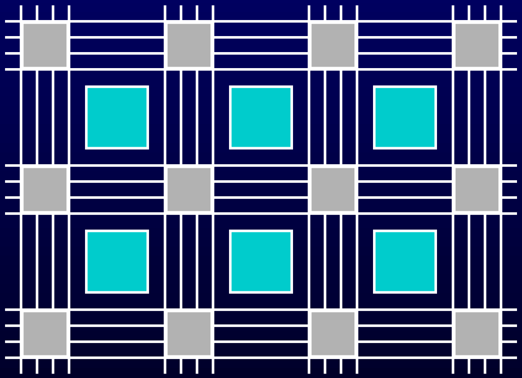
Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip

Jan Gray, Gray Research LLC

jsgray@acm.org

www.fpgacpu.org



```
lw r12,4(r3)
```

About this talk

- Themes
  - FPGA soft CPU cores can be quite compact
  - For best results, design with the FPGA in mind
  - CPU design is not rocket science
- Approach – let's design one
  - Study one implementation *in detail*
  - Highlight FPGA optimizations

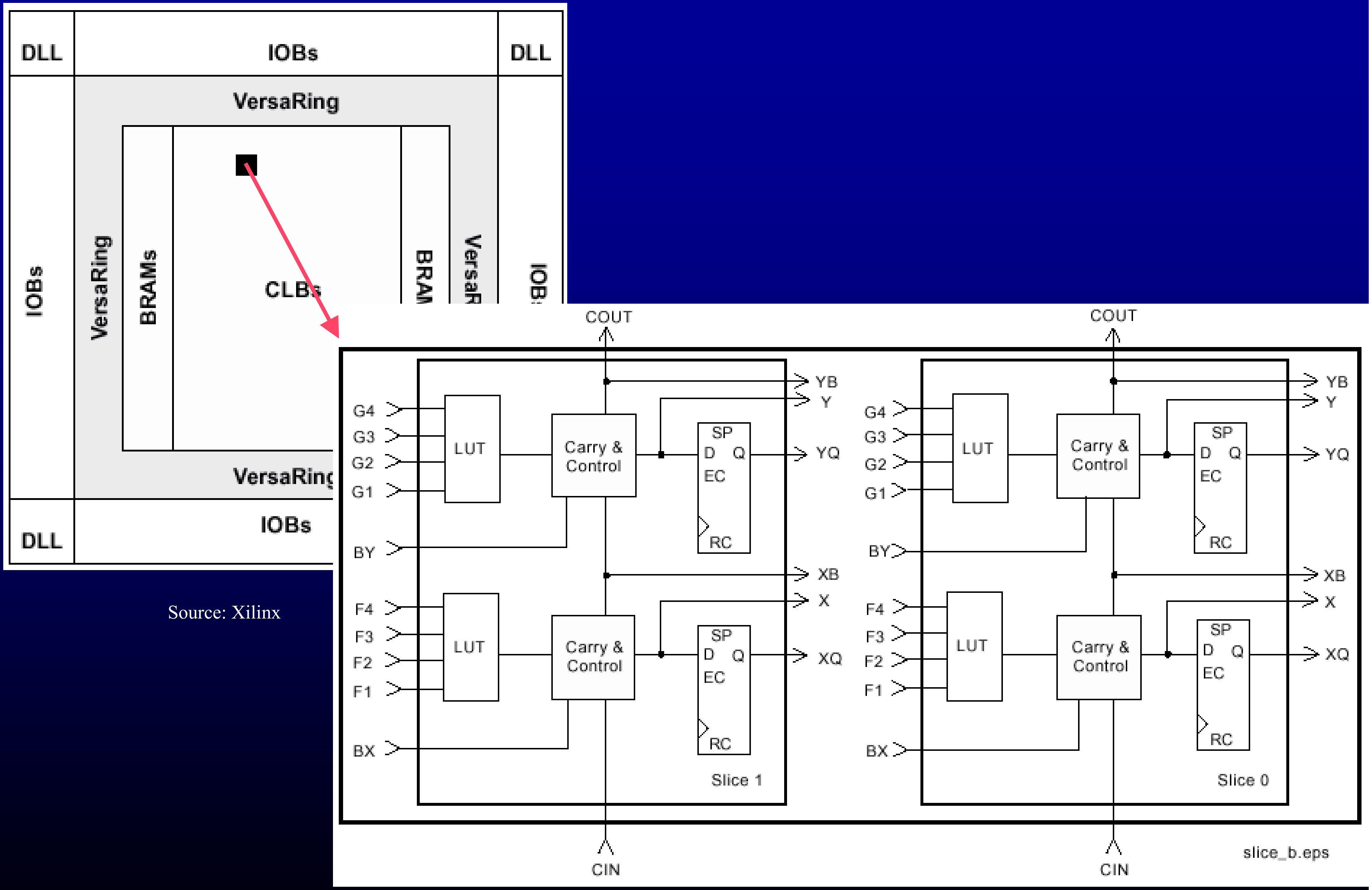
Introduction

- FPGAs: not just glue logic
  - $10-$20 for 100K gates (‘2S100)
  - CPU <10% of chip => cost effective SoC
  - Skip the discrete CPU, skip the ASIC, ship the FPGA
- Advantages
  - Integration, TTM, low NREs, field upgrades
  - Custom instructions, function units, coprocessors
  - Own your IP, control your destiny (end of life)
  - Skip cosimulation?
- Or trade software for complex blocks of logic

Outline

- Introduction
- FPGA Architecture
- Design of CPU and SoC
  - RISC CPU core
  - System-on-a-chip and peripherals
- Results, comparisons
- Software tools
- Conclusions

