505 Registers or Bust

Wilson Snyder
Nauticus Networks
wsnyder@wsnyder.org
http://veripool.com

Agenda

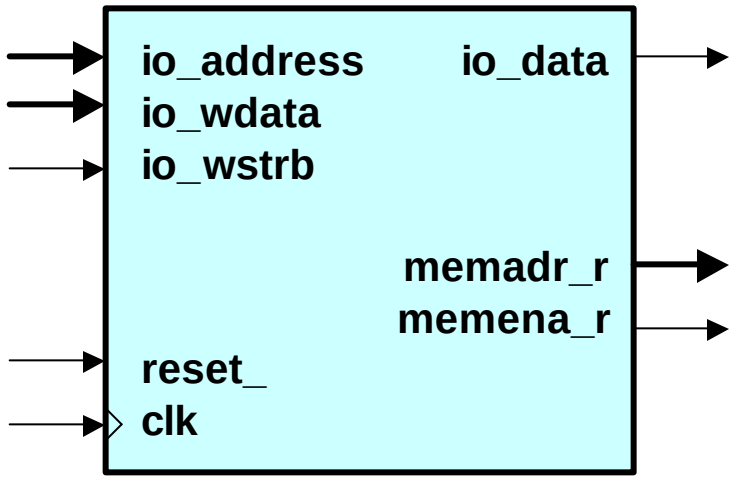
- Introduction
- CSR RTL Optimizations
  - Hiding Constants
  - Loading using Concatenation
  - Reset Area
- Simulation Speedups
- Vregs - Extracting CSRs from Documentation
  - Definitions
  - Enumerations
  - Classes
  - Registers
- Obtaining Vregs

Introduction

- Last Tapeout (Conexant, Inc.)
  - 505 CSR Layouts Documented
  - Representing 190KB of storage
- Maintaining the spec relative to RTL design was painful
- This presentation covers the techniques we used to reduce the engineering cost of these CSRs:
  - Simplify RTL code
  - Remove redundant information
  - Speed up simulation
  - Spec as golden reference

Sample CSR Code (1 of 2)

```
module regex_1 (/*AUTOARG*/);
   input            clk;
   input            reset_;          // Synchronous reset
   // I/Os from the CPU or memory bus
   input  [31:2]    io_address;      // I/O Address bus (byte address)
   input  [31:0]    io_wdata;        // I/O Write data bus
   input            io_wstrb;        // I/O Write strobe pulse
   output [31:0]    io_rdata;        // I/O Read data bus
   // The I/O CSR values
   output [31:0]    memadr_r;        // MemAdr CSR (at address 4)
   output           memena_r;        // MemCtl CSR (at address 0[0])
   /*AUTOREG*/
```



Sample CSR Code (2 of 2)


```
   //==== Writing I/O regs
   always @ (posedge clk) begin
       if (~reset_) begin
             memadr_r <= 31'h0;
             memena_r <= 1'b0;
       end
       else begin
             if (io_wstrb && {io_address[31:2],2'b00} ==
                      32'h0000_0000) begin
                memena_r <= io_wdata[0];
             end
             if (io_wstrb && {io_address[31:2],2'b00} ==
                      32'h0000_0004) begin
                memadr_r <= io_wdata[30:0];
             end
       end
    end
    //==== Reading I/O regs
    always @ (/*AS*/) begin
       casex ({io_address[31:2],2'b00})
            32'h0000_0000: io_rdata = {30'b0, memena_r};
            32'h0000_0004: io_rdata = {1'b0, memadr_r};
            default:       io_rdata = 32'h0; // Illegal address
       endcase
    end
endmodule
```

Hide Constants

```
`define RA_MemEna   32'h0000_0000
`define RA_MemAdr   32'h0000_0004
//==== Writing I/O regs
always @ (posedge clk) begin
    if (~reset_) begin
          memadr_r <= 31'h0;
          memena_r <= 1'b0;
    end
    else begin
          if (io_wstrb && {io_address[31:2],2'b00} ==
                `RA_MemEna    ) begin
             memena_r <= io_wdata[0];
          end
          if (io_wstrb && {io_address[31:2],2'b00} ==
                `RA_MemAdr    ) begin
             memadr_r <= io_wdata[30:0];
          end
    end
end
//==== Reading I/O regs
always @ (/*AS*/) begin
    casex ({io_address[31:2],2'b00})
          `RA_MemEna:    io_rdata = {30'b0, memena_r};
          `RA_MemAdr:    io_rdata = {1'b0, memadr_r};
         default:        io_rdata = 32'h0; // Illegal address
    endcase
end
```

Constant Hiding

All constants in the code should be extracted into defines.

