TSEA44: Computer hardware – a system on a chip

Lecture 6: Lab 2 intro, Pitfalls when coding, debugging, Design for FPGAs

Material by Andreas Ehliar

LINKÖPING UNIVERSITY

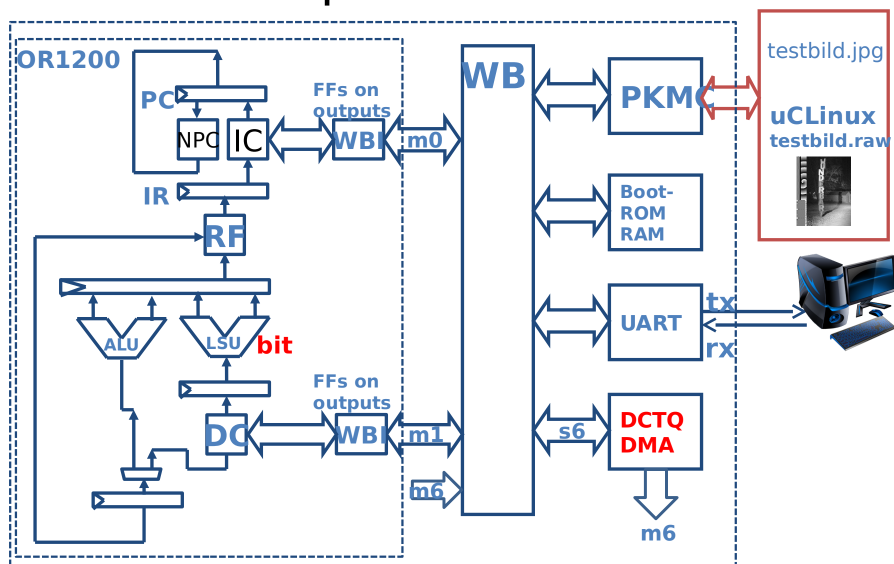
Agenda

- Lab2 introduction (shown already at end of lecture 4)
- Pitfalls when writing code
- Debugging
- Influence of goal hardware on architecture and code style

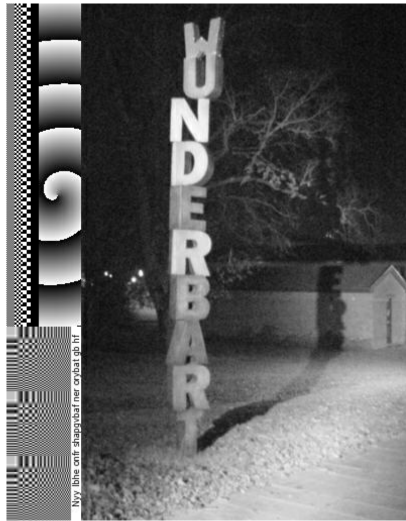
Lab 2 – A JPEG accelerator

1. Design HW
2. Change existing software jpegfiles under uCLinux
   a) insert your accelerator
   b) insert your DMA
   c) insert your instruction

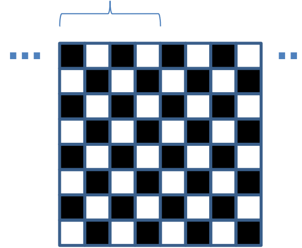
Our FPGA computer with accelerator



Raw image format in memory



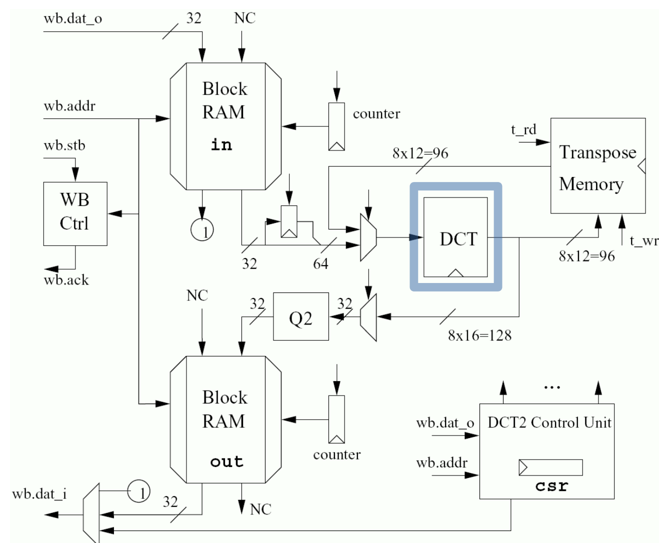
0x00ff00ff



8 bit pixels [0,255]
4 pixels/word

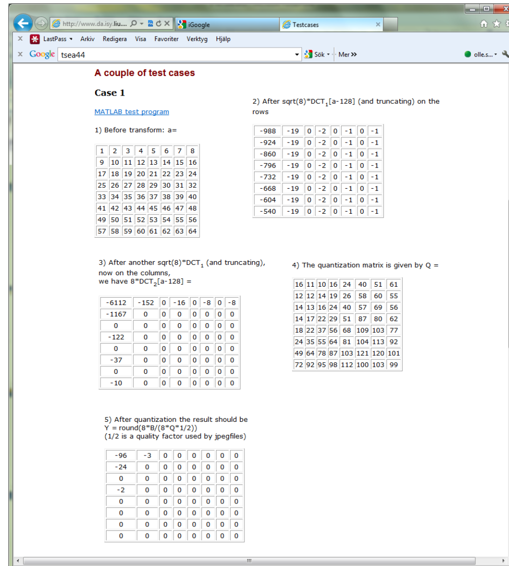
Somewhere 128 must be subtracted from each pixel!