Xilinx Virtex Architecture



Source: Xilinx

Xilinx XC2S50-5TQ144 FPGA

- 2.5V 144-pin quad flat pack, 92 I/Os
- 16R x 24C array of config logic blocks (CLBs)
  - Each with 2 slices each with 2 logic cells
  - Each cell with a 4-input lookup table, flip-flop
  - Two LUTs = one 16x1 dual port RAM
- 8 block RAMs
  - Dual ported, 4 Kb (256 x 16b), *0 cycle latency*
- Programmable interconnect
  - Hierarchical, plus buses via TBUFs (3-state buffers)

Outline

- Introduction
- FPGA Architecture
- Design of CPU and SoC
  - RISC CPU core
  - System-on-a-chip and peripherals
- Results, comparisons
- Software tools
- Conclusions

Designing a *Simple* CPU and System

- No longer rocket science; thanks to
  - FPGAs: abstraction of a perfect digital world
  - Tools: design implementation; retargetable compilers
- Simple is beautiful
  - Simpler is smaller; smaller is …
    - Cheaper – less area
    - Faster – shorter ‘wires’, easier to fix speed problems
    - Power frugal – less wires to discharge each cycle
  - Simpler is easier to test

Example System: RISC MCU SoC

- GR0040 – a simple CPU core
- GR0041 – GR0040 plus interrupt handling
- SOC – a simple system-on-a-chip
  - GR0041 CPU
  - 1 KB block RAM instruction and data memory
  - Glueless on-chip bus
  - Peripherals: parallel port, counter/timer
- See paper for annotated Verilog source code

GR0040 CPU Core

- An simple RISC core for integer C code
  - One instruction per cycle, non-pipelined
  - Compact => inexpensive
  - 200 lines of Verilog
  - FPGA optimized: ISA, implementation
- 16-bits, 16 registers
  - Scales to 32-bit registers
- Key Ideas
  - Use 0-cycle BRAM for instruction store
  - Use a dual-port LUT RAM bank for register file

GR0040 Instruction Set Architecture

Format	15–12	11–8	7–4	3–0
rr	op	rd	rs	fn
ri	op	rd	fn	imm
rri	op	rd	rs	imm
i12	op	imm12		
br	op	cond	disp8	

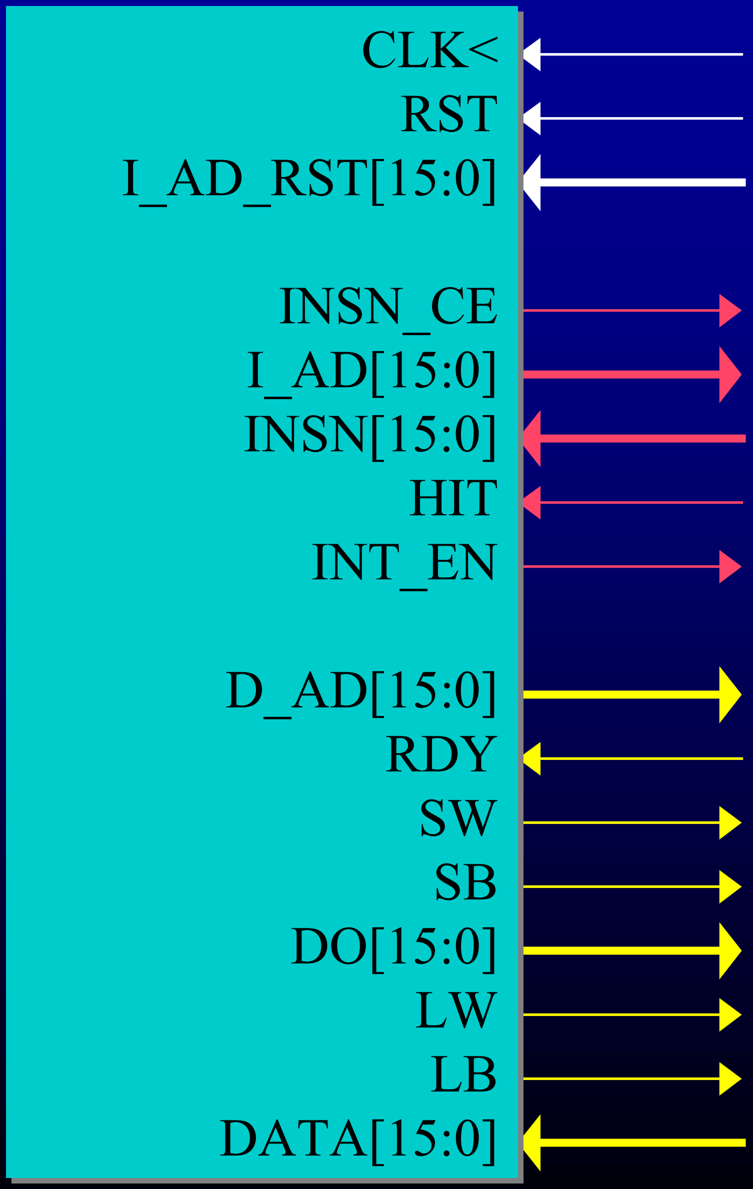
Hex	Fmt	Assembler	Semantics
0*dsi*	rri	`jal rd,imm(rs)`	rd = pc, pc = imm+rs;
1*dsi*	i12	`addi rd,rs,imm`	rd = imm+rs;
2*ds**	rr	`{add sub adc sbc and or xor andn cmp srl sra } rd,rs`	rd = rd *fn* rs;
3*d*I*	ri	`{- rsubi adci rsbci andi ori xori andni rcmpi } rd,imm`	rd = imm *fn* rd;
4*dsi*	rri	`lw rd,imm(rs)`	rd = *(int*)(imm+rs);
5*dsi*	rri	`lb rd,imm(rs)`	rd = *(byte*)(imm+rs);
6*dsi*	rri	`sw rd,imm(rs)`	*(int*)(imm+rs) = rd;
7*dsi*	rri	`sb rd,imm(rs)`	*(byte*)(imm+rs) = rd;
8*iii*	i12	`imm imm12`	imm'next$_{15:4}$ = imm12;
9**dd*	br	`{br brn beq bne bc bnc bv bnv blt bge ble bgt bltu bgeu bleu bgtu} label`	if (*cond*) pc += 2*disp8;

