/3085132136/

Brainstorming: Pong game logic

State?

Behavior?

Brainstorming: Pong game logic

State?

- Position of left paddle
- Position of right paddle
- Position/speed of ball

Behavior?

Brainstorming: Pong game logic

State?

- Position of left paddle
- Position of right paddle
- Position/speed of ball

Behavior?

- Collision paddles
- Collision borders

Brainstorming: Pong game logic

State?

- Position of left paddle
- Position of right paddle
- Position/speed of ball

Behavior?

- Collision paddles
- Collision borders

Verilog implementation:
`https://github.com/tdudziak/fpga-pong`

Some applications of FPGAs

- Retro games `http://bit.ly/1r9EMks`
- Signal processing
- Hardware Prototyping
- Encryption of network traffic `http://bit.ly/WkzpcE`
- Parallel algorithms `http://bit.ly/ZYOTJG`,
  DES attack (2003) `http://bit.ly/11ZKxze`
- Earlier: Bitcoin mining (now: ASICs)