Proposed architecture



Testcases available

- Lab page of the course webpages
- Includes code for quantization



A couple of test cases

Case 1

MATLAB test program

1) Before transform: a=

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

2) After sqrt(8)*DCT_1[a-128] (and truncating) on the rows

-988	-19	0	-2	0	-1	0	-1
-924	-19	0	-2	0	-1	0	-1
-860	-19	0	-2	0	-1	0	-1
-796	-19	0	-2	0	-1	0	-1
-732	-19	0	-2	0	-1	0	-1
-668	-19	0	-2	0	-1	0	-1
-604	-19	0	-2	0	-1	0	-1
-540	-19	0	-2	0	-1	0	-1

3) After another sqrt(8)*DCT_1 (and truncating), now on the columns, we have 8*DCT_2[a-128] =

-6112	-152	0	-16	0	-8	0	-8
-1167	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-122	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-37	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-10	0	0	0	0	0	0	0

4) The quantization matrix is given by Q =

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

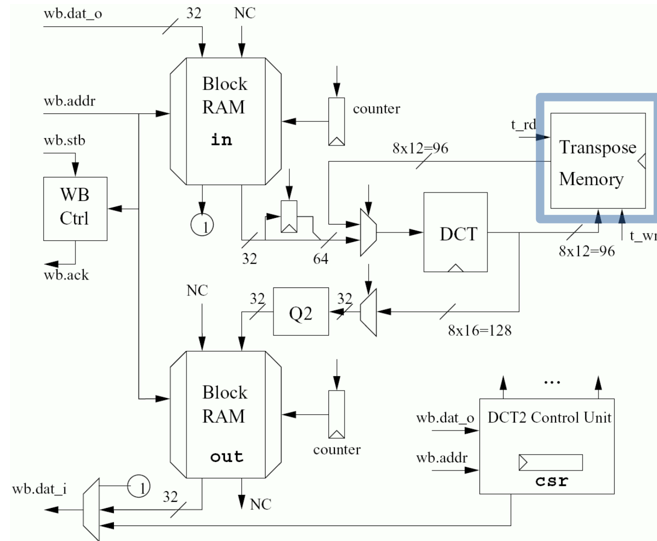
5) After quantization the result should be
Y = round(8*B/(8*Q*1/2))
(1/2 is a quality factor used by jpegfiles)

-96	-3	0	0	0	0	0	0
-24	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-2	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

DCT module

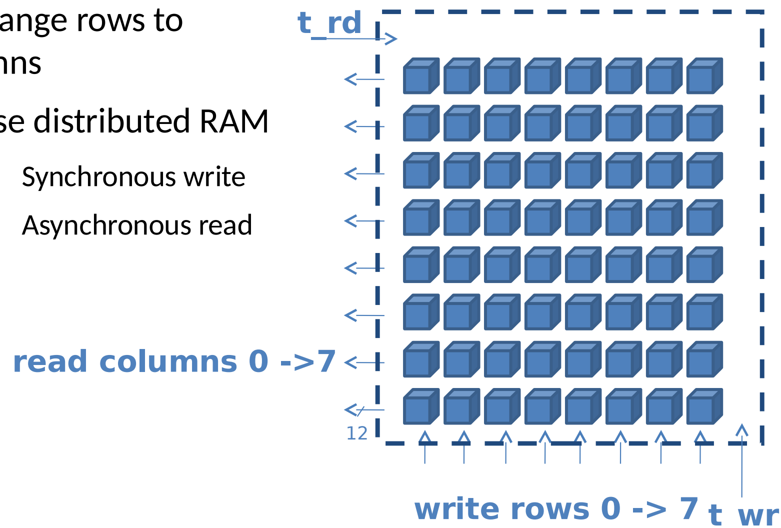
- Given to you
  - 1D DCT
    - 8 in ports (12 bits), 8 out ports (16 bits)
    - Fix point arithmetic
    - Straightforward implementation of Loeffler's algorithm

Proposed architecture

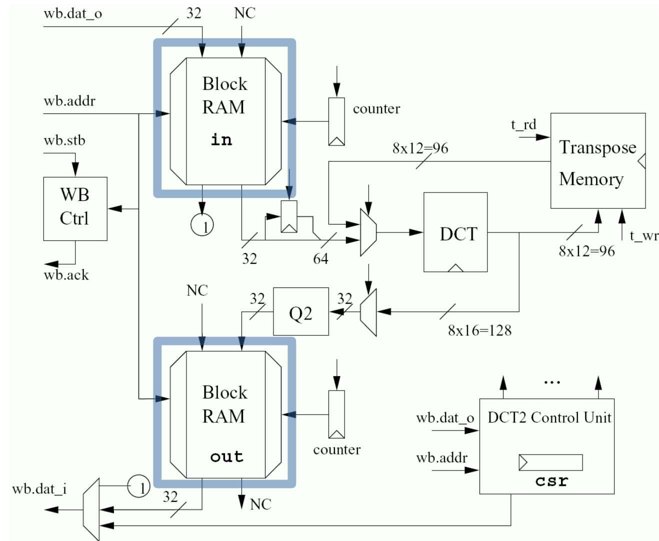


Transpose Memory

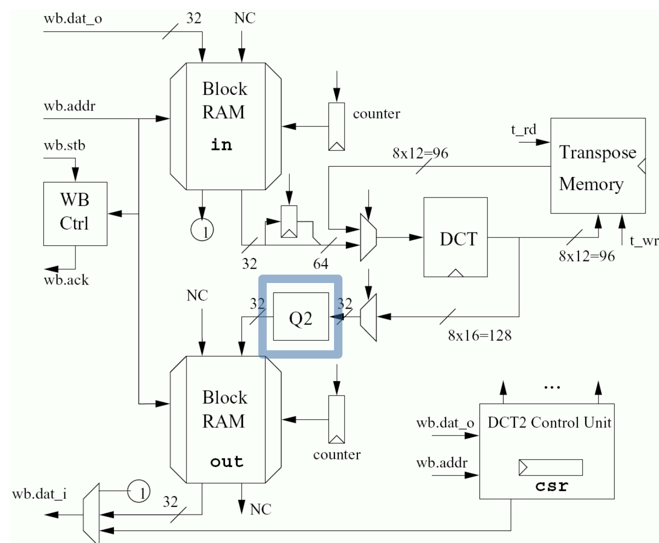
- Rearrange rows to columns
  - Use distributed RAM
    - Synchronous write
    - Asynchronous read