GR0040 Synthesized Instructions

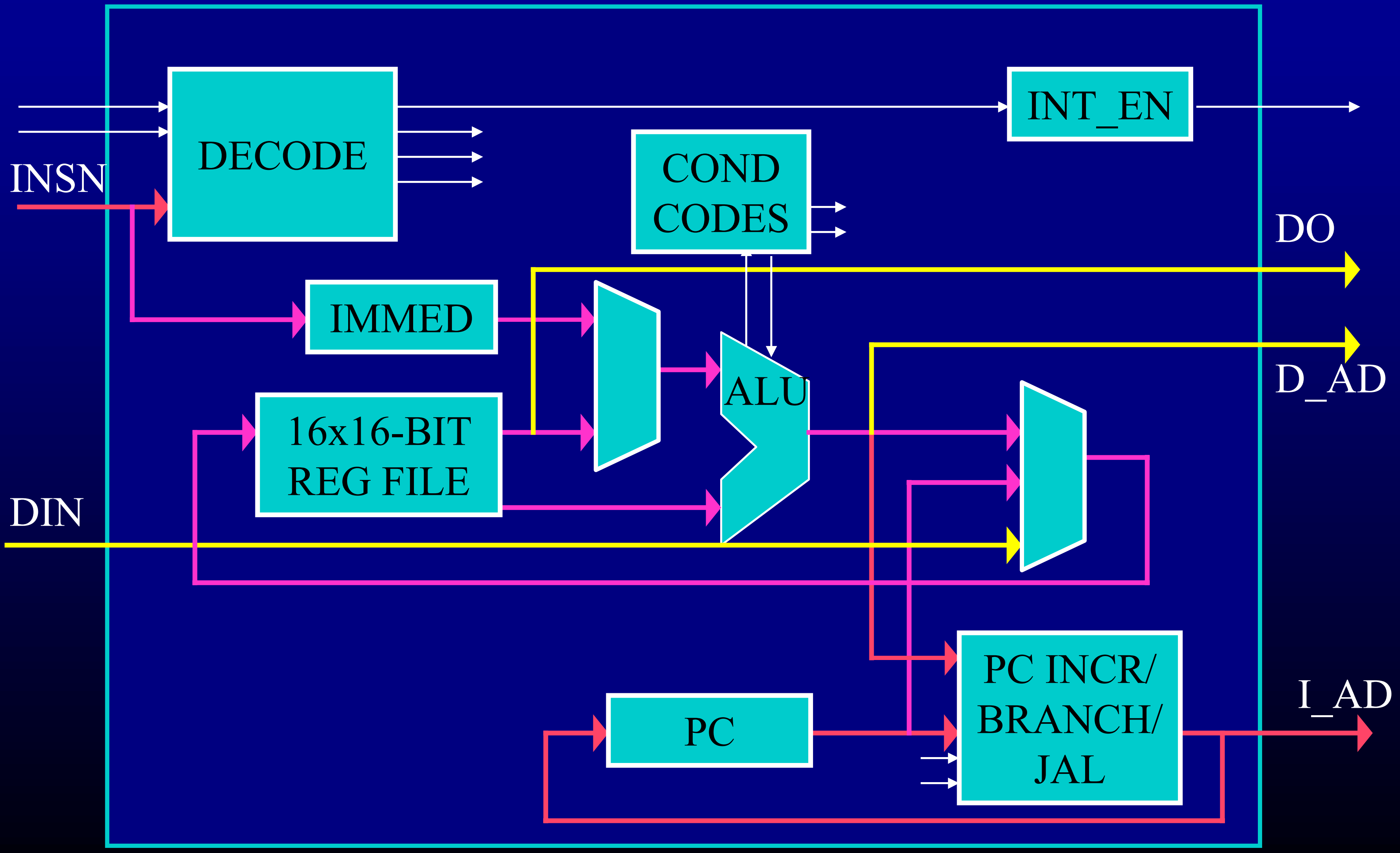
Assembly	**Maps to**
`nop`	`xor   r0,r0`
`mov   rd,rs`	`addi  rd,rs,0`
`subi  rd,rs,imm`	`addi  rd,rs,-imm`
`neg   rd`	`rsubi  rd,0`
`com   rd`	`xori  rd,-1`
`sll   rd`	`add   rd,rd`
`lea   rd,imm(rs)`	`addi  rd,rs,imm`
`j     ea`	`imm   ea`$_{15:4}$ `jal   r1,ea`$_{3:0}$
`call  fn`	`imm   fn`$_{15:4}$ `jal   r15,fn`$_{3:0}$
`ret`	`jal   r1,2(r15)`

`lbs   rd,imm(ra)`
(load-byte,
sign-extending)
	`lb    rd,imm(ra)`
	`lea   r1,0x80`
	`xor   rd,r1`
	`sub   rd,r1`

GR0040 Core Symbol



GR0040 Logical Block Diagram



GR0040 Implementation Outline

- Interface
- Instruction decoding
- Register file and PC
- Immediate literals
- Operand selection
- ALU
  - Adder/subtractor
  - Condition codes
  - Logic unit, shifts
- Result multiplexer
- Jumps and branches
  - Branch decoding
  - Branch logic
  - Next instruction address
- Data load/store controls
- Interrupt enable

GR0040 Interface (1)

- Module

```
module gr0040(
  clk, rst, i_ad_rst,
  insn_ce, i_ad, insn, hit, int_en,
  d_ad, rdy, sw, sb, do, lw, lb, data);

  input  clk;           // clock
  input  rst;           // reset (sync)
  input [`AN:0] i_ad_rst; // reset vector
```

