TSEA44 - Design for FPGAs

Andreas Ehliar <ehliar@isy.liu.se>

2014-11-18

Administrative issue

- Did the email regarding no lecture this Wednesday reach you?

Outline

- What can go wrong?
    - Design mistakes
    - Synthesis errors
    - Runtime errors
- Crossing clock domains
    - Handshaking
    - Asynchronous FIFOs
- FPGA architecture
    - Adders
    - Multiplexers
    - Memories
    - Multipliers
- Case study

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

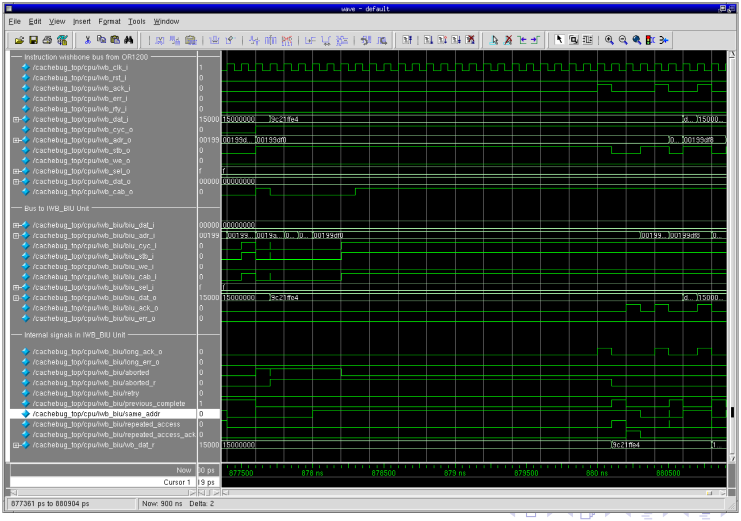
- Use checkpointing to reduce/eliminate the need for logging
  - Add no signals to wave window (and log for that matter)
  - Modify UART so printouts are displayed in the transcript window (using `$display()`)
  - `run 100ms; checkpoint 100ms.chk`
  - `run 100ms; checkpoint 200ms.chk`
  - `run 100ms;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

Handling long simulation runtimes

- 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)

So what was the bug?

Wrong content:	AAAA	AAAA	CCCC	DDDD
Correct content:	AAAA	BBBB	CCCC	DDDD



```
— Instruction wishbone bus from OR1200 —
/cachebug_top/cpu/iwb_clk_i                       1
/cachebug_top/cpu/iwb_rst_i                       0
/cachebug_top/cpu/iwb_ack_i                       0
/cachebug_top/cpu/iwb_err_i                       0
/cachebug_top/cpu/iwb_rty_i                       0
/cachebug_top/cpu/iwb_dat_i                       15000   15000000   9c21ffe4   d...   15000...
/cachebug_top/cpu/iwb_cyc_o                       0
/cachebug_top/cpu/iwb_adr_o                       00199   00199d...   00199df0   0...   00199df8
/cachebug_top/cpu/iwb_stb_o                       0
/cachebug_top/cpu/iwb_we_o                        0
/cachebug_top/cpu/iwb_sel_o                       f       f
/cachebug_top/cpu/iwb_dat_o                       00000   00000000
/cachebug_top/cpu/iwb_cab_o                       0
— Bus to IWB_BIU Unit —
/cachebug_top/cpu/iwb_biu/biu_dat_i               00000   00000000
/cachebug_top/cpu/iwb_biu/biu_adr_i               00199   00199...   0019a...   0...   0...   00199df0   00199...   00199df8   0...
/cachebug_top/cpu/iwb_biu/biu_cyc_i               0
/cachebug_top/cpu/iwb_biu/biu_stb_i               0
/cachebug_top/cpu/iwb_biu/biu_we_i                0
/cachebug_top/cpu/iwb_biu/biu_cab_i               0
/cachebug_top/cpu/iwb_biu/biu_sel_i               f       f
/cachebug_top/cpu/iwb_biu/biu_dat_o               15000   15000000   9c21ffe4   d...   15000...
/cachebug_top/cpu/iwb_biu/biu_ack_o               0
/cachebug_top/cpu/iwb_biu/biu_err_o               0
— Internal signals in IWB_BIU Unit —
/cachebug_top/cpu/iwb_biu/long_ack_o              0
/cachebug_top/cpu/iwb_biu/long_err_o              0
/cachebug_top/cpu/iwb_biu/aborted                 0
/cachebug_top/cpu/iwb_biu/aborted_r               0
/cachebug_top/cpu/iwb_biu/retry                   0
/cachebug_top/cpu/iwb_biu/previous_complete       1
/cachebug_top/cpu/iwb_biu/same_addr               0
/cachebug_top/cpu/iwb_biu/repeated_access         0
/cachebug_top/cpu/iwb_biu/repeated_access_ack     0
/cachebug_top/cpu/iwb_biu/wb_dat_r                15000   15000000   9c21ffe4   1...

Now        00 ps
Cursor 1   19 ps
877361 ps to 880904 ps    Now: 900 ns   Delta: 2
```