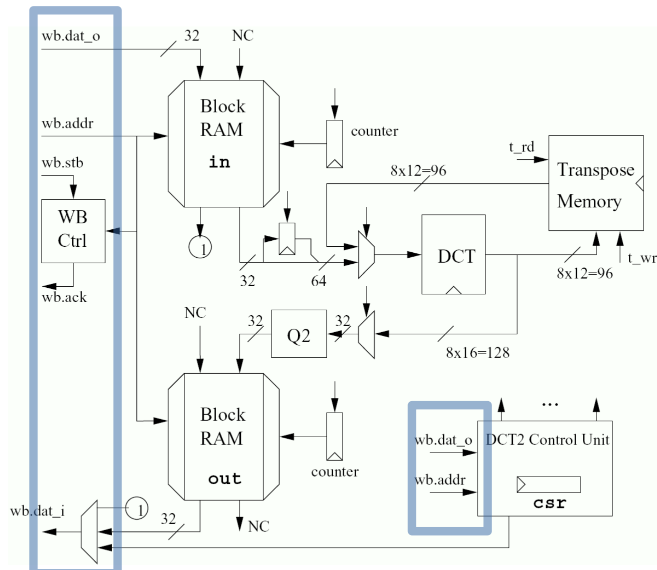
Proposed architecture



Proposed architecture

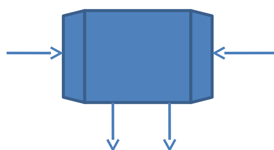


Proposed architecture

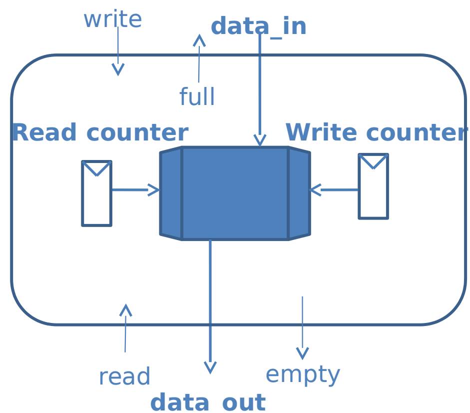


Some ideas

You can read 8 pixels per clock, If you use both ports

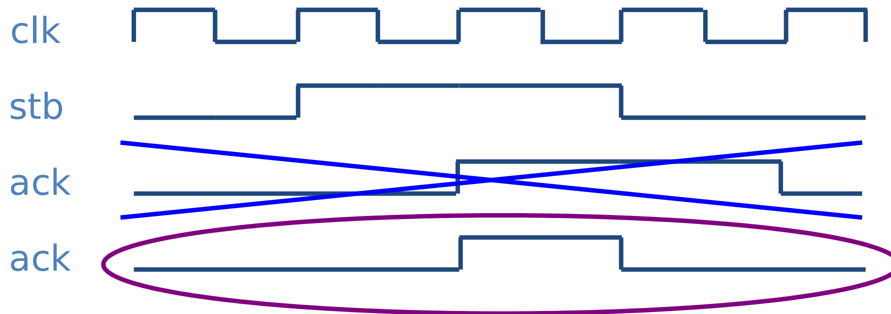


You can rebuild the BRAM to a FIFO



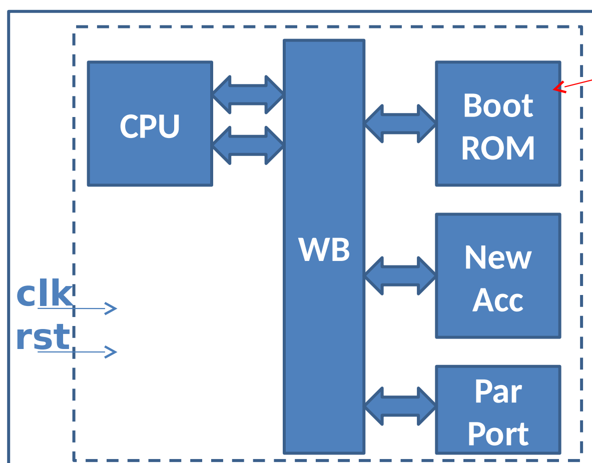
Some notes on the WB I/F

- Be careful with wb.ack



Test benches – 2 alternatives

1) Simulate the whole computer – make sim



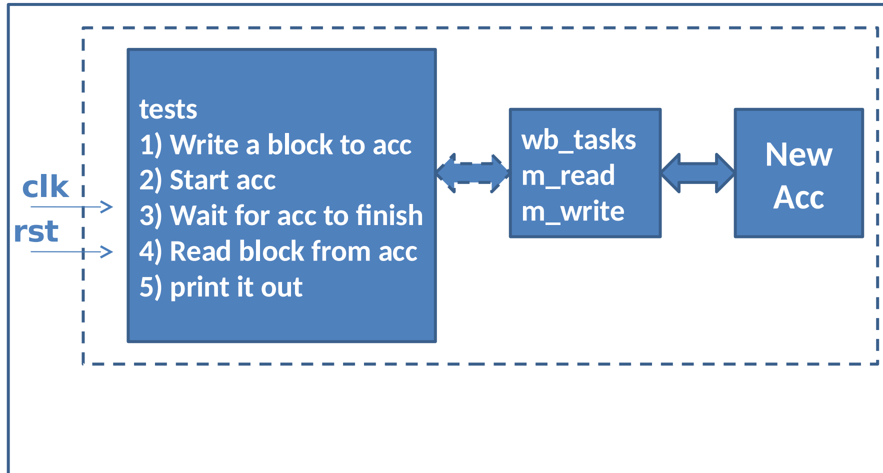
Insert some code
In the beginning of
the monitor `mon2.c`

There are some
alternatives to
uncomment

Tip: You can write to
parport to make it
easier to find things in
ModelSim

Test benches – 2 alternatives

2) Simulate the accelerator – make sim_jpeg



wb_tasks.sv

```
module wishbone_tasks(wishbone.master wb);
   int result = 0;
   reg oldack;
   reg [31:0] olddat;

   always @(posedge wb.clk) begin
      oldack <= wb.ack;
      olddat <= wb.dat_i;
   end

   task m_read(input [31:0] adr, output logic [31:0] data);
      begin
      @(posedge wb.clk);
      wb.adr <= adr;
      wb.stb <= 1'b1;
      wb.we  <= 1'b0;
      wb.cyc <= 1'b1;
      wb.sel <= 4'hf;

      @(posedge wb.clk);
      #1;
      while (!oldack) begin
        @(posedge wb.clk);
              #1;
      end
      wb.stb <= 1'b0;
      wb.we  <= 1'b0;
      wb.cyc <= 1'b0;
      wb.sel <= 4'h0;

      data = olddat;
      end
   endtask // m_read
   ...
endmodule // wishbone_tasks
```

Potential pitfalls when creating a design

- What can go wrong?
  - Design mistakes
  - Synthesis errors
  - Runtime errors
- Crossing clock domains
  - Handshaking
  - Asynchronous FIFOs

A design bug

- Symptom: The boot sequence of uClinux hangs after a second when the Icache is on.
- Uclinux boots ok with Icache off
- No problems detected in the monitor when the icache is on

First try

- Modify the testbench so uClinux is present in SDRAM models
- Add interesting signals to the wave window
- Run the simulation over night

Oops…

- In the morning the simulation was not running any longer
- The log files had filled up all free space on the fileserver…
  - … which promptly crashed, causing all sorts of merriment