GR0040 Interface (2)

- Instruction Port

```
output insn_ce;      // insn clock enable
output [`AN:0] i_ad;// next insn address
input  [`IN:0] insn;// current insn
input  hit;          // insn is valid
output int_en;       // OK to intr. now
```

- Data Port

```
output [`AN:0] d_ad;// load/store addr
input  rdy;          // memory ready
output sw, sb;       // executing sw (sb)
output [`N:0] do;    // data to store
output lw, lb;       // executing lw (lb)
inout  [`N:0] data; // results, load data
```

Instruction Field Cracking

```
// instruction decoding
wire [3:0] op   = insn[15:12];
wire [3:0] rd   = insn[11:8];
wire [3:0] rs   = insn[7:4];
wire [3:0] fn   = `RI? insn[7:4] : insn[3:0];
wire [1:0] logop= fn[1:0];
wire [3:0] imm  = insn[3:0];
wire [11:0] i12 = insn[11:0];
wire [3:0] cond = insn[11:8];
wire [7:0] disp = insn[7:0];
```

Opcode Decoding

```
 // opcode decoding
`define JAL      (op==0)
`define ADDI     (op==1)
`define RR       (op==2)
`define RI       (op==3)
`define LW       (op==4)
`define LB       (op==5)
`define SW       (op==6)
`define SB       (op==7)
`define IMM      (op==8)
`define Bx       (op==9)
`define ALU      (`RR|`RI)
```

Function Field Decoding

```
 // fn decoding
`define ADD     (fn==0)
`define SUB     (fn==1)
`define ADC     (fn==2)
`define SBC     (fn==3)
`define AND     (fn==4)
`define OR      (fn==5)
`define XOR     (fn==6)
`define ANDN    (fn==7)
`define CMP     (fn==8)
`define SRL     (fn==9)
`define SRA     (fn=='hA)
`define SUM     (`ADD|`SUB|`ADC|`SBC)
`define LOG     (`AND|`OR|`XOR|`ANDN)
`define SR      (`SRL|`SRA)
```

Register File Design

- Key issue
- FPGA RAM primitives
  - Single port or dual port LUT RAM
  - Single/dual port block RAM
- Getting to 2R-1W per cycle
  - Time multiplex access
  - Replicas
- GR0040: dual port LUT RAM
  - Write <u>and read</u> via write-port, read via read-port
  - Perfect fit for `rd = rd` *`op rs`*`;`

Register File and Program Counter

```
// register file and program counter
wire valid_insn_ce = hit & insn_ce;
wire rf_we = valid_insn_ce & ~rst &
        ((`ALU&~`CMP)|`ADDI|`LB|`LW|`JAL);
wire [`N:0] dreg, sreg; // d, s registers

ram16x16d regfile(.clk(clk), .we(rf_we),
  .wr_addr(rd), .addr(`RI ? rd : rs),
  .d(data), .wr_o(dreg), .o(sreg));

reg [`AN:0] pc;      // program counter
```

12-bit Immediate Prefix

- Example

  -
    ```
    imm  0x123
    addi r2,r1,4  ; r2 = r1 + 0x1234
    ```
- Verilog

```
// immediate prefix
reg imm_pre;              // immediate prefix
reg [11:0] i12_pre;       // imm prefix value

always @(posedge clk)
  if (rst)
    imm_pre <= 0;
  else if (valid_insn_ce)
    imm_pre <= `IMM;
always @(posedge clk)
  if (valid_insn_ce)
    i12_pre <= i12;
```

16-bit Immediate Operand

- 4-bits sign-extended, zero-extended, scaled x2, 12-bit prefix || 4-bit immediate

```
// immediate operand
wire word_off = `LW|`SW|`JAL;
wire sxi = (`ADDI|`ALU) & imm[3];
wire [10:0] sxi11 = {11{sxi}};
wire i_4 = sxi | (word_off&imm[0]);
wire i_0 = ~word_off&imm[0];

wire [`N:0] imm16 = imm_pre ? {i12_pre,imm}
                            : {sxi11,i_4,imm[3:1],i_0};
```

Operand Selection

- Examples

```
add  r2,r1      // a=r2 b=r1
addi r4,r3,4    // a=4  b=r3
andi r5,3       // a=3  b=r5
lw   r6,2(sp)   // a=2  b=r14
```

- Verilog

```
// operand selection
wire [`N:0] a = `RR ? dreg : imm16;
wire [`N:0] b = sreg;
```

- Possible <u>tech mapping</u> opt for 50% area savings
  - `o = s1 ? (s2?a:b) : s2;` (one 4-LUT per bit)

ALU: Adder/Subtractor

```
// adder/subtractor
wire [`N:0] sum;
wire add = ~(`ALU&(`SUB|`SBC|`CMP));
reg  c; // carry-in if adc/sbc
wire ci = add ? c : ~c;
wire c_W, x;

// assign {co,sum,x}= add ? {a,ci}+{b,1'b1}
//                        : {a,ci}-{b,1'b1};

addsub adder(.add(add), .ci(ci), .a(a),
        .b(b), .sum(sum), .x(x), .co(c_W));
```

Oops!

- Synplicity missed resource sharing opportunity
  - Got mux, add, sub – 200% larger than necessary
- Fix

```
module addsub(add, ci, a, b, sum, x, co);
  input  add, ci;
  input  [15:0] a, b;
  output [15:0] sum;
  output x, co;
  assign {co,sum,x} = add ? {a,ci}+{b,1'b1}
                          : {a,ci}-{b,1'b1};
endmodule
```

- Moral: check synthesis results!

Condition Codes