What if you can't 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 (more on this later)

Clock domain crossings

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

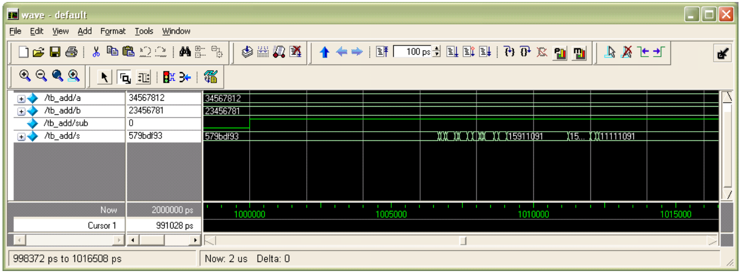
An example of a synthesis bug

- RTL simulation works fine
- Real hardware dosen't work
- (There are no clock domain crossings involved)

```
// Work around possible synthesis bug in an
// old version of synthesis tool
logic signed [27:0] foo;
logic signed [27:0] bar;
assign        foo = x2[2] * S6;
assign        bar = x2[3] * C6;
// ********* Third stage ************
always_ff @(posedge clk_i) begin
   if (en) begin
      x3_0 <= x2[0] + x2[1];
      x3_1 <= x2[0] - x2[1];
      x3_2 <= x2[2]*C6 + x2[3]*S6;
      x3_3 <= bar - foo;
```

Post synthesis simulation

- Synthesize the design
- Convert the synthesized design back to Verilog (`netgen`)
- The simulation is done using FPGA components like LUTs instead of at the behavioral level
- Can even be done after place and route if you want to simulate timing (very useful for power simulations!)



```
/tb_add/a      34567812     34567812
/tb_add/b      23456781     23456781
/tb_add/sub    0
/tb_add/s      579bdf93     579bdf93   15911091  15...  11111091
Now            2000000 ps
Cursor 1       991028 ps
998372 ps to 1016508 ps        Now: 2 us   Delta: 0
```

Writing testbenches for post synthesis netlists 1

```
// Original testbench code

initial begin // Test adder
    @(posedge clk);
    a <= 5;
    b <= 3;
    @(posedge clk);
    if (result !== 8) begin
        $display("Adder has funny ideas of addition");
        $stop;
    end
end

// This code will probably lead to unknown values all over
// the simulation.
```

Writing testbenches for post synthesis netlists 1

```
// Modified testbench

initial begin // Test adder
    @(posedge clk);
    #4; // (Use an appropriate delay so the setup/hold
        //  of flip-flops in the circuit are honored.)
    a <= 5;
    b <= 3;
    @(posedge clk);
    // However, we should still check the result on the
    // clock edge!
    if (result !== 8) begin
        $display("Adder has funny ideas of addition");
        $stop;
    end
end
```

Dealing with bugs that you are unable to reproduce in simulation

- Add extra debugging logic to the FPGA design
- Chipscope/SignalTap
  - Logic analyzer that store samples to built in BlockRAMs
  - Communicates with the host computer via JTAG
- Warning!
  - Many people use ChipScope/SignalTap as a substitute for a comprehensive self-checking testbench.
  - Don't! You will most likely waste time better spent writing high quality testbenches.

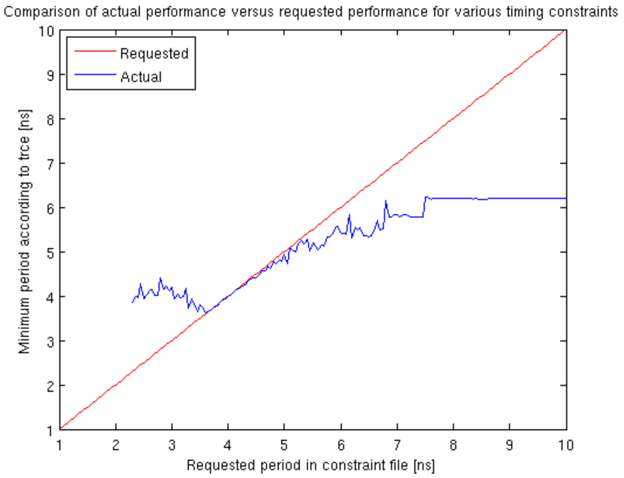
Single event upset - SEU

- Very important in aerospace applications
- Can be important for ground based safety critical applications.
- (Fortunately not a problem in this course)
- Some (partial) solutions:
  - Triplicate logic
  - Reprogram FPGA often

Now for something else...