Handling long simulation runtimes

- Use checkpointing to reduce/eliminate the need for logging
  - Add no signals to wave window (and log for that matter)
  - Modify UART so printouts are displayed in the transcript window (using $display())
  - run 100 ms; checkpoint 100ms.chk
  - run 100 ms; checkpoint 200ms.chk
  - run 100 ms; checkpoint 300ms.chk
  - …

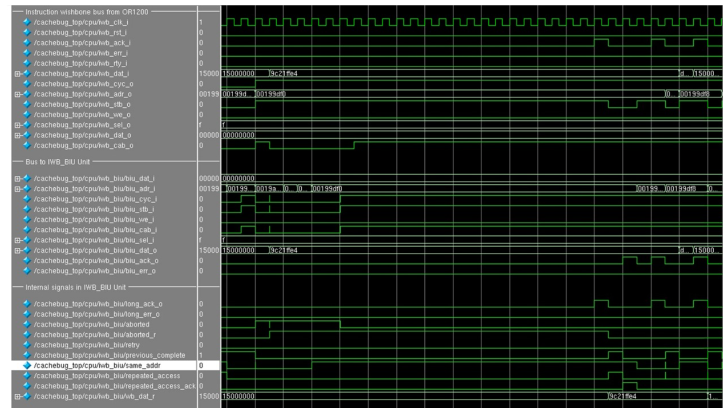
Handling long simulation runtime, cont.

- Now you can pinpoint the time interval where the crash happened
  - Restore the checkpoint in Modelsim that occured closest before the actual crash
  - vsim -restore 600ms.chk
  - Debug as usual (by adding signals to wave window/etc)

2022-11-22 10:36

So what was the bug?

- Cacheline filled up incorrectly (AAAA AAAA CCCC DDDD instead of AAAA BBBB CCCC DDDD)



What if you cannot find a bug during simulation?

- Very likely you have some undefined behavior in your design
  - Race condition in RTL code (blocking vs non-blocking assignment)
  - Incorrect use of ”don't cares”
  - You are not crossing clock domains correctly
  - etc.
- Not so likely:
  - You have triggered a bug in the CAD tools

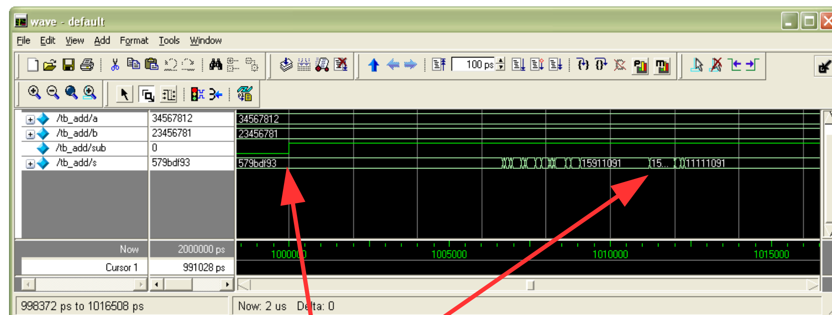
Clock domain crossing

- Why do we need synchronous designs?
  - Race conditions
  - Metastability
- Crossing clock domains
  - (Avoid if possible)
  - Using handshakes
  - Using asynchronous FIFOs
  - Your own solution
    - (Only if you like debugging systems where bugs cannot be deterministically reproduced...)
- Do not forget that the reset signal has to be passed to each clock domain!

Troubleshooting

- Post Place-and-Route (PAR) simulation
  - Generate a new netlist using netgen
  - Simulation done with LUTs and FF

Available for lab0!
make sim_lab0_sdf
See lab webpage



32-bit add/sub example:
Output s takes 15ns to stabilize after `sub 0->1`

Testbenches that work with PAR netlists

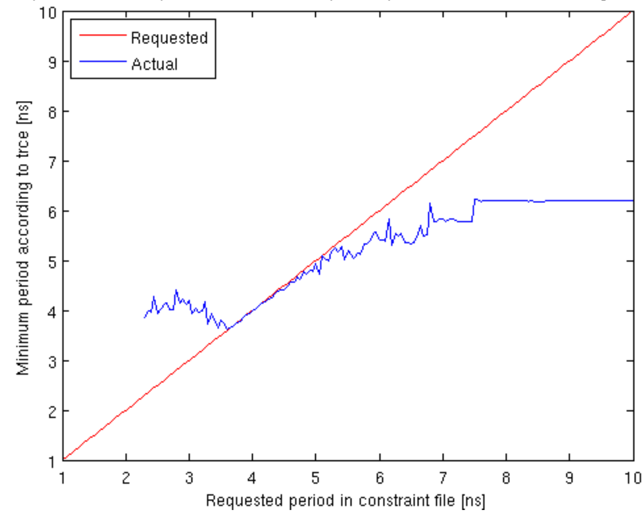
- Avoid violating setup and hold times of flipflops
  - Delay test values
- Test results at the end of the clock cycle
  - Test values at the clock cycle transition, before updates moved on from input flipflops

```
initial begin // Test adder
      @(posedge clk);
      #4; // delay after clockedge
      a <= 5;
      b <= 3;
      @(posedge clk);
      if (result != 8) begin
         $display(”Adder fail”);
         $stop;
     end
end
```


Simulation ok, but still not working?

- Add measurement logic to the FPGA Design
  - Use switches and LEDs
- Chipscope/Signaltap
  - Add logic analyzer function to the FPGA design
  - Store samples in blockRAM or similar
  - Communicate with PC over JTAG
- Warning!
  - Many people think signaltap/chipscope replace simulation. It does not! Better to spend time writing better testbench

Clock cycle constraints effects on result



To get the best out of the FPGA

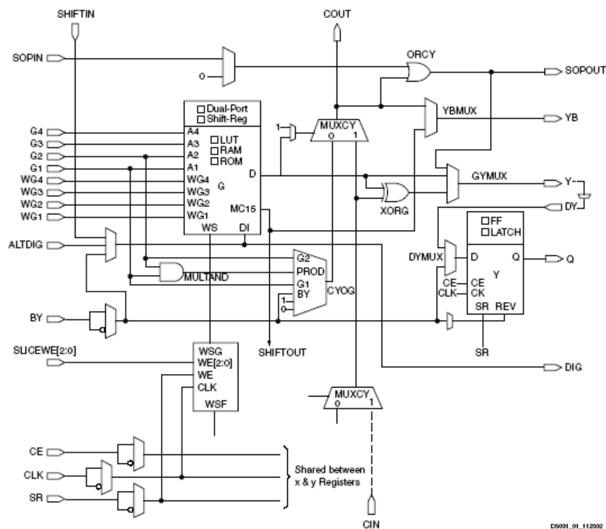
- Understand the architecture
- Use suitable descriptions
- Use available tools to extract implementation information
  - FPGA editor
  - Floorplanner
  - Planahead
  - Datasheets
  - Timing reports