```
// condition codes
wire z = sum == 0;                  // zero
wire n = sum[`N];                   // negative
wire co = add ? c_W : ~c_W;         // carry-out
wire v = c_W^sum[`N]^a[`N]^b[`N];// overflow

reg  ccz, ccn, ccc, ccv; // CC register
always @(posedge clk)
  if (rst)
    {ccz,ccn,ccc,ccv} <= 0;
  else if (valid_insn_ce)
    {ccz,ccn,ccc,ccv} <= {z,n,co,v};
```

Add/Sub with Carry

```
// add/subtract-with-carry state
always @(posedge clk)
  if (rst)
    c <= 0;
  else if (valid_insn_ce)
    c <= co & (`ALU&(`ADC|`SBC));
```

ALU: Logic Unit and Shift Right

```
// logic unit
reg [`N:0] log;
always @(a or b or logop)
  case (logop)
  0: log = a & b;
  1: log = a | b;
  2: log = a ^ b;
  3: log = a & ~b;
  endcase

// shift right
wire [`N:0] sr = {(`SRA?b[`N]:0),b[`N:1]};
```

- Delete `or` and `andn`, save >50%

GR0040 Implementation Outline

- Interface
- Instruction decoding
- Register file and PC
- Immediate literals
- Operand selection
- ALU
  - Adder/subtractor
  - Condition codes
  - Logic unit, shifts
- Result multiplexer
- Jumps and branches
  - Branch decoding
  - Branch logic
  - Next instruction address
- Data load/store controls
- Interrupt enable

Result Multiplexer

```
// result mux
wire sum_en = (`ALU&`SUM) | `ADDI;
assign data = sum_en       ? sum : 16'bz;
assign data = (`ALU&`LOG) ? log : 16'bz;
assign data = (`ALU&`SR)  ? sr  : 16'bz;
assign data = `JAL        ? pc  : 16'bz;
```

- FPGA optimization of wide n-input multiplexer
  - Use *free* TBUFs instead of logic muxes
  - Saves tons of logic and interconnect
  - *Impractical on TBUF-poor Virtex-II*

Brief Digression on Customization

- Single cycle – add new function unit

```
wire [`N:0] pop = b[0] + b[1] + … + b[`N];
`define POP (fn == ‘hB)
assign data = (`ALU&`POP) ? pop : 16’bz;
```

- Multicycle – add memory mapped coprocessor

Conditional Branch Decoding

```
// conditional branch decoding
`define BR      0
`define BEQ     2
`define BC      4
`define BV      6
`define BLT     8
`define BLE     'hA
`define BLTU    'hC
`define BLEU    'hE
```

Conditional Branch Decision

```
// conditional branches
reg br, t;
always @(hit or cond or op or
         ccz or ccn or ccc or ccv) begin
  case (cond&4'b1110)
  `BR:   t = 1;
  `BEQ:  t = ccz;
  `BC:   t = ccc;
  `BV:   t = ccv;
  `BLT:  t = ccn^ccv;
  `BLE:  t = (ccn^ccv)|ccz;
  `BLTU: t = ~ccz&~ccc;
  `BLEU: t = ccz|~ccc;
  endcase
  br = hit & `Bx & (cond[0] ? ~t : t);
end
```

Jumps and Branches

- Next PC is one of:

`i_ad_rst`	on reset
`PC`	`~hit` => cache miss / no instruction; try to rerun current instruction
`PC+2`	linear execution or branch not taken
`PC+2*disp`	taken branch
`sum`	`jal` (jump and link)

Jumps, Branches, Instruction Fetch

```
// jumps, branches, instruction fetch
wire [6:0]  sxd7   = {7{disp[7]}};
wire [`N:0] sxd16  = {sxd7,disp,1'b0};
wire [`N:0] pcinc  = br ? sxd16 : {hit,1'b0};

wire [`N:0] pcincd = pc + pcinc;
assign i_ad  = (hit & `JAL) ? sum : pcincd;

always @(posedge clk)
  if (rst)
    pc <= i_ad_rst;
  else if (valid_insn_ce)
    pc <= i_ad;
```

Instruction Fetch, cont'd

- Await `rdy` during valid loads and stores

```
wire   mem      = hit & (`LB|`LW|`SB|`SW);
assign insn_ce = rst | ~(mem & ~rdy);
```

- Another technology mapping optimization

```
    i_ad = jal ? sum : (pc + pcinc);
```

  - 4 inputs per bit => 1 4-LUT per bit
  - Folding in the `?:` mux saves 50% of LUTs
  - (So far) synthesis tools miss these

Load/Store Data

- Examples

```
sw r2,2(r3)   ; mem[2+r3] = r2
lw r4,4(r5)   ; r4 = mem[4+r5]
```

- Verilog

```
// data loads, stores
assign d_ad = sum;
assign do = dreg;

