bitvis

1100010110100111101001110110110100111110011

The Good, The Bad and the Ugly

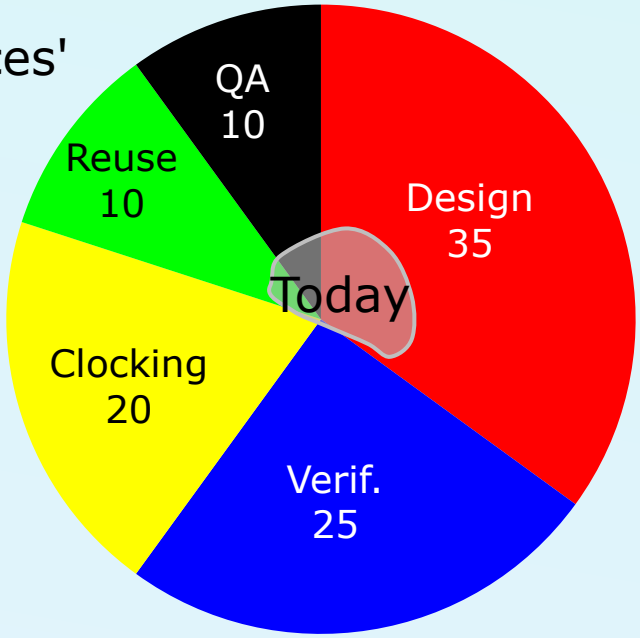
Bitvis

- Independent Design Centre for
  -Embedded SW
  -FPGA (& ASIC)
  The leading independent design centre in Norway
- 13 designers – (6 in March 2012...)
- Specification, Design, Verification, Implementation, Test

- Methodology partner
- Sparring and review partner
- Verification IP provider
  - Bitvis Utility Library (free and open source) ++
- Course provider "FPGA Development Best Practices"

FPGA Development Best Practices

- Two-day course: 'FPGA Development Best Practices'
  - Based on the Digitas course presented in DK, SE, NO
  - Focus on improving your FPGA-based projects
    - Design Structure and Methodology
    - Verification Structure and Methodology
    - Focus on Efficiency and Quality improvement
    - Focus on a Best Practices approach that you can apply today
  - Practical and Pragmatic approaches
  - Very good feedback from participants in all countries
  - Quotes from last three courses:
    - 'The only bad thing about this course - is that we didn't do it earlier'
    - 'An eye opener' & 'Most issues apply directly in our organisation'
    - 'A very good course on very relevant improvement potentials'



Future course:
Design & Clocking will be increased. Most verification out as a separate day.

Abstract

- The way you implement your FPGA design and write your code has a huge impact on your development efficiency and product quality. The strange thing is that even many experienced designers tend to write both bad and ugly code.
  Does it matter if the code is ugly if it works in the lab?
  Yes – definitely, and for several reasons. First of all – the probability that ugly code has serious bugs is far higher than for good code. Also any change made to ugly code has a far higher risk of introducing bugs. And of course – ugly code makes it far more difficult to do a proper review. More time consuming, often frustrating, and with a far worse review quality.
  Bad and ugly code often results in errors that may be difficult to find and terrible to correct.
  This presentation will show some examples of bad and ugly code, how they result in inefficiency or bugs, and also suggest some remedies and suggestions for improvements – in order to write good code. (Examples will be in VHDL, but apply equally well for other languages)

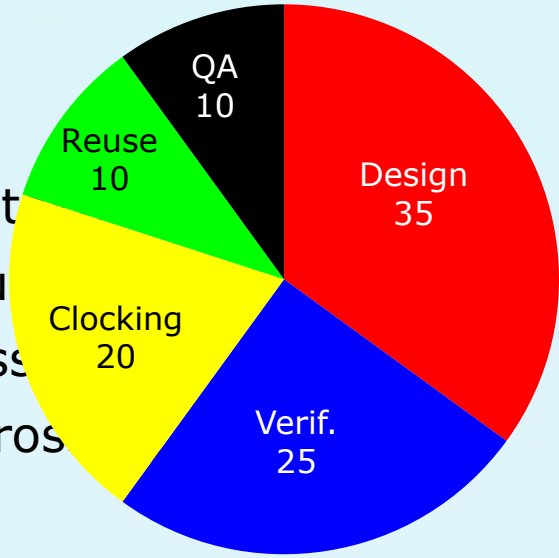
Main problem areas

Bad & Ugly code:

- Micro architecture
- HDL coding style
- Naming

Bad & Ugly design

- Architecture
- HDL coding mist
- HDL language u
- Digital design iss
- Clock domain cros
- Timing closure
- etc.....



➔ Seriously affects:

Quality, Schedule and Cost,
Frequency, Power and Area,
Readability, Modifiability and Risk

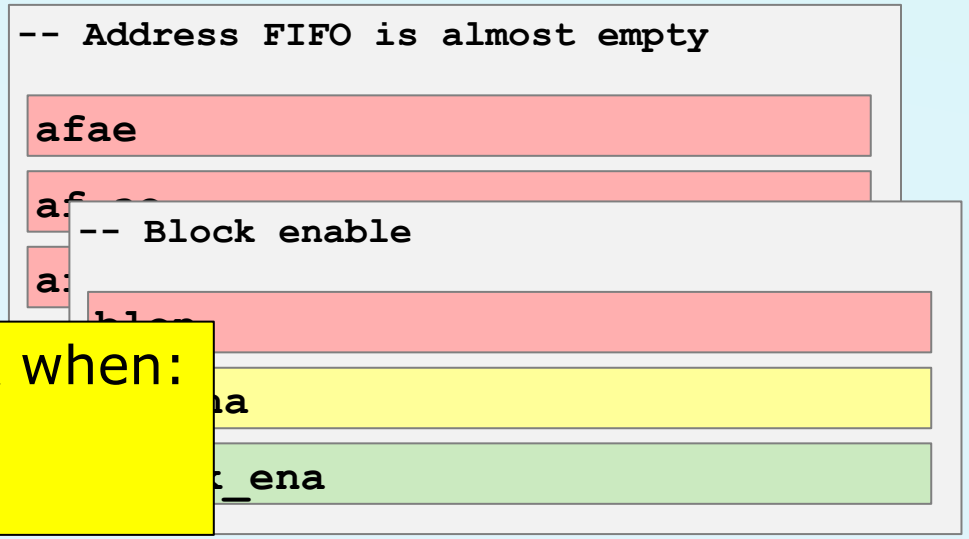
Sources of examples

- Mainly from 16 years as a consultant
  - Reviews
  - Problem debugging & fixing
  - Simulation debugging and related bug fixing
  - Design modification and extension

- Cases and examples have been slightly modified to hide company, application and designer
  - Unfortunately excludes some excellent examples...

Bad names - Abbreviations

- None standard abbreviations
- Extremely common



Abbreviations only ok when:
- clearly defined or
- obvious to anyone

Bad names - Variants of a signal

- Signals that have been slightly modified
- Extremely common problem

```
frame_bit_counter <= ....... (actual counter)

frm_bit_counter   <= a snapshot when address
                     field completed

frame_bit_cnt     <= at end of previous frame

                  p  <= value held for bit period
                     - in case of jitter

                  t  <= expected number of bits
```

```
rx(1:0)  -- Dual input representing single bit

);--shift reg

rx0int
```

Extreme case,
- but lots of cases with 2-3 variants.
  They get mixed up all the time....

```
if frame_bit_cnt = num_bits_frame then
```

Bad names - N dimensions

- Signal array with N dimensions
  e.g.
  - line number (A/B)
  - channel number (1-6)
  - bit number (N)
  - delay number (0-3)
- Extremely confusing
- Use Conventions for delay
- Use Enumerated
  - And arrays of enumerated

```
data(1,3,2)   -- Line A, Ch 3, Bit 2
data_a(3,2)   -- Line A, Ch 3, Bit 2
data_d2(A,Ch3) -- Line A, Ch 3, Delay 2
```

Typically lots of signal and variable variants
- Special naming - hopefully structured
- Names refer to different dimensions
They get mixed up all the time....