FPGA components

- CLB:s
  - Slices
  - LUT
- Hard blocks
  - Block memory
  - Multipliers
  - I/O units

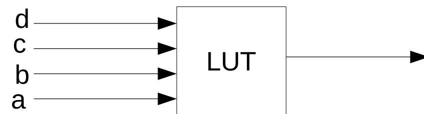
1/2 slice (total 8 of these in one CLB)

- Note
  - 4-input LUT G
  - XORG
  - CYOG
  - MUXCY
  - MULTAND



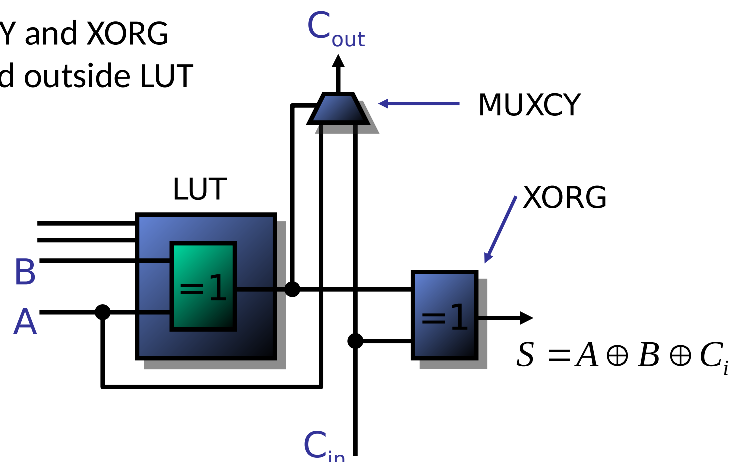
Combinatorial logic using a LUT

- 4 inputs give any logic function of at most 4 inputs



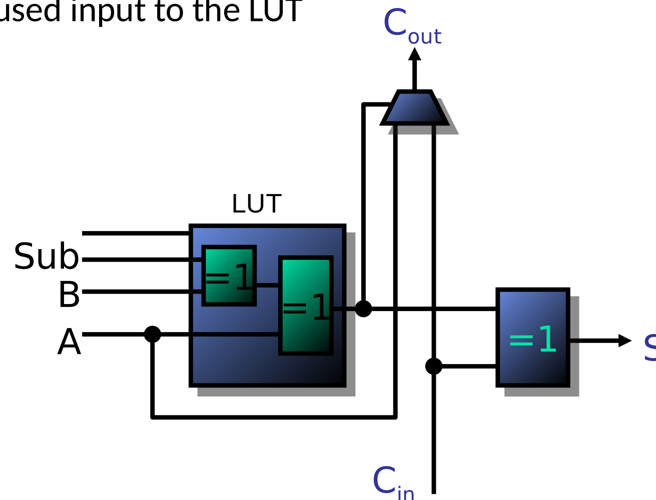
Adders and carrychains in Xilinx FPGAs

- 1 fulladder structure using carry chain acceleration
  - MUXCY and XORG located outside LUT
- 1 LUT/bit



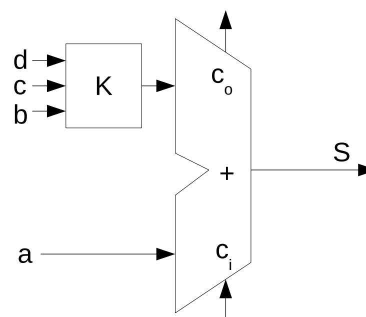
Extend to Add/subtract in Xilinx FPGAs

- Still one unused input to the LUT



Rule of thumb for efficient adders in 4-input LUT based FPGAs

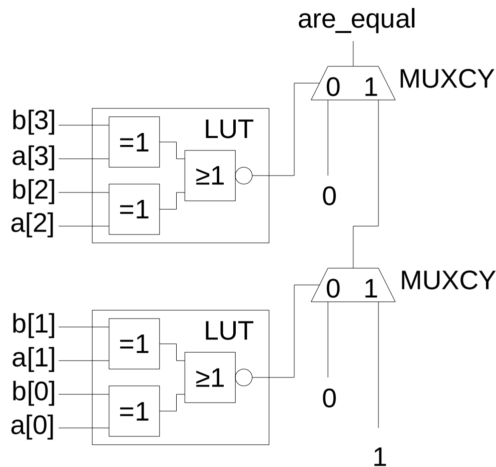
- S = a + K(b,c,d)
- Plain adder
- Adder/subtracter
- 2-to-1 mux and adder
- More strange versions
  - S = (opb | opc | opd) + opa
  - S = (opb & opc) + (opb & opa)
    (Uses MULT_AND located under LUT in slice figure)



2022-11-22 10:36

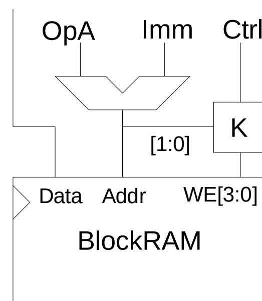
Carry chain for other purposes: Comparators

- Compare 2 bits per LUT
- Compare 4 bits per LUT if one value is constant!

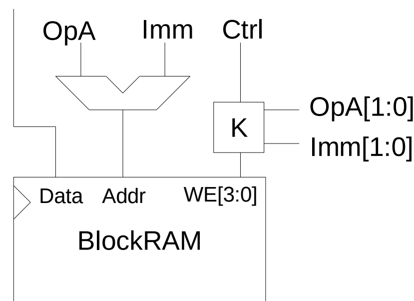


Carry chain drawbacks

- Example: Address calculation selecting one byte memory



WE delayed by carry chain



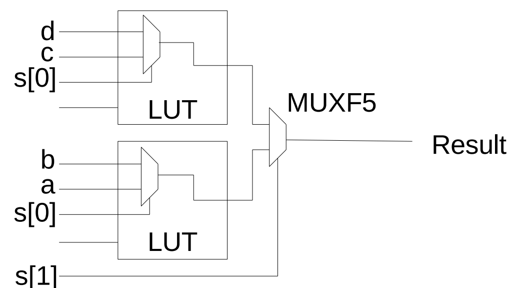
2-bit adder in K using 1 LUT gives faster implementation