- Adapting designs to FPGAs
- Why?
  - Clock frequency
  - Area
  - Power
- Target FPGA architecture: Xilinx FPGAs with 4 input LUTs (such as Virtex-II)

Determining the maximum frequency of a design



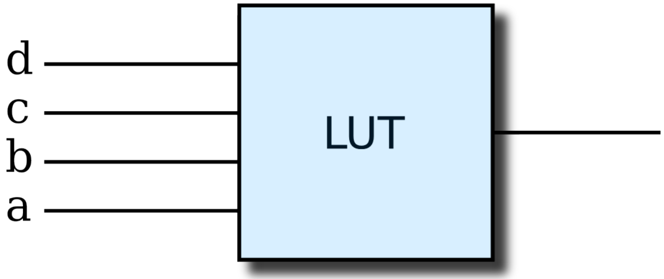
FPGA architecture

- Knowing the FPGA architecture makes it much easier to optimize your code
- FPGA editor, floorplanner, Planahead, datasheets, timing reports, XDL

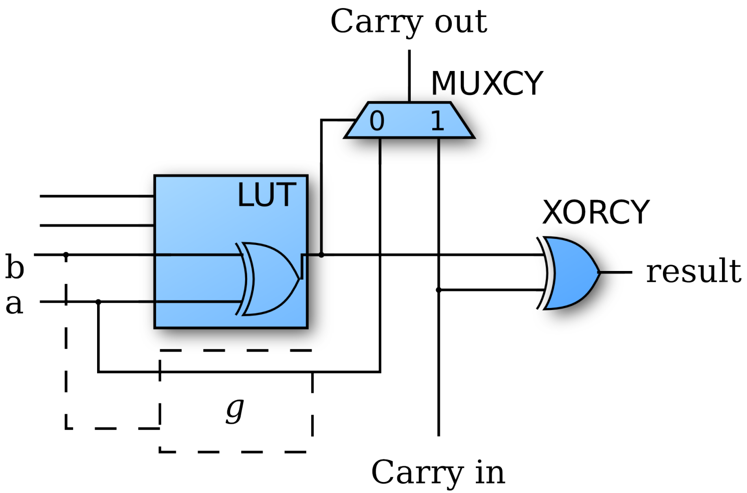
FPGA components

- Slices
- CLBs
- Hard blocks (memories, multipliers, I/O, etc)

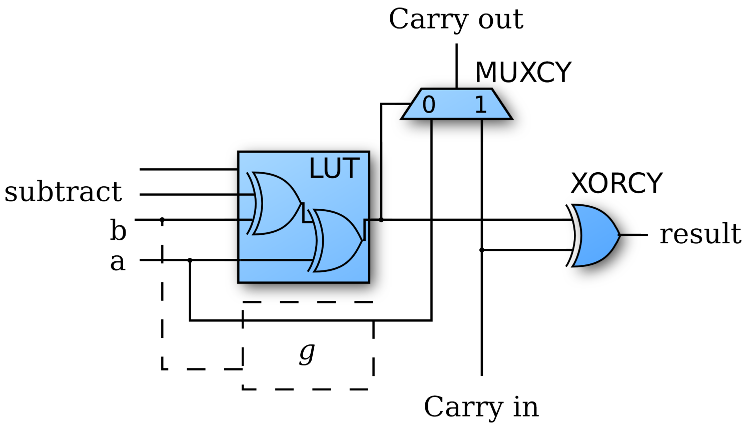
Combinatorics using a Lookup Table (LUT))