Bad names - Logical mismatch

- Name clearly indicates a function, but does something slightly different.
  - 'read' when trigger read is intended.
    E.g. one FSM triggering another
  - 'crc' when 'crc_error' is intended
  - '*_mask' when enable is intended (e.g. for interrupts)

- Functionality changed, but name is kept
  - 'clk32' when frequency actually changed to 16 MHz

Bad names - Unknown number unit

- Name clearly indicates a function or number, but not specific enough
  E.g. number of bits in a frame
  - frame_**size** Bits, bytes, words, ...?
    Often varying unit in same design.

```
-- Frame size in number of bits
   frame_size
   frame_size_bits
   num_bits_frame
```

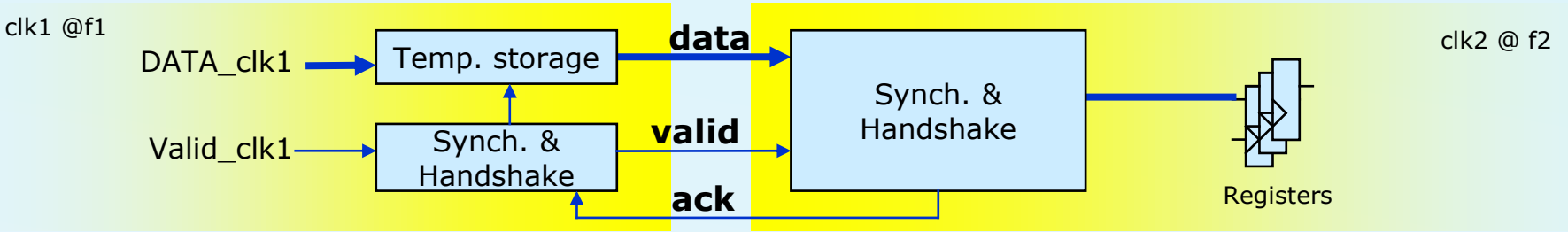
Bad names - Toggle-signals

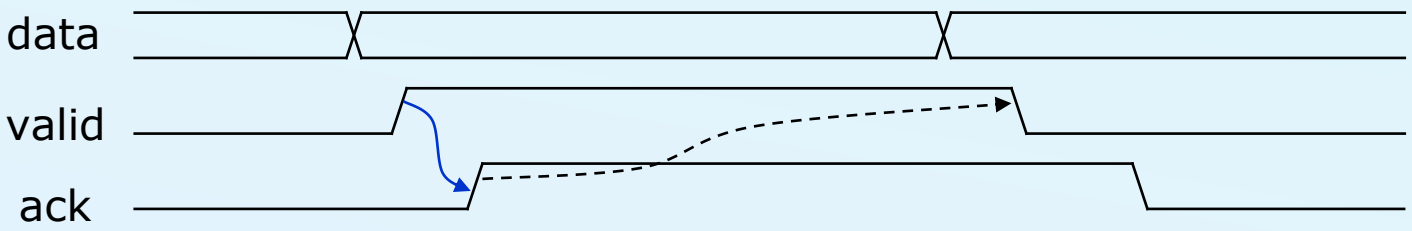
- Signal is "valid" on toggle
- Must find alternative name

t_valid

toggle_valid

valid_on_toggle





Bad names - Number suffix

- Differentiate between variants of number suffixes

```
variables:
  status
  status1 or status_1
  status2 or status_2
```

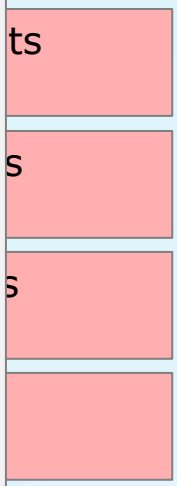
Typically used for lots of ad-hoc "conventions"

a) 3 different types of status?

b) status + pipeline stage 1 and 2

c) status + synchronized once and twice

d) status and slightly modified versions
(e.g. masked, enabled, snapshot, etc....