- The carry chain itself is extremely fast
- Getting on the chain is not very fast

Multiplexers in FPGAs

- A big difference between ASIC and FPGAs: Multiplexers are cheap in ASIC and expensive in FPGAs
- 4-input LUT: One 2-to-1 mux
- Specialized multiplexers in the slices are used to combine LUTs into larger multiplexers

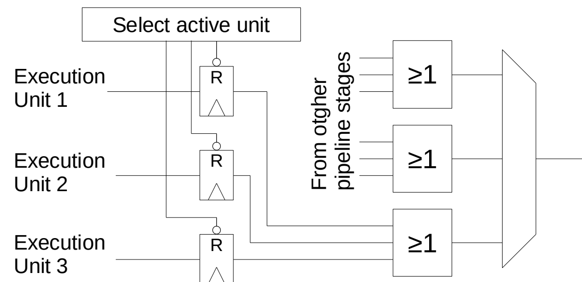
Multiplexers in Xilinx FPGAs



- Possible use of spare input:
  - Invert output, set output to one or zero
  - Tricky variants based on a,b, and s[0]
- How many 4-input LUTs needed for a 4-to-1 mux (without MUXFx components)?

2022-11-22 10:36

Avoiding multiplexers in pipelined designs



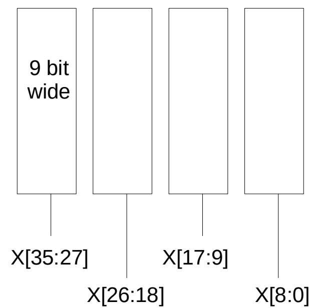
- Multiplexers are costly in FPGAs
- Alternative 1: Use or gates and make sure unused inputs are set to 0 using reset input of flip-flops
- Alternative 2: Use and gates and make sure unused inputs are set to 1. (see MULT_AND as well!)

Memory guidelines

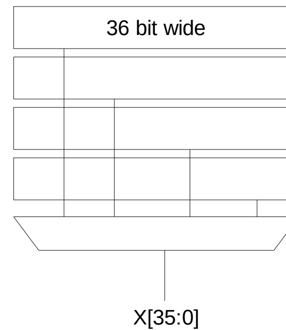
- Standard rule: Large memories should be synchronous
- For high frequency design you want to register the output of the memory as well.
- For power reasons you should not enable the memory unless necessary
  - Double check that your enables work when inferring a memory!
- Smaller memories may be asynchronous if necessary
- You should not have a reset signal for your memory array
  - Easy to forget for shift registers!

Memories larger than one BlockRAM

72 kilobit using 4 BlockRAMs that are 9 bits wide



72 kilobit using 4 BlockRAMs that are 36 bits wide



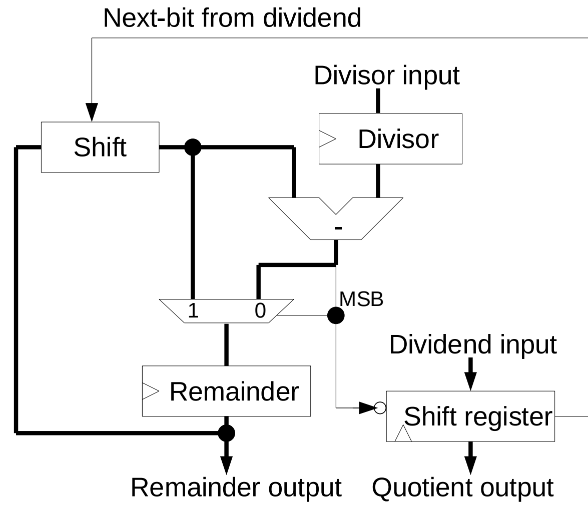
- Why use the right variant? Reduced power consumption!

A case study: A divider for a RISC processor

- Used in a 32-bit RISC processor
- Target frequency: 320 MHz in a Virtex-4 (speedgrade -12)
- Uses restoring division algorithm (basic operations are shift, subtract, and select)
  - Serial computation
  - Very similar to manual division

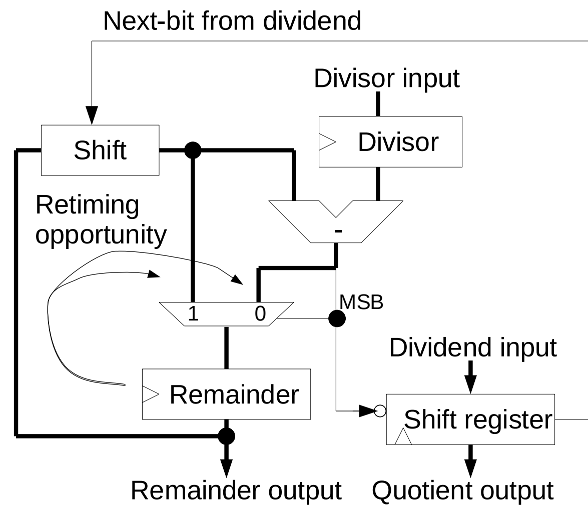
$$\frac{dividend}{divisor} = quotient \times divisor + remainder$$

Initial divider architecture



Initial divider architecture

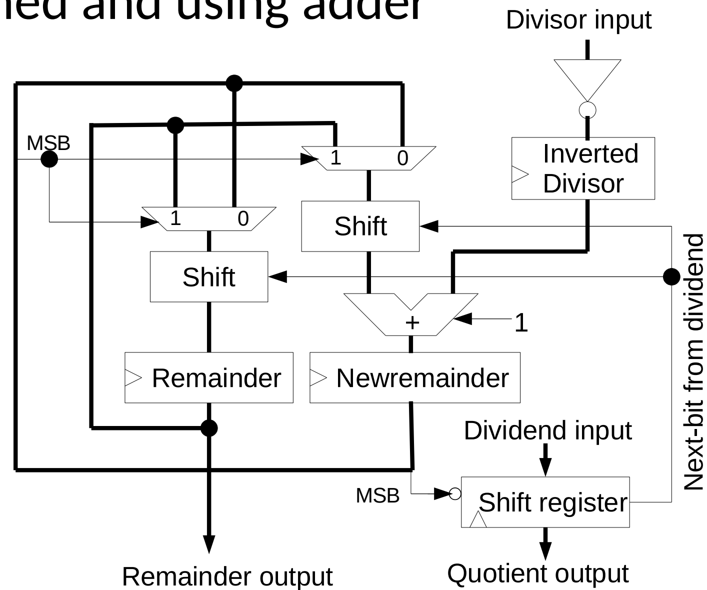
Initial speed:
300 MHz



Issues

- Cannot combine subtracter and 2-to-1 multiplexer!
- Solution: Preprocess divisor and use an addition instead