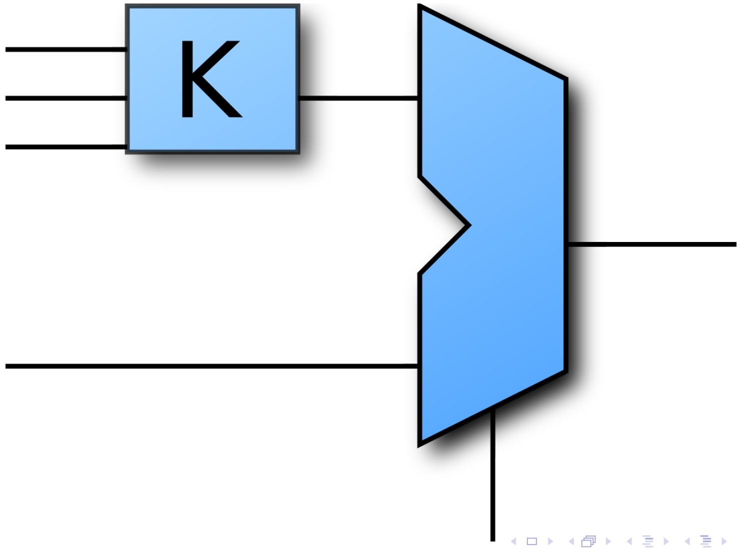
Adders and carrychains in Xilinx FPGAs



Add/subtract using one LUT/bit in Xilinx FPGAs



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

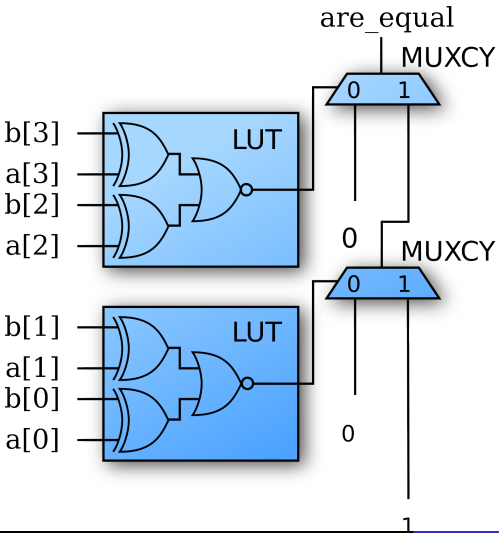


Summary: Adders in 4-input LUT based architectures

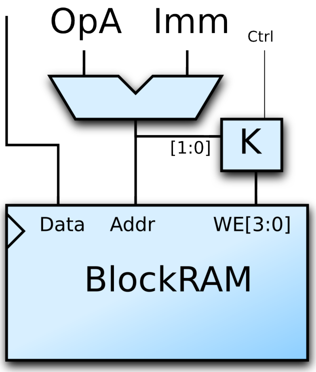
- Plain adder
- Adder/subtracter
- 2-to-1 mux and adder
- Some more esoteric versions:
  - `result = (opa | opb | opc) + opd;`
  - `result = (opa & opb) + (opc & opd);` (Using MULT_AND)

Using the carry chain for other purposes: Comparators

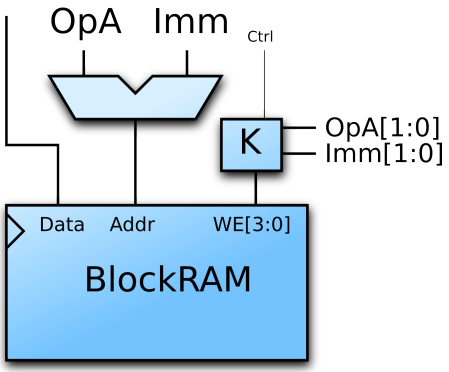
- Comparing 2 bits per LUT
- Comparing 4 bits per LUT if one value is constant!



Carry chain drawbacks



WE signal delayed by going through carry-chain



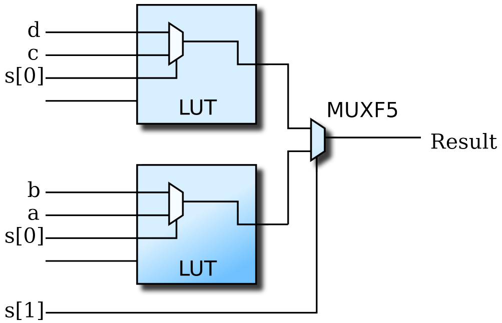
2 bit adder implemented in K using one level LUT logic leads to a faster implementation

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

Multiplexers in FPGAs

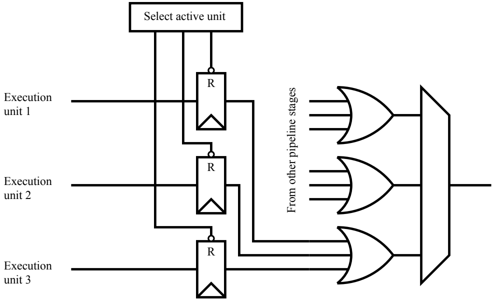
- A big difference between ASIC and FPGAs: Multiplexers are cheap in ASIC and expensive in FPGAs
- 4-input LUT: One 2-to-1 mux
- Specialized multiplexers in the slices are used to combine LUTs into larger multiplexers

Multiplexers in Xilinx FPGAs



- Possible uses for spare input:
    - Invert output, set output to one or zero
    - Tricky variants based on a, b, and s[0]
- Trivia question: How many 4-input LUTs do you need to create a 4-to-1 mux (without MUXFx components)?

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!)