Numbers - from 0 or 1

- Often confusing whether number N is the N^{th} or $(N+1)^{th}$ occurrence.
  E.g. whether 13 is the 13^{th} or 14^{th} occurrence.

E.g.

- bit_
- bit_
- bit_
- bit_pointer

E.g.

w for your code?

- Conventions (e.g. bit_0idx)?
- Special names (e.g. idx vs cnt)
- Comment on non-obvious

ode?

ything?

Sometimes obvious - Often not

channels?

events?

strings?

node?

Non feasible flexibility

- No point in generics for something that is not really generic
  - E.g. GC_BUS_DATA_WIDTH
    - But the design only works for 8-bit
  - E.g. GC_NUM_CHANNELS
    - But only designed for 6
  - E.g. GC_NUM_BITS_IN_FRAME
    - But design only works for 512

Constants for obvious values


```
constant C_ENABLE  : std_logic := '1';
constant C_DISABLE : std_logic := '0';
.......

if (......) then
  bit_cnt_ena  <= C_DISABLE;
```

Defining new types

- Most designers should define more user types
  - Excellent for consistent design and interfaces
  - e.g. enumerated, record, sub-types, ...
- **Don'**
  - Vector types?
  - re
  - re

```
signal IRR : std_logic_vector (4 downto 0);
```

```
-- in local package:
                                                    5;
  std_logic_vector (C_IRR_WIDTH-1 downto 0);
```

```
Generic constant:  GC_NUM_IRQ : integer := 5;
........
signal IRR : std_logic_vector (GC_NUM_IRQ-1 downto 0);
```

Numeric constants for non-numeric objects


```
-- (0:Cyclone, 1:Spartan, 2: Igloo
constant C_DEVICE : natural := 2;
.....
if C_DEVICE = 2 then....
```

```
type t_device is (cyclone, spartan, igloo);
constant C_DEVICE : t_device := igloo;
.....
if C_DEVICE = igloo then....
```

Repeated complex expressions (1)

- Used in 'if-else' or 'case'
- Quite common
- Simple copy'n'paste
- Often complex to read
- Error prone if modified

➔ Use temporary variables

➔ Explain them

➔ Explain branch differences

```
E.g.

A: (bit_cnt > 16)
B: (my_rec.my_sig(5) = '1')
C: (addr = "110001")
```

```
if A and B or C and P and Q then
  <something>
elsif A and B or C and R then
  <something>
elsif A and B or C and (S or T) then
  <something>
end if;
```

```
  <something>
elsif v_x and (S or T) then
  <something>
end if;
```

```
  end if;
end if;
```

Repeated complex expressions (2)


```
case my_fsm is
  ....
 when STATE_1 =>
    if addr >                                          then
    .....
 when STATE_4
    if addr >                                          then
    .....

   etc...
```

```
v_all
if ad                                                  2 then
  v_a
end i

case
  ...
  whe
    i                                                  ' then
    .
  whe
    i                                                  ' then
    .....

   etc...
```

```
v_x := A;  --Explain and/or good name

if v_x and P then
  <something>
endif
..
..
if v_x or Q then
  <something>
endif
```

➔ Use te

➔ Explai

Simplify complex expressions

Example: Clear an interrupt on writing '1'
(Inside a clocked process)

```
if 'CPU writes to irq-reg' then
  irq := NOT data-in AND irq;
end if;
```

```
if 'CPU w
  if (dat
    irq
  end if;
end if;
```