Retimed and using adder



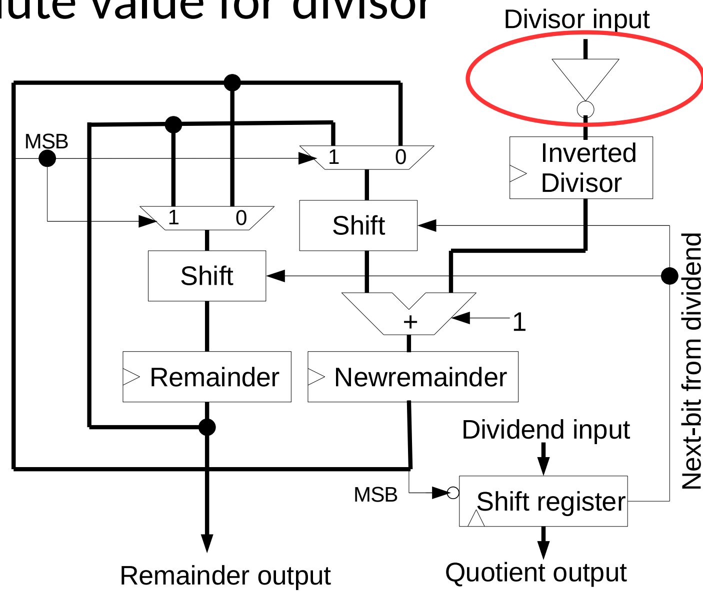
Other issues

- Synthesis tool was too clever
- Manually instantiating the components worked
- Alternatively a complete rewrite of the module worked as well
- Improves clock frequency to 377 MHz (from 300 MHz)

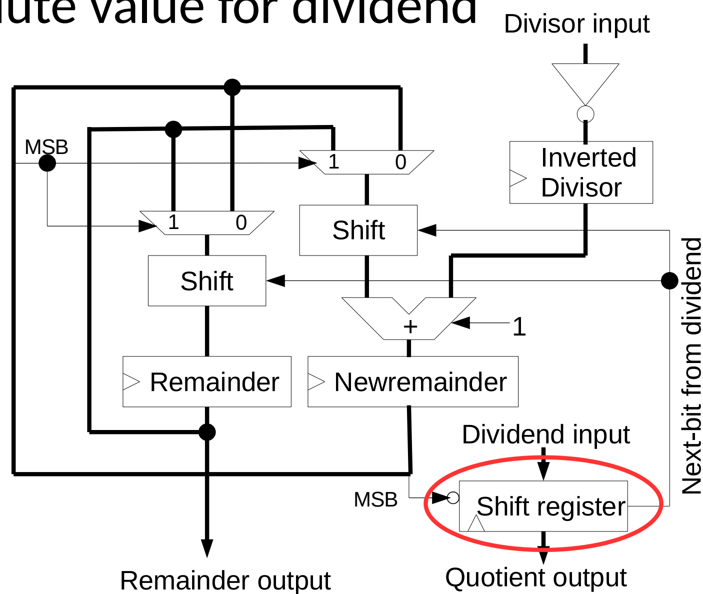
Dealing with negative numbers

- Idea: Take absolute value of dividend and divisor
- Negate quotient and remainder if necessary
- For a 32 bit divider this seems to require around 128 extra LUTs…

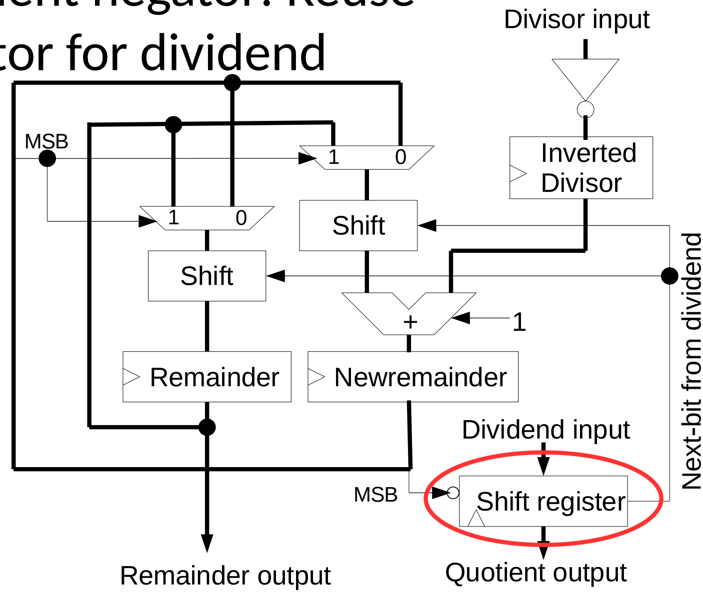
Absolute value for divisor



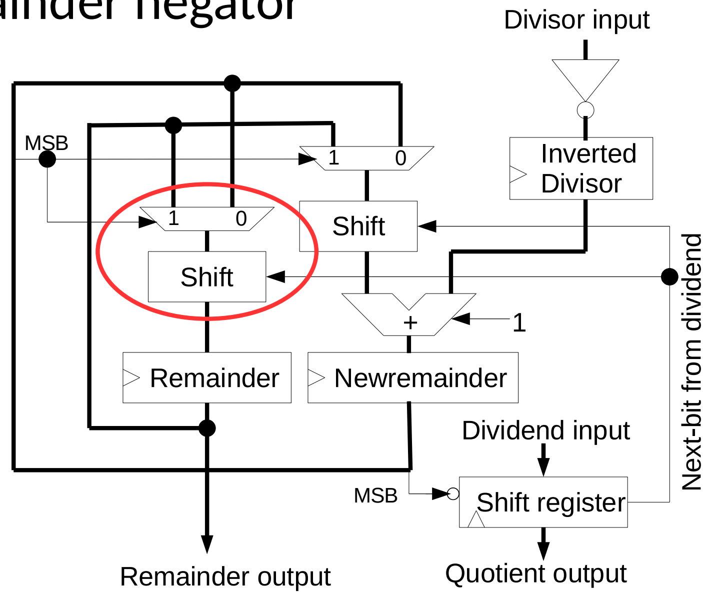
Absolute value for dividend



Quotient negator: Reuse negator for dividend



Remainder negator



Tricky to do in practice

- Required signals for shift register:
  1. Load enable/shift enable
  2. Invert enable
  3. Input data of new dividend
  4. Input data of new dividend (MSB bit)
  5. Current value of register
- 5 inputs to a 4 input LUT?