assign lw = hit & `LW;
assign lb = hit & `LB;
assign sw = hit & `SW;
assign sb = hit & `SB;
```

Interrupt Support

- Inhibit interrupts in interlocked sequences
- Verilog

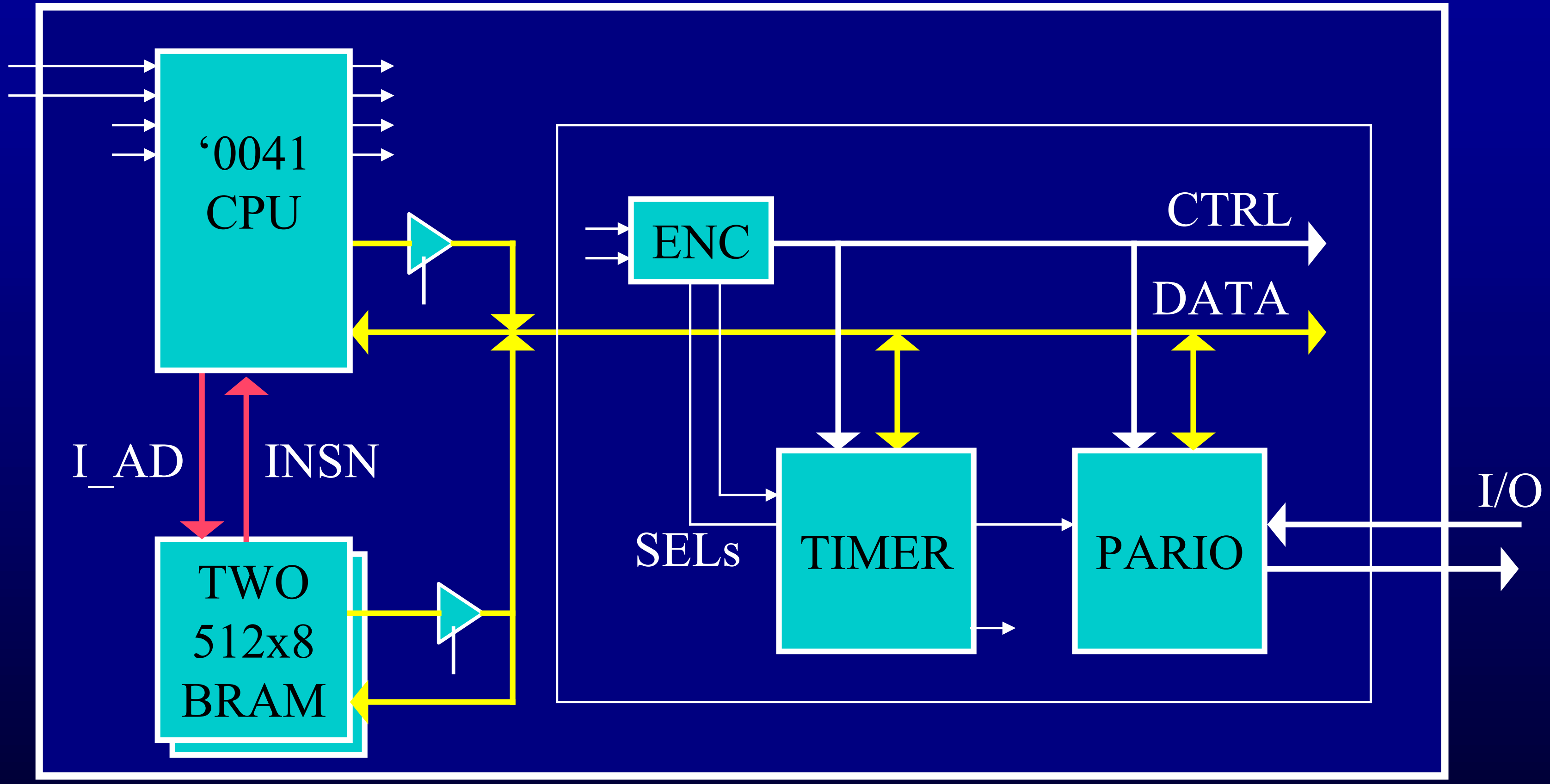
```
  // interrupt support
  assign int_en = hit & ~(`IMM|`ALU&(`ADC|`SBC|`CMP));
endmodule
```

- *Done!*

GR0041 Interruptible CPU Core

- Layer interrupts onto GR0040
- Hold request pending until `int_en` asserted
- Force `insn` to `jal r0,2(r0)`
- Handler at `0002` takes request, then returns to `*r0`
- Just one CLB (or so)
- See paper for details

Simple SoC Logical Block Diagram



- Byte addressable 1 KB code and data storage
- Interrupt timer/counter and parallel I/O

System-On-Chip

```
…
module soc(clk, rst, par_i, par_o);
  input  clk;           // clock
  input  rst;           // reset (sync)

  input  [7:0] par_i; // parallel inputs
  output [7:0] par_o; // parallel outputs
```

Embedded Processor Core

```
// processor ports and control signals
wire [`AN:0] i_ad, d_ad;
wire [`N:0]  insn, do;
tri  [`N:0]  data;
wire int_req, zero_insn;
wire rdy, sw, sb, lw, lb;

gr0041 p(
 .clk(clk), .rst(rst),
 .i_ad_rst(16'h0020), .int_req(int_req),
 .insn_ce(insn_ce), .i_ad(i_ad),
 .insn(insn), .hit(~rst),
 .zero_insn(zero_insn),
 .d_ad(d_ad), .rdy(rdy),
 .sw(sw), .sb(sb), .do(do),
 .lw(lw), .lb(lb), .data(data));
```

Memory Control

- Decoding
  - 32 KB code and data RAM
  - 32 KB memory mapped I/O => 8 periph selects
- Wait states
  - Assert `rdy` unless selected peripheral is not ready
- Byte lane enables

Embedded RAM

- Ideas
  - 16-bits, byte addressable – RMW?
  - Trade off code vs. data storage
  - Each BRAM port can be 256x16, 512x8, …
- 2 BRAMs: `ramh` & `raml`
  - Each with 512x8 instruction and data ports
  - On byte stores, deassert one write enable
  - Byte lane muxes => TBUFs
  - *Boot RAM*
- Expandable

Embedded RAM, cont'd

```
…
// embedded RAM
wire h_we = ~rst&~io_nxt&(sw|sb&~d_ad[0]);
wire l_we = ~rst&~io_nxt&(sw|sb&d_ad[0]);
wire [7:0] do_h = sw ? do[15:8] : do[7:0];
wire [`N:0] di;

RAMB4_S8_S8 ramh(
 .RSTA(zero_insn), .WEA(1'b0),
 .ENA(insn_ce), .CLKA(clk),
 .ADDRA(i_ad[9:1]), .DIA(8'b0), .DOA(insn[15:8]),

 .RSTB(rst), .WEB(h_we), .ENB(1'b1), .CLKB(clk),
 .ADDRB(d_ad[9:1]), .DIB(do_h), .DOB(di[15:8]));