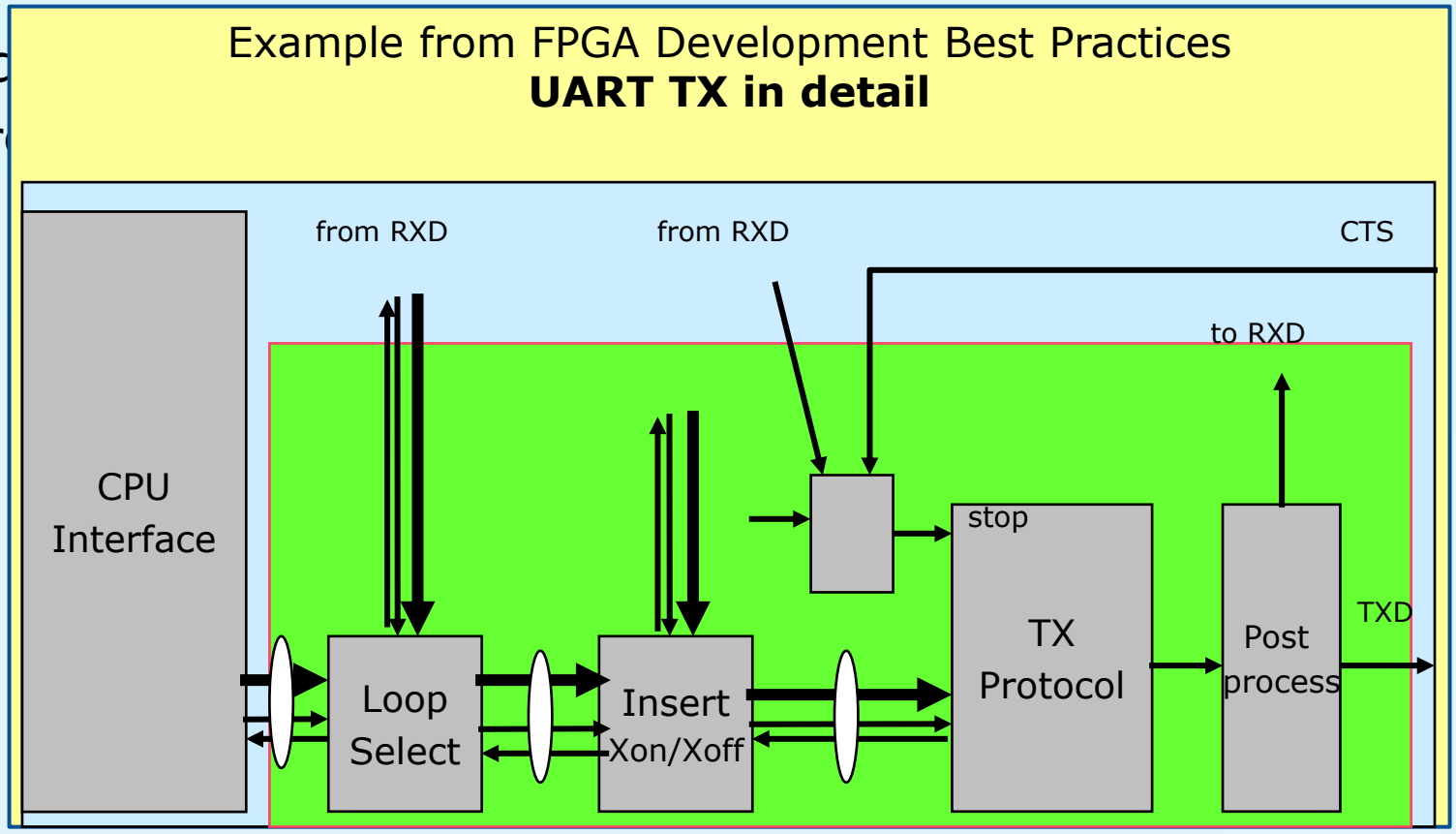
Extremely simple.
Still –
need to stop and think

```
if 'CPU writes                    n = '1') then
  irq := '0';
end if;
```