Types of memories

- Flip-flops
- Distributed memories
- BlockRAMs

Distributed memories

- A LUT (or a combination of LUTs) can be used for small memories
  - Synchronous write
  - Asynchronous read
- Possible configurations
  - 16x1 bit wide, 1 read/write port: 1 LUT
  - 32x1 bit wide, 1 read/write port: 2 LUT
  - 16x1 bit wide, 1 read port, 1 write port: 2 LUTs
- Other combinations of note:
  - 16x1 bit wide, 2 read ports, 1 write port: 4 LUTs
  - A LUT can also be used as a shift register with (up to) 16 bits

Block RAMs

- Features:
  - 18 kbit memory
  - True dual port memory with independent clock domains for each port
  - Configurable bit width (1, 2, 4, 9, 18, or 36 bits)
  - Read before write, read after write
- Requirements:
  - **Synchronous read and write**
- For an example: See mon_prog_bram.sv

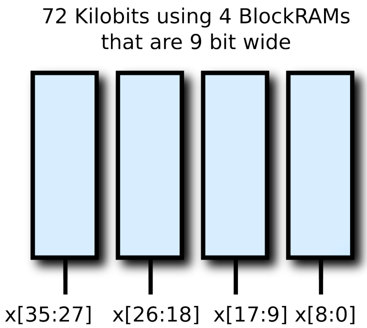
Multipliers

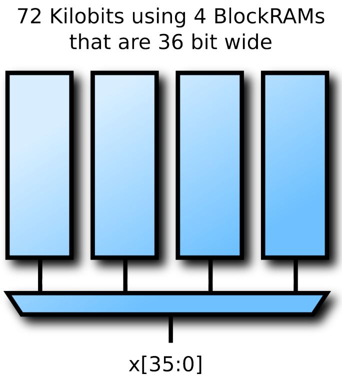
- The Virtex-II FPGA also contains numerous multipliers
- Features:
  - Combinational 18x18 bit two's complement multiplier
- (Newer FPGAs have more advanced *"DSP blocks"* where the multiplier is combined with adders and registers)

Memory guidelines

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

Memories larger than one BlockRAM

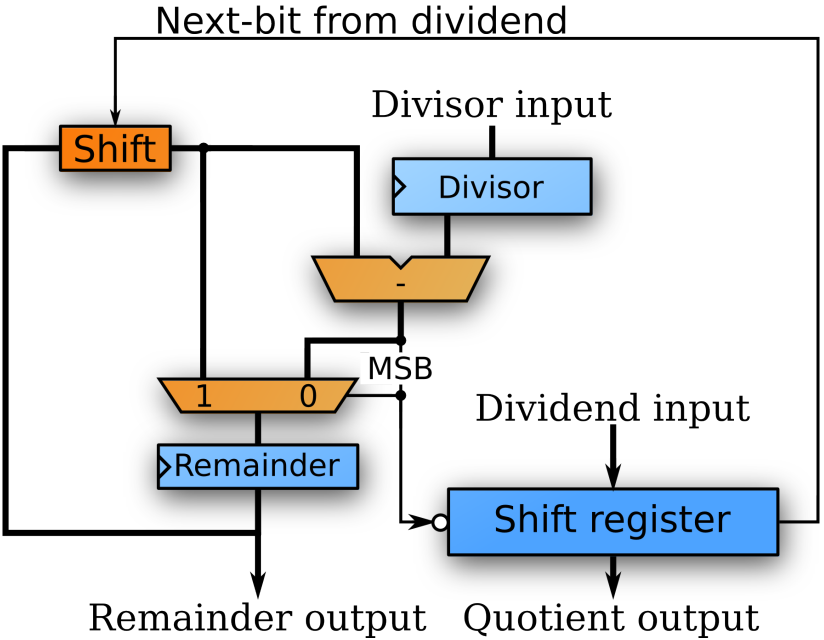


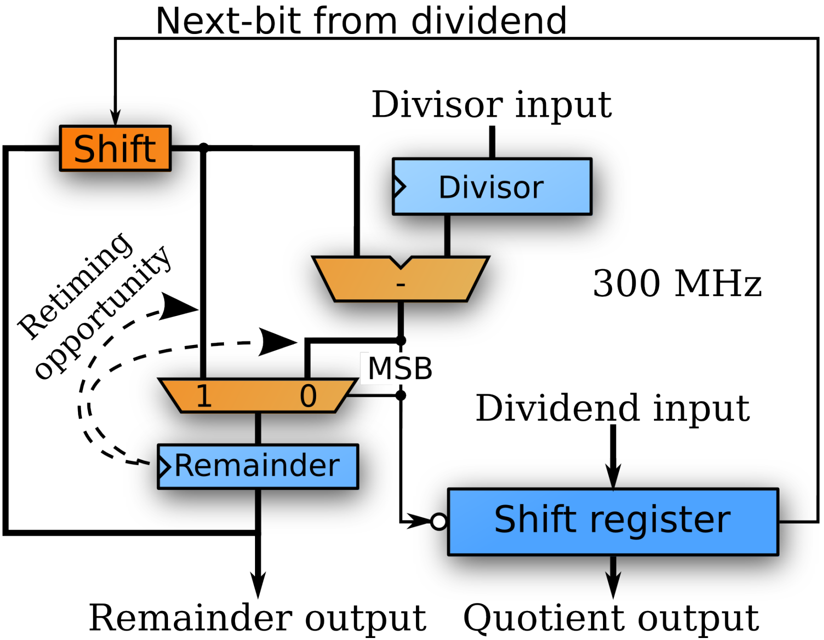


- Why use the right variant? Reduced power consumption!