Concatenation w/ "unused" signal

```
`define RWIRES_MemEna     {unused[31:1], memena_r}
`define RWIRES_MemAdr     {unused[31], memadr_r}
always @ (posedge clk) begin
    if (~reset_) begin
          memadr_r <= 31'h0;
          memena_r <= 1'b0;
    end
    else begin
          if (io_wstrb && {io_address[31:2],2'b00} ==
                `RA_MemEna) begin
             RWIRES_MemEna <= io_wdata;
          end
          if (io_wstrb && {io_address[31:2],2'b00
                `RA_MemAdr) begin
             RWIRES_MemAdr <= io_wdata;
          end
    end
    unused = 32'h0;
 end
 //==== Reading I/O regs
 always @ (/*AS*/) begin
    unused = 32'h0;
    casex ({io_address[31:2],2'b00})
         `RA_MemEna:    io_rdata = RWIRES_MemEna;
         `RA_MemAdr:    io_rdata = RWIRES_MemAdr;
         default:       io_rdata = 32'h0; // Illegal address
    endcase
 end
```

Concatenation

Allows same definition to be used for reading and writing, and places register to signal mapping in one place. Unused is set to zero to prevent any logic from being made during synthesis.

Synchronous Reset Area

```
always @ (posedge clk) begin
    if (~reset_) begin
          memadr_r <= 31'h0;
          memena_r <= 1'b0;
    end
    else begin
          if (io_wstrb && {io_address[31:2],2'b00} ==
                `RA_MemEna) begin
             RWIRES_MemEna <= io_wdata;
          end
          if (io_wstrb && {io_address[31:2],2'b00} ==
                `RA_MemAdr) begin
             RWIRES_MemAdr <= io_wdata;
          end
    end
    unused = 32'h0;
end
```

Sync Reset

We can zero the data at the "top" of the chip and just load all registers on reset.

```
wire [31:0] io_wdata_zrst = {32{reset_}} & io_wdata;
always @ (posedge clk) begin
    `define WRREGTEST ~reset_ || io_wstrb && {io_address[31:2],2'b00}
    if (WRREGTEST == `RA_MemEna) RWIRES_MemEna <= io_wdata;
    if (WRREGTEST == `RA_MemAdr) RWIRES_MemAdr <= io_wdata;
    unused = 32'h0;
end
```

Direct Routines (1 of 2)

- Initializing large simulations can be a problem
  - 190KB of CSR data => 20 minutes of VCS simulation time
- Eliminate CSR write time!
  - Most time is in clocking the model
  - Have tasks for read/write of CSR in zero time:

    `direct_read(adr), direct_write(adr, data)`
- Usage:
  - Verify direct functions work the same as non-direct functions
  - Tests can then direct_write to initialize
  - Software initialization can be trapped to use the same tasks.
  - 20 minutes is now ~20 seconds.

Direct Routines (2 of 2)


```
task direct_write;
   input [31:0] address;
   input [31:0] wdata;
   reg   [31:0] unused;
   begin
     unused = 32'h0;
     case (address)
        `RA_MemEna: `RWIRES_MemEna <= wdata;
        `RA_MemAdr: `RWIRES_MemAdr <= wdata;
     endcase
   end
endtask


task direct_read;
   input  [31:0] address;
   output [31:0] rdata;
   reg    [31:0] unused;
   begin
     unused = 32'h0;
     case (address)
        `RA_MemEna: rdata = `RWIRES_MemEna;
        `RA_MemAdr: rdata = `RWIRES_MemAdr;
     endcase
   end
endtask
```

Vregs

Specification as Golden Reference

- Typically
  - Architects/HW team write the CSR definitions into a specification
  - HW team writes RTL code implementing CSRs
  - HW team verifies the CSRs
  - SW team uses the CSRs
- Each of these stages often recreate the same CSR information
- Solution:
  - Write specification in standard way
  - Derive RTL defines from specification
  - Derive Verification tests from specification
  - Derive SW defines (C defines and classes) from specification

Vregs

- Our solution: Vregs

- Vregs Flow:
  - Write specification in Word/ Framemaker/ etc.
  - Save document as HTML
  - Run Vregs
    - Vregs writes Verilog header file
    - Vregs writes C++ header file
    - Vregs writes C++ class file
    - Vregs writes Test file
    - … and can write anything else you teach it.

Vregs - Definitions

- Specification Document:

Mnemonic	Constant	Definition
VR_WORD_BITS	4'd32	Bits in a word. (All but 1st paragraph ignored.)
VR_UNINIT_VAL	32'hfeed_face	Default value for VR ram.

- Creates in Vregs_spec_defs.v:

```
`define VR_WORD_BITS   4'd32        // Bits in a word
`define VR_UNINIT_VAL  32'hfeedface // Default value for VR
```

- Creates in Vregs_spec_defs.h:

```
#define VR_WORD_BITS   32          /* Bits in a word */
#define VR_UNINIT_VAL  0xfeedfaceUL /* Default value for VR*/
```

Vregs - Enumerations

- Specification Document:

Enum Ex

Mnemonic	Constant	Definition
INIT	4'b0000	Initial state.
READ	4'b0110	Read Memory State.