RAMB4_S8_S8 raml(…);
```

Some FPGA On-Chip Buses

- Contenders
  - AMBA: ARM, Altera, LEON-1
    - APB? ASB? AHB?
  - CoreConnect: IBM, Xilinx
  - Wishbone: Silicore, OpenCores.org
- Non-contenders ☺
  - XSOC

XSOC On-Chip Bus

- Simple 16-bit on-chip data bus using TBUFs
- Bus/memory controller
  - Address decoding, bus controls (output enables, clock enables), RAM controls
- Make core reuse easy: *no glue logic required*
  - <u>Abstract control signal bus</u>
  - Encoded in SoC controller
  - Locally decoded within each core
  - Just add core, attach data, control, and select lines
  - Add features without invalidating designs or cores

Control, Select Bus Encoding

```
// control, sel bus encoding
wire [`CN:0] ctrl;
wire [`SELN:0] sel;
ctrl_enc enc(
  .clk(clk),.rst(rst), .io(io), .io_ad(io_ad),
  .lw(lw), .lb(lb), .sw(sw), .sb(sb),
  .ctrl(ctrl), .sel(sel));

wire [`SELN:0] per_rdy;
assign io_rdy  = | (sel & per_rdy);
```

Using Peripherals

```
  timer timer(
    .ctrl(ctrl), .data(data),
    .sel(sel[0]), .rdy(per_rdy[0]),
    .int_req(int_req), .i(1'b1),
    .cnt_init(16'hFFC0));

  pario par(
    .ctrl(ctrl), .data(data),
    .sel(sel[1]), .rdy(per_rdy[1]),
    .i(par_i), .o(par_o));
  …
endmodule // soc
```

8-bit Parallel I/O (1)

```
// 8-bit parallel I/O peripheral
module pario(ctrl, data, sel, rdy, i, o);
  // XSOC boilerplate
  input  [`CN:0] ctrl;
  inout  [`DN:0] data;
  input  sel;
  output rdy;

  // parallel I/O
  input  [7:0] i;
  output [7:0] o;
  reg    [7:0] o;
```

8-bit Parallel I/O (2)

```
  // XSOC boilerplate
  wire clk;
  wire [3:0] oe, we;
  ctrl_dec d(.ctrl(ctrl), .sel(sel),
             .clk(clk), .oe(oe), .we(we));
  assign rdy = sel;

  // parallel port specific
  always @(posedge clk)
    if (we[0])
      o <= data[7:0];
  assign data[7:0] = oe[0] ? i[7:0] : 8'bz;
endmodule
```

16-bit Timer/Counter

- Timer: count when enabled
- Counter: count rising edges when enabled
- Interrupt on count overflow
- Memory mapped control registers
  - CR#0: struct { timer : 1; int_en : 1 };
  - CR#1: struct { irq : 1 };
- See paper for Verilog source

Outline

- Introduction
- FPGA Architecture
- Design of CPU and SoC
  - RISC CPU core
  - System-on-a-chip and peripherals
- Results, comparisons
- Software tools
- Conclusions

Synthesis results

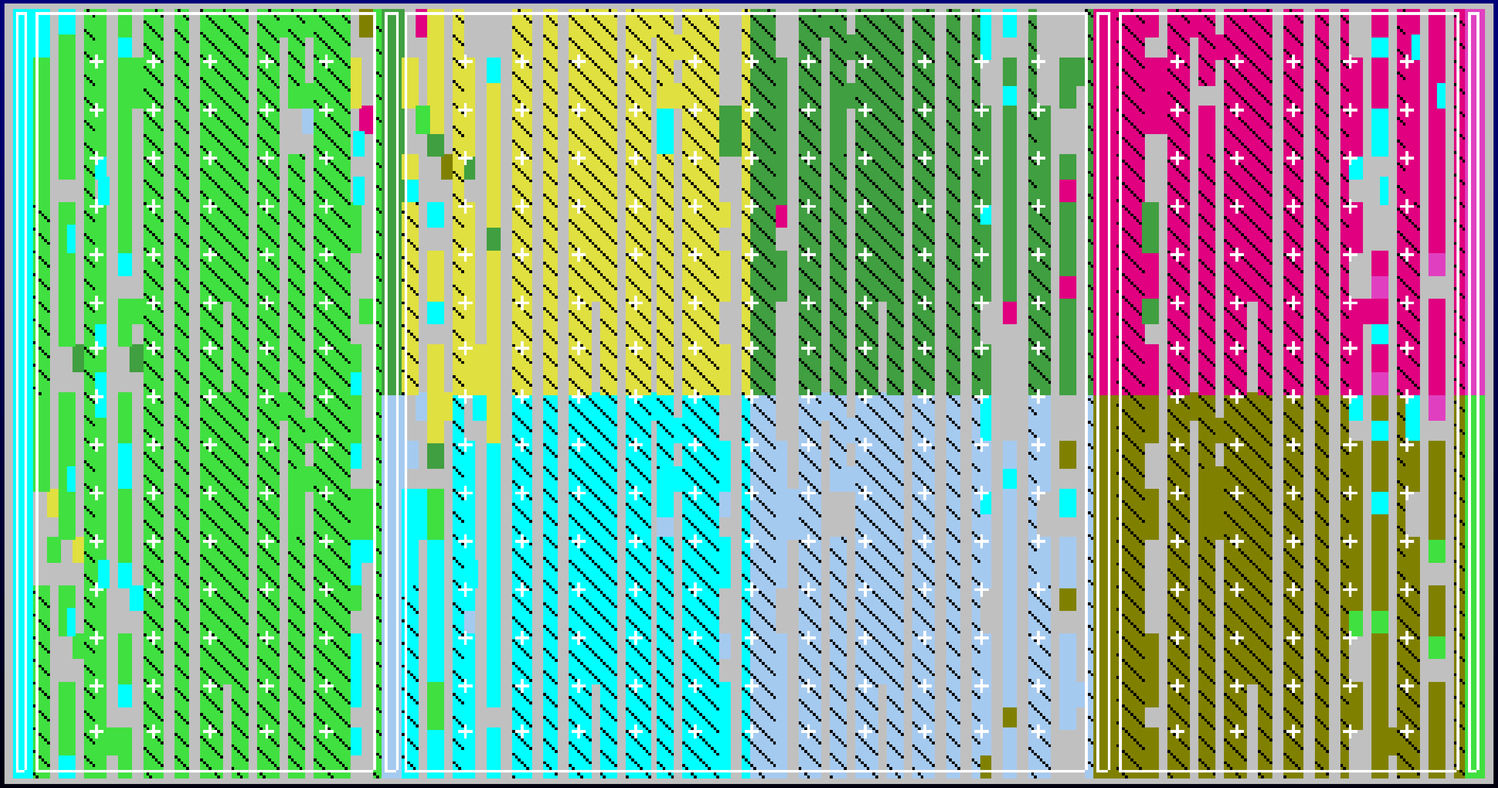
- "Push button"
  - Synplify, Xilinx Alliance 3.1i
  - Complete rebuild in one minute
  - 2 BRAMs, 257 LUTs, 71 FFs, 130 TBUFs
  - 10% of Spartan-II-100, 27 ns cycle time (TRCE)
- Apply tech mapping and floorplanning to source
  - `/*synthesis xc_map="lut"*/`
  - `/*synthesis xc_props="RLOC=R1C0"*/`
  - 2 BRAMS + ~180 LUTs (8x6 CLBs)
  - ~20 ns cycle time

Comparisons

Core	Data width	Logic cells	Freq (MHz)
KCPSM	8	35 CLBs = 140 LCs?	35
gr0040	16	200	50
xr16	16	300	65
Nios	16	1100 LEs	50
gr0050 *hyp*	32	330 *est*	?
Nios	32	1700 LEs	50
ARC basecase	32	1538 slices = 3000+ LCs?	37

Chip Multiprocessors

- 8 gr0040s in one small 16x24 CLB XCV50E
  - Each 8x6 CLBs, 2 BRAMs
  - Each with 1 KB shared local program/data RAM



Achilles Heel: Software Tools

- CPU core is much easier than the compiler
- Retargetable C compilers
  - LCC, *Fraser and Hanson*
  - GCC
- Assemblers, linkers, C libs, debuggers, RTOS …
  - Here GCC shines
  - Nios

Porting LCC

- Download from cs.princeton.edu/software/lcc
  - mips.md ⇒ gr0040.md
  - 32 registers ⇒ 16 registers
  - sizeof(int)==sizeof(void*)==4 ⇒ 2
  - Misc: branches; long ints; software mul/div
- Assembler
  - Lex, parse, fix far branches, apply fixups, emit
  - “Link in the assembler”
- Instruction set simulator
- Alas no C runtime library

LCC Generated Code