A case study: A divider for a RISC processor

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

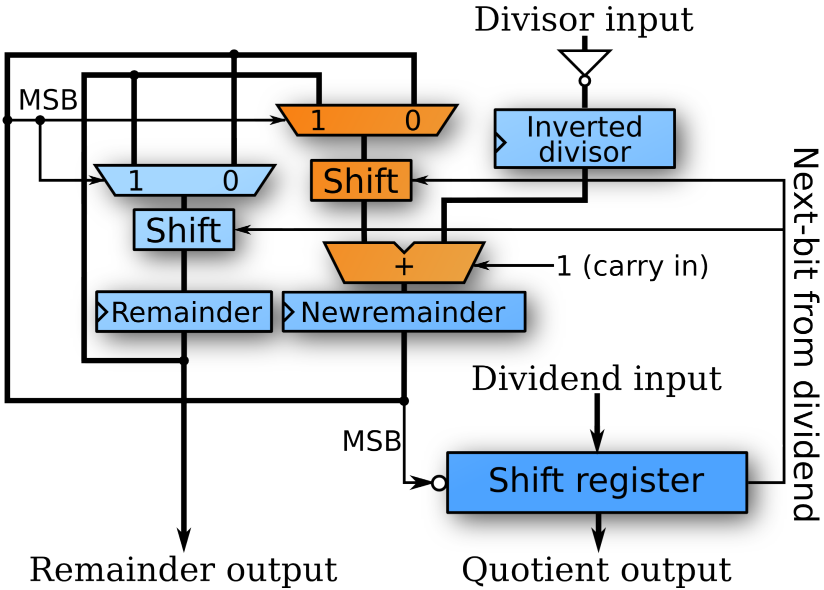

Next-bit from dividend
Divisor input
Shift
Divisor
-
MSB
1
0
Dividend input
Remainder
Shift register
Remainder output
Quotient output



Next-bit from dividend
Divisor input
Shift
Divisor
-
300 MHz
Retiming opportunity
MSB
1
0
Dividend input
Remainder
Shift register
Remainder output
Quotient output

Issues

- Can't combine subtracter and 2-to-1 mux!
- Solution: Preprocess divisor and use an adder instead.


Divisor input
MSB
1
0
1
0
Inverted divisor
Shift
Shift
+
1 (carry in)
Remainder
Newremainder
Next-bit from dividend
Dividend input
MSB
Shift register
Remainder output
Quotient output

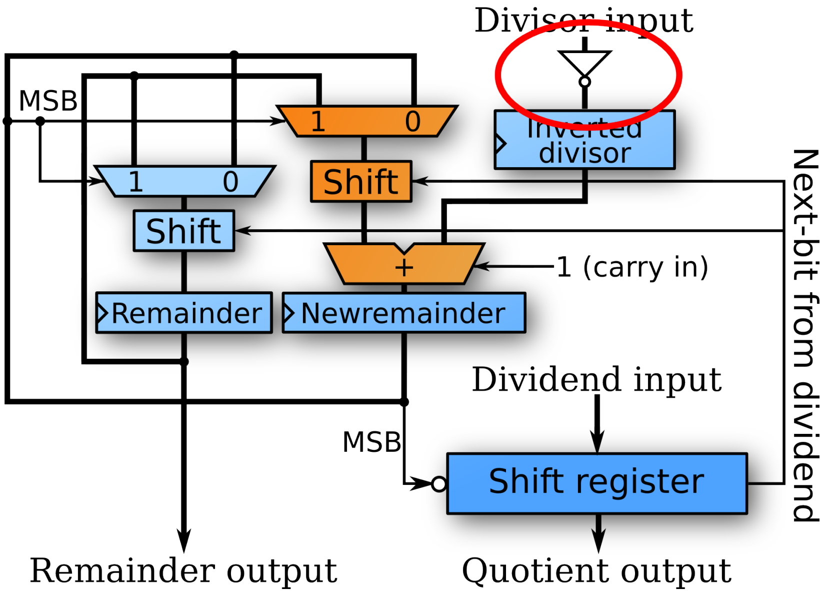
Other Issues

- 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)

