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

Jan Gray
Gray Research LLC, P.O. Box 6156, Bellevue, WA, 98008
jsgray@acm.org

Abstract – This paper presents the complete design of a simple FPGA RISC processor core and system-on-a-chip in synthesizable Verilog. It defines a RISC instruction set architecture and then describes how to implement every part of the processor. Next, an interrupt facility is added. The second half of the paper describes the design and implementation of the system-on-a-chip – on-chip RAM, peripheral bus, and peripherals. Throughout, FPGA-specific issues and optimizations are cited. The paper concludes with a comparison with other FPGA processor cores and a brief discussion of software tools issues.

1 Introduction

In the past, field programmable gate arrays (FPGAs) have been used to absorb glue logic, perform signal processing, and even to prototype system-on-chip (SoC) ASICs. Now with the advent of large, fast, cheap FPGAs, such as Xilinx Spartan-II (100,000 1 ns "gates" for $15, quantity one), it is practical and cost-effective to skip the ASIC and ship volume embedded systems in a single FPGA plus off-chip RAM and ROM -- the FPGA implements all of the system logic including a processor core [1-3].

An FPGA SoC platform offers many potential advantages, including high integration, short time to market, low NRE costs, and easy field upgrades of entire systems, even over the Internet. A soft CPU core enables custom instructions and function units, and can be reconfigured to enhance SoC development, debugging, testing, and tuning. And if you control your own "cores" intellectual property (IP), you will be less at the mercy of the production and end-of-life decisions of chip vendors, and can ride programmable logic price and size improvement curves.

Processor and SoC design is not rocket science, and is no longer the exclusive realm of elite designers in large companies. FPGAs are now large and fast enough for many embedded systems, with soft CPU core speeds in the 33-100 MHz range. HDL synthesis tools and FPGA place-and-route tools are now fast and inexpensive, and open source software tools help to bridge the compiler chasm.

To prove the point, this paper and accompanying 50-minute class will present the complete design and implementation, including full synthesizable Verilog, of a streamlined, but real, system-on-a-chip and FPGA RISC processor.

2 Review of FPGA Device Architecture

Our example SoC will target one of the smaller members of the Xilinx Spartan-II family, the XC2S50-5TQ144, a 2.5V FPGA in a 144-pin plastic thin quad flat pack. [4] This SRAM-based device is configured at power-up by an external configuration ROM. It has a 16 row × 24 column array of configurable logic blocks (CLBs), eight dual-ported synchronous 256x16 block RAMs, and 92 I/O blocks (IOBs) in a sea of programmable interconnect. It has other features, like four digital locked loops for clock de-skew and multiplication, that we will not use in the example SoC.

Every CLB has two "slices" and each slice has two 4-input lookup tables (4-LUTs) and two flip-flops. Each 4-LUT can implement any logic function of 4 inputs, or a 16×1-bit synchronous static RAM, or ROM. Each slice also has "carry logic" to help build fast, compact ripple-carry adders and some muxes to help cascade 4-LUTs into larger logic structures.

Each IOB offers input and output buffers and flip-flops. The output buffer can be 3-stated for bidirectional I/O.

The programmable interconnect routes CLB/IOB output signals to other CLB/IOB inputs. It also provides wide-fanout low-skew clock lines, and horizontal *long lines* which can be driven by 3-state buffers (TBUFs) near each CLB.

Spartan-II is a member of the Virtex family of FPGAs. All Virtex devices also provide a number of true dual-ported 256x16 synchronous static RAM blocks. These are very fast, with zero cycle latency – you present address and control signals just ahead of the clock rising edge, and the data are written/read just a few ns afterwards. Our 2S50 device has eight block RAMs, four each on the left and right edges of the device.

Spartan-II almost seems to have been designed with CPUs in mind! Just 16 LUTs can build a single-port 16×16-bit register file (using LUTs as RAM), a 16-bit adder/subtractor (using carry logic), or a 4 function 16-bit logic unit. Since each LUT has a flip-flop, the device is *register rich*, enabling a pipelined implementation style; and as each flip-flop has a dedicated clock enable input, it is easy to stall the pipeline when necessary. Long line buses and 3-state drivers can form efficient n-input word-wide multiplexers or can implement an on-chip 3-state peripheral bus. The block RAMs make excellent instruction and data memories (or caches).

3 Processor Design

Now let's get right down to work and design a simple, FPGA-optimized, 16-bit, 16 register RISC processor core, for hosting embedded applications written in (integer) C, with code-size-efficient 16-bit instructions.