-
  ```
  typedef struct TN {
    int k;
    struct TN *left, *right;
  } *T;

  T search(int k, T p) {
    while (p && p->k != k)
      if (p->k < k)
        p = p->right;
      else
        p = p->left;
    return p;
  }
  ```

-
  ```
  _search:               ; r3=k r4=p
      br L3
  L2: lw r9,(r4)
      cmp r9,r3          ; p->k < k?
      bge L5
      lw r4,4(r4)        ; p=p->right
      br L6
  L5: lw r4,2(r4)        ; p=p->left
  L6:
  L3: mov r9,r4
      cmp r9,r0          ; p==0?
      beq L7
      lw r9,(r4)
      cmp r9,r3          ; p->k != k?
      bne L2
  L7: mov r2,r4          ; retval=p
  L1: ret
  ```

Compact Soft CPU Core Applications

- Trade off software for hardware, for shorter TTM
- Interface to outside world: protocol mgmt, UIs
- Custom datapaths and coprocessors
- Absorbing a discrete MCU
- Ephemeral self test

Recap / Conclusions

- It’s easy to build an FPGA SoC
  - KIS and save: CPU <$1 of programmable logic
- Design for the FPGA
  - Know and use the device primitives
    - 4-LUTs, carry, FF clock enables, TBUFs, BRAM ports, …
  - *4-LUTs 4-LUTs 4-LUTs*
  - Review the synthesized netlist
  - Study the static timing analysis report
  - Explicitly map and floorplan key datapaths
- Soft CPUs prominent in the Platform FPGA future

Resources

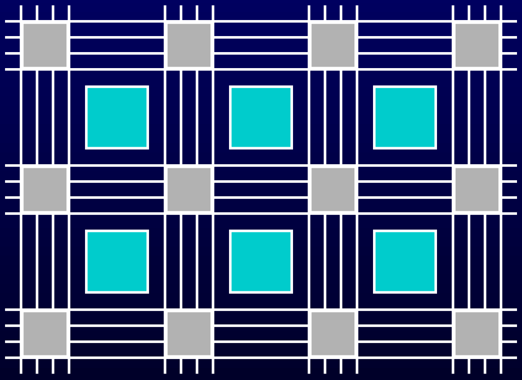
- Online
  - comp.arch.fpga; www.fpgacpu.org; fpga-cpu@egroups.com
- FPGAs
  - www.xilinx.com
  - Trimberger, S., *Field-Programmable Gate Array Technology*.
- LCC
  - C. Fraser and D. Hanson, *A Retargetable C Compiler: Design and Implementation*.
- RISC architecture/implementation
  - D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*

Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip

Jan Gray, Gray Research LLC

jsgray@acm.org

www.fpgacpu.org



```
lw r12,4(r3)
```