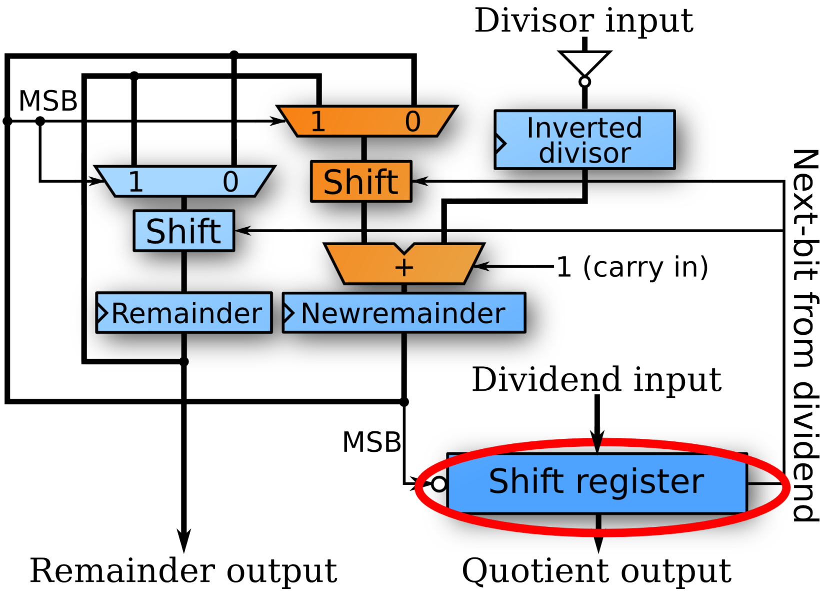
Dealing with negative numbers

- 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...