3.1 Instruction set architecture

First we'll choose an instruction set architecture. To simplify the development tools chain, it is tempting to reuse an existing (legacy) ISA, however a new, custom instruction set can be better optimized to minimize the area and hence the cost of an FPGA implementation. In FPGAs, wires (programmable interconnect) and 4-LUTs are the most precious resources, and most legacy ISAs were not designed with that in mind.

Here are the two key ideas behind this new instruction set.

1) Using the zero-cycle latency on-chip block RAM for an instruction store, (either RAM or i-cache), each new instruction is available almost immediately. Therefore, as compared to our earlier CPUs (that sport an instruction fetch pipeline stage to compensate for the latency of off-chip memory), it should be possible to build a simpler, non-pipelined processor with good performance. [5]

2) In a non-pipelined RISC CPU, a two-operand architecture (all register-register operations of the form *dest = dest op src;*) enables the register file to be implemented with a single bank of dual-port distributed RAM.

With these two key decisions made, the rest of the design flows naturally. So here is our streamlined *GR0000* 16-bit RISC instruction set architecture. There are sixteen 16-bit registers, r0-r15, and a 16-bit program counter PC.

There are five instruction formats:

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	

and 22 operation plus 16 branch instructions:

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 and xor adc sbc cmp srl sra } rd,rs`	*[rd =]* rd *fn* rs;
3*d*I*	ri	`{- rsubi adci rsbci andi xori 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} lab`	if (*cond*) pc += 2*disp8;

Some instructions are interlocked and uninterruptible. These include `imm`, `adc*`, `sbc*`, and `*cmp*`. `Imm` establishes the upper 12 bits of the immediate data of the instruction that follows. The carry-out of `adc*/*sbc*` becomes the carry-in of the `add*/sub/adc*/*sbc*` that follows. `*Cmp*` establishes condition codes (not programmer visible) for the conditional branch which follows. These compose, e.g.

```
imm   0xABC
rcmpi rd,0xD
ble   label
```

At first glance, this appears quite austere, but note that many apparently "missing" instructions are easily synthesized.

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`

`or rd,rs`	`mov r1,rd`
	`and r1,rs`
	`xor rd,rs`
	`xor rd,r1`

`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`

Multiply, wide shifts, etc. are done in software.

3.2 Clocking

Our non-pipelined implementation will execute up to one instruction per clock. Assuming the embedded application is running out of on-chip RAM, only loads need be multi-cycle.

3.3 Core interface

The core interface is relatively simple.

```
// Copyright (C) 2000, Gray Research LLC.
// All rights reserved.  Use subject to the
// XSOC License Agreement, see LICENSE file,
// http://www.fpgapcu.org/xsoc/LICENSE.html.
// Version 2000.11.26
```

The 16-bit core is parameterized to make it easier to derive 8- and 32-bit register variants:

```
`define W    16  // register width
`define N    15  // register MSB
`define AN   15  // address MSB
`define IN   15  // instruction MSB

 module gr0040(
```

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

Reset is synchronous, sampled on rising edge of the clock. On reset, the processor jumps to address `i_ad_rst`.

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

The processor core has a Harvard architecture, with separate instruction-fetch and load/store-data ports. Here's the 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
```

As each instruction completes, late in the clock cycle, the core asserts the next instruction address `i_ad`, qualified by `insn_ce`. After `clk` rises, the system drives `insn` with the next instruction word, and asserts `hit` ("cache hit").

If `insn` is not ready, or upon an i-cache miss, `hit` is deasserted, so the processor ignores (annuls) the current, invalid instruction. Therefore, in the implementation that follows, certain decode signals must be qualified by `hit`.

`Int_en` (with `insn_ce`) signals that the currently completing instruction is interruptible, and the system may safely insert an interrupt instruction. Somewhat surprisingly, this signal is all that is necessary to implement interrupt handling in a modular way, entirely outside of the processor core itself.

During a load or store instruction, the core requests a data transfer on the load/store-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
```

The data port outputs `sw`, `sb`, `lw`, and `lb` are valid well ahead of `clk`. The system can sample these and determine whether to assert `rdy` in the current clock cycle.

Memory is byte addressable, and so `d_ad` is the big-endian effective address of the load or store.

During a store instruction, the processor asserts `d_ad` with `do` each cycle until the system signals `rdy` indicating it has captured the store data. `Sw` (store word) data are on `do[15:0]` while `sb` (store byte) data are on `do[7:0]` only.

During a load instruction, the core asserts `d_ad` and awaits `rdy` to indicate that the load data are valid on `data[15:0]`. During `lb` (zero-extending load byte), the system must drive `data[15:8]` with `8'b0`.

Besides loaded data, the tri-state `data` bus is also used within the core to carry all other instruction result values.