Tricky to do in practice - Solution

- Solution: Skip MSB of dividend input for ABS operation
- Always invert the dividend, only add 1 as a carry in if appropriate
  - This can be implemented by adding a few extra LSB bits
  - If we had a positive value we can compensate for the inversion at shift out
  - We can even add a control bit to select between signed/unsigned division
- Manual instantiation was necessary to actually implement this

Results for Virtex-4, speedgrade 12

- Unoptimized, unsigned: 300 MHz, 107 LUTs
- Retimed, unsigned: 377 MHz, 140 LUTs
- Retimed, signed: 361 MHz, 151 LUTs
- Retimed, signed or unsigned: 363 MHz, 153 LUTs

Manual instantiation

- Last resort when synthesis attributes and rewriting the RTL code does not work
- Not portable between FPGA vendors
  - Suprisingly portable to ASIC however

Manual instantiation of flip-flops

- Allows you to ensure that the correct signals are corrected to the D, CE, and SR inputs
  - XST (Xilinx own synthesis tool, not used in the lab) often seem to select the wrong input for SR
  - Background: SR input is quite slow compared to D input
- Can sometimes be avoided by rewriting the code or using synthesis attributes
- Often easier to just instantiate flip-flop primitives directly

Manual instantiation of Memories and DSP Blocks

- Well documented in various application notes

Synthesis attributes

- A convenient way to force the synthesis tool to do what you mean
- In VHDL:

```
attribute keep :  string;
attribute keep of mysignal:  signal is "TRUE"
```

- In Verilog:

```
(* KEEP = "TRUE" *) wire mysignal;
```

- Note: Synthesis attributes discussed here are for XST, not Precision!
  - (Read the Precision manual)

Synthesis attribute KEEP

- Preserves the selected signal
- Use case:
  - The synthesis tool makes a bad optimization decision.
  - By using KEEP you can ensure that a certain signal is not hidden inside a LUT and hence guide the optimization process

KEEP example from a display controller

```
wire inimagey = (yctr > 31) && (yctr < 192);
wire inimagex = (xctr > 15) && (xctr < 26);
…
always @(posedge clk) begin
  if (inimagey && (xctr == 15) ) begin
  …
  end else if(inimagey && (xctr == 26)) begin
  …
  if (inimagey && (xctr == 15) ) begin
  …
  end else if(inimagey && (yctr[2:0] == 7)) begin
  …
```

- Problem: Synthesis tool merged inimagey test with other tests in suboptimal way

Solution: Force inimagey and inimagex to be separate signals

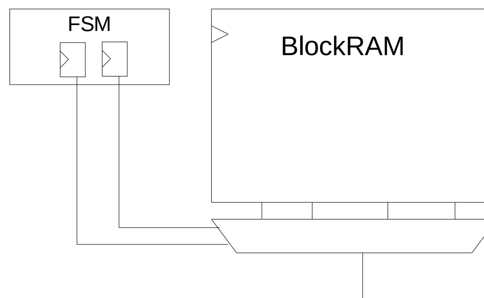
```
(* KEEP = "TRUE" *) wire inimagey;
(* KEEP = "TRUE" *) wire inimagex;

assign inimagey = (yctr > 31) && (yctr < 192);
assign inimagex = (xctr > 15) && (xctr < 26);
```

- Saved area in an area constrained situation
- Especially important when targetting both CPLD and FPGAs with a single IP core

SIGNAL ENCODING attribute

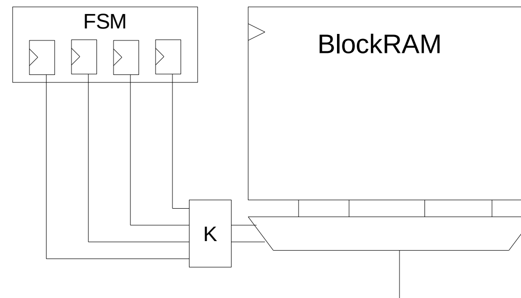
- Allows you to select encoding for state machines
- Useful when synthesis tool make suboptimal state machine encoding choices
- (Alternatively: You can disable FSM optimization if you **really** want to)

Example: Memory byte select in a processor



- Signal encoding specified 2 FF, 4 states.
- Two signals into mux control signal

Example: Memory byte select in a processor

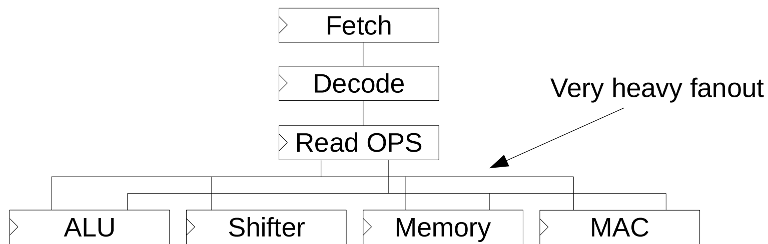


- Heuristics in the synthesis tool selected one-hot coding for the FSM…

EQUIVALENT REGISTER REMOVAL attribute

- Allows you to specify that certain registers should not be optimized away.
- Perfect when you do not want the synthesis tool to touch your carefully optimized (duplicated) flip-flops

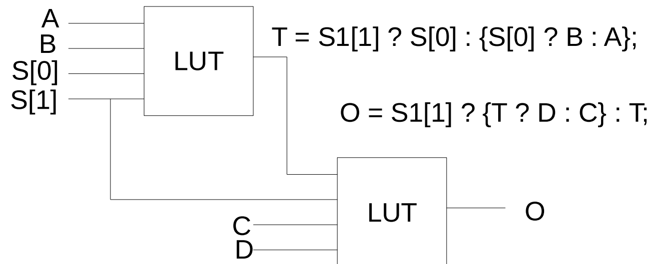
Example: Operand bus in a processor



- Problem: Manual register duplication in read operand stage is removed by synthesis tool
- Solution: Disable optimization locally by setting EQUIVALENT_REGISTER_REMOVAL to "no"

4-to-1 multiplexer using two LUT4

4-to-1 multiplexer using two LUT4



Conclusions

- By mapping your design to the FPGA in an efficient manner you can significantly improve the performance of your design
- Keep this in mind early in the design phase.
- (However, don't optimize unless you really need to.)

www.liu.se