More readable even when more lines

Main Micro architecture issues

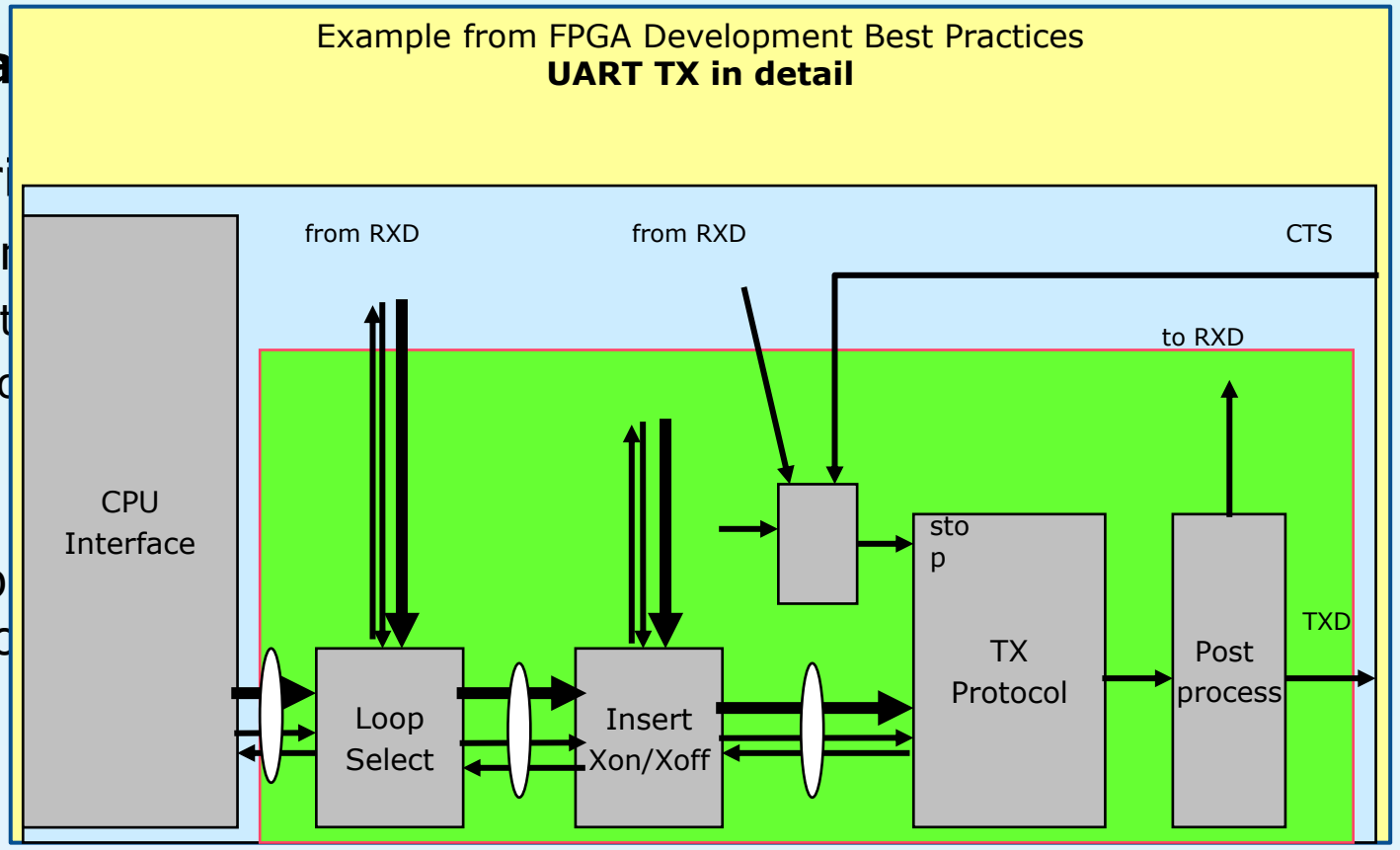
- Block d
  - "Mor
  - "It's
  - "The



Divide and conquer

Strive for **Ma**

- The majori
  - Results i
    - Difficult
    - More c
    - Higher
    - Slower
  - Best imp
    - and pro



Paradox 3 (from 'FPGA BP course')

There seems to be a significant focus on fast code writing.

Yet - code readability and understanding is **far** more important…

The Good, The Bad and the Ugly

Thank you

Quality in every bit

More info on
- Bitvis Utility Library
- FPGA Best Practices

under
www.bitvis.no