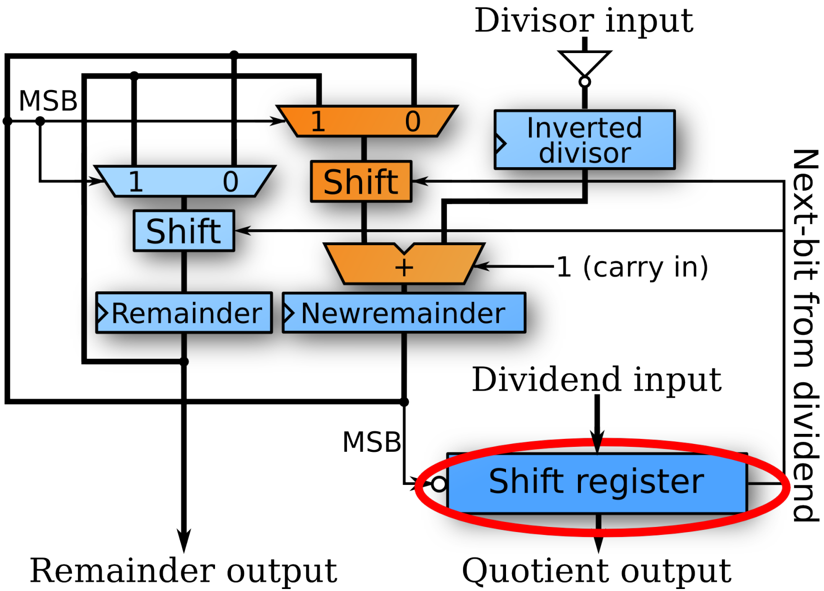
Absolute value for divisor



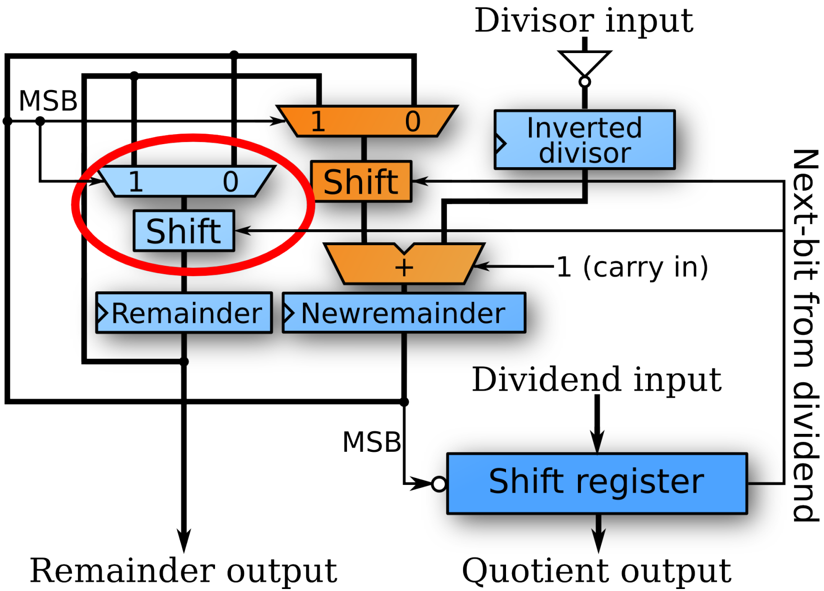
Absolute value for dividend



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

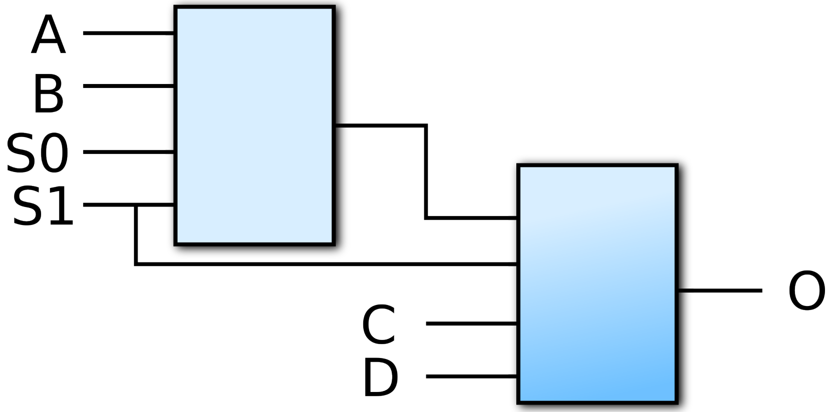
Results for Virtex-4, speedgrade 12

- 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
- Interested? Download the code at `http://www.da.isy.liu.se/~ehliar/ae_instlib/`

Manual instatiation

- Last resort when synthesis attributes and rewriting the RTL code doesn't work
- Not portable between FPGA vendors
  - Surprisingly portable to ASIC however

4-to-1 mux with two LUT4



Manual instantiation of flip-flops

- Allows you to ensure that the correct signals are corrected to the D, CE, and SR inputs
  - XST 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 LUTs

- Often much harder than to rewrite the code or using synthesis attributes
- A good idea to hide instantiation details in a library if possible
  - Carry chain instantiation (MUXCY, XORCY, generate loop, etc)
  - Large multiplexers (MUXF5, MUXF6, MUXF7, etc)
- See `http://www.da.isy.liu.se/~ehliar/ae_instlib/` for the library I'm using.

Manual instantiation of Memories and DSP blocks

- Well documented in various app-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

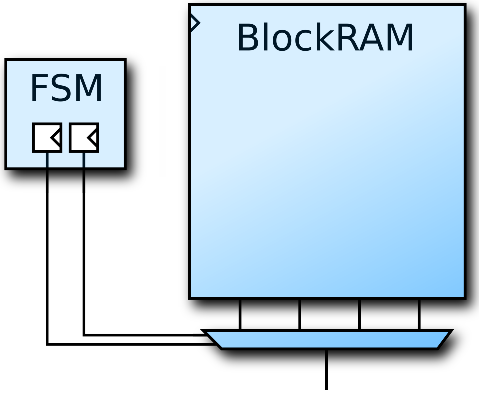
```
(* KEEP = "TRUE" *) wire inimagey;
(* KEEP = "TRUE" *) wire inimagex;

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

- Allowed me to save area in an area constrained situation
  - Especially important when targetting both CPLD and FPGAs with a single IP core

SIGNAL_ENCODING attribute

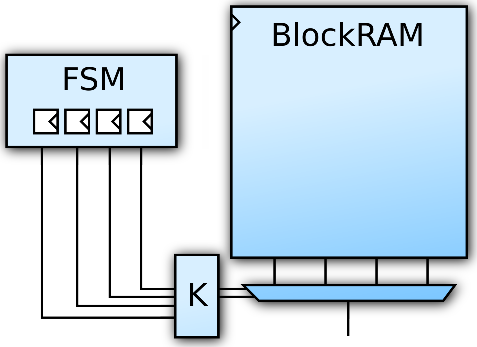
- Allows you to select encoding for state machines
- Useful when synthesis tool make suboptimal state machine encoding choices
- (Alternatively: You can disable FSM optimization if you *really* want to)

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

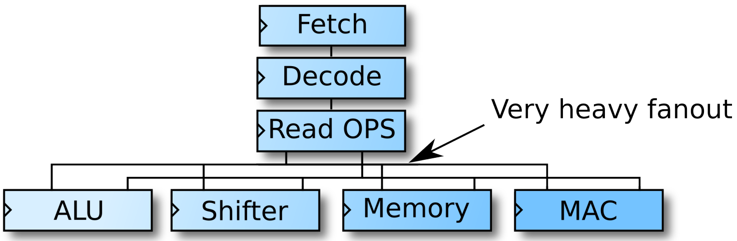


- Heuristics in synthesis tool selected one-hot coding for FSM...

EQUIVALENT_REGISTER_REMOVAL attribute

- Allows you to specify that certain registers should *not* be optimized away.
- Perfect when you don't 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"

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.)
- Looking for master's thesis projects?
    - Talk to me if you are interested in for example FPGA related projects