- Creates in Vregs_spec_defs.v:

```
`define EX_INIT   4'h0                // Initial State
`define EX_READ   4'hc                // Read Memory State
```

- Creates in Vregs_spec_class.h:

```
// Actually a class is used so the ASCII name of the states
// can be printed out, but effectively:
enum Ex {
        INIT = 0x0,   // Initial State
        READ = 0xc,   // Read Memory State
        MAX  = 0xd};  // Maximum State Value
```

Vregs - Classes (1 of 2)

- Specification Document:

Class IpHdr

Bit	Mnemonic	Type	Constant	Definition
w0[31:28]	VerNum		4	Protocol version number.
w0[15:0]	TotLen	size_t		Total Length.
w1[15]	Flag2			Fragment Flag MSB

- Creates in Vregs_spec_defs.v:

```
`define CE_IpHdr_VerNum   31      // MSB: Protocol Version
`define CB_IpHdr_VerNum   28      // LSB: Protocol Version
`define CE_IpHdr_TotLen   15      // MSB: Total Length
`define CB_IpHdr_TotLen    0      // LSB: Total Length
`define CE_IpHdr_Flag2    47     // MSB: Fragment Flag
`define CB_IpHdr_Flag2    47     // LSB: Fragment Flag
```

Vregs - Classes (2 of 2)

- Creates in Vregs_spec_class.h:

```
class IpHdr {
  const static size_t SIZE = 8;              // Bytes
  nint32_t  m_w[2];                          // Data to Store
  uint32_t  w(int b) { return m_w[b]; }      // Read word
  void      w(int b, uint32_t val) { m_w[b] = val; } //Write
  uint32_t  verNum (void) { return (extract[31:28]); }
  void      verNum (void, uint32_t b) {deposit[31:28]=b; }
  uint32_t  totLen (void) { return (extract[31:15]); }
  void      totLen (void, uint32_t b) {deposit[31:15]=b; }
  bool      flag2 (void) { return (extract[47]); }
  void      flag2 (void, bool b) {deposit[47]=b; }
};
```

Also, additional code for endianess correction, type-casting, and dumping the structure.

Vregs - CSRs (1 of 2)

- Specification Document:

  Register MemCfg

  Address 0x2100_0100

Bit	Mnemonic	Access	Reset	Definition
4	EccError	RW	0	ECC Error Detected.
3:2	Banks	RO	pin	Number of banks. Read from config pins.
0	Enable	RW	0	Set to enable memory controller.

- Creates in Vregs_spec_defs.v:

```
`define RA_MemCfg 32'h21000100 // Address of register
`define CE_MemCfg_Enable    0  // MSB: Set to enable memory
`define CB_MemCfg_Enable    0  // LSB: Set to enable memory
`define CE_MemCfg_Banks     3  // MSB: Number of banks
`define CB_MemCfg_Banks     2  // LSB: Number of banks
`define CE_MemCfg_EccError  4  // MSB: ECC Error Detected
`define CB_MemCfg_EccError  4  // LSB: ECC Error Detected
```

Vregs - CSRs (2 of 2)

- Creates in Vregs_spec_class.h:

```
class R_MemCfg {
public:
  const static size_t SIZE = 4;              // Bytes
  nint32_t  m_w[1];                          // Data to Store
  uint32_t  w(int b) { return m_w[b]; }      // Read word
  void      w(int b, uint32_t val) { m_w[b] = val; } //Write
  bool      enable (void) { return (extract[0]); }
  void      enable (void, bool b) {deposit[0]=b; }
private: // Banks is read-only by applications, so private
  uint32_t  banks (void) { return (extract[3:2]); }
public:
  void      banks (void, bool b) {deposit[3:2]=b; }
  bool      eccError (void) { return (extract[4]); }
  void      eccError (void, bool b) {deposit[4]=b; }
};
```

Vregs - Using a CSR

- Example User's
  C Code to read then write a CSR:

```
#include "vregs_spec_class.h"

void init_memory (void) {
  R_MemCfg  csr;
  //              (address, dataptr, # bytes)
  read_memory   (RA_MemCfg, &csr, csr.SIZE);
  csr.enable(true);   // Sets bit 0, the enable bit
  write_memory  (RA_MemCfg, &csr, csr.SIZE);
}
```

Conclusions

- With the suggested RTL coding techniques, we
  - Improved readability
  - Removed redundancy
  - Reduced maintaince overhead
  - Reduced area
  - Accelerated simulation time

- With Vregs converting the document to headers, we
  - Removed redundancy
  - Reduced the design cycle
  - Have the specification as the primary reference document
    (Which also forces designers to keep it up to date!)

Download Vregs from Veripool.com

- Downloading Vregs:
  - Perl Based
  - Object Oriented with hooks for customization
  - GNU Licensed
  - http://veripool.com/vregs.html

- Also on my site:
  - Dinotrace – Waveform Viewer
  - Gspice – Library Cell Characterization
  - Schedule::Load – Load Balancing (ala LSF)
  - SystemPerl – Simplify and Lint SystemC
  - Verilator – Verilog to SystemC translator
  - Verilog-Pli – Verilog Perl preprocessor and signal renaming
  - Verilog-Mode - /*AUTO…*/ Expansion, Highlighting