3.4 Implementation overview

Here's a brief overview of the CPU implementation that follows. The current instruction `insn` is split into constituent fields and decoded. Two register operands are read from the register file. An immediate operand (if any) is formed. Two operands, `a` and `b` are selected. The ALU, consisting of adder/subtractor, logic unit, and shift right unit, operate upon `a` and `b`. The result multiplexer selects a result to write back to the register file. Any conditional branch is evaluated against the prior instruction's condition code result. The next PC value is determined. During loads/stores, the data port signals are driven and the core awaits `rdy`.

3.5 Instruction decoding

With the current instruction `insn` in-hand, we pull it apart into its constituent fields. Note the `fn` field multiplexer.

```
  // 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)

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

  // 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 [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];
```

3.6 State – register file and program counter

The architected (programmer visible) state consists of the register file and the program counter.

As with most RISCs, register `r0` always reads as zero, but in gr0000, this is not hardwired, but rather due to a software convention. As we shall see, during an interrupt, `r0` is borrowed briefly to capture the interrupt return address.

As noted above, each instruction reads a maximum of two registers (`rd` and `rs`), and writes back one result on one of these ports (`rd`). This enables a compact implementation, using only one 16x16 bank of 16x1-bit dual-port distributed RAMs, consuming 32 4-LUTs. The implementation of the `ram16x16d` module, as a vector of 16x1-bit dual-port RAM primitives, follows in Appendix A.

This RAM is read asynchronously, driving its `wr_o` or `o` outputs when its `wr_addr` or `addr` inputs change. It is written synchronously, storing the input data `d` to address `wr_addr` if `rf_we` is asserted as `clk` rises.

In this case, the `rd` and `rs` fields select registers that are read out onto the `dreg` and `sreg` buses. (We'll consider the peculiar `addr` port `` `RI `` expression momentarily.)

```
  // 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));
```

Later each cycle, the current result, which is available on the `data` bus, is written back into the register file, to the register designated by the `rd` field. This write back occurs only for instructions that produce a result – compute instructions (except `cmp` which discards its result), loads, and `jal`. This is controlled by `rf_we` (register file write enable).

The program counter is a 16-bit register, using 16 flip-flops.

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

3.7 Immediate operand

Many instructions include a 4-bit `imm` immediate operand field, which is sign- or zero-extended to 16-bits. All such immediate instructions can also be preceded by an interlocked immediate prefix `imm`, that establishes the uppermost 12-bits of the immediate operand.

The register `imm_pre` records that there has just been an `imm` prefix instruction, and the 12-bit register `i12_pre` captures the immediate prefix proper.

```
  // 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;
```

The resulting immediate operand is one of:

1) a sign-extended 4-bit immediate (`addi`, all `ri` format);
2) a zero-extended 4-bit immediate byte offset (`lb`, `sb`)
3) a zero-extended scaled-by-two 4-bit immediate word offset (`lw`, `sw`, jal).

   Here the scaling is done by taking `imm[3:0]` and forming the 5-bit value`{imm[0],imm[3:1],1'b0}`. (This helps conserve gates versus a more obvious 2-1 mux of `{1'b0,imm}` and `{imm,1'b0}`.)
4) a 16-bit immediate formed by concatenating `i12_pre` and `imm`.

Therefore the immediate operand is formed as follows:

```
  // 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};
```

3.8 Operand selection

Reviewing the instruction set architecture of section 3.1, we see that with the exception of the `ri` format instructions (`op==3`), all instructions have two operands: a) either an immediate constant or the register selected by `rd` and b) the register selected by `rs`. All `ri` format instructions have two operands: a) an immediate constant and b) the register selected by `rd`. Therefore we obtain operands `a` and `b` as follows.

Operand `a` is the `rd` register for `rr` format instructions and the immediate operand otherwise.

Operand `b` is usually the `rs` register, except for `ri` format instructions, in which case it is `rd`. The preceding `regfile`'s

`.addr()` port assignment "`` `RI ? rd : rs ``" handles this case, so that either register `rd` or `rs` is read out onto `sreg`.

The code is quite simple:

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

Now for a technology mapping optimization. The multiplexer that sources bus `a` selects between `dreg` and `imm16`. The latter is itself a mux of bits of `i12_pre` and `imm`, and the sign-extension constant `sxi`. Do `a` and `imm16 together` require two 16-bit muxes? No.

In 4-LUT FPGA, a 2-1 mux of `o = sel ? x : y` can obviously be implemented in one 4-LUT per bit: o_i `= sel ?` x_i `:` y_i (three inputs). But in the same 4-LUTs, you can also implement a 2-1 mux with a 'force-to-constant' feature: o_i `= mux ? (sel ?` x_i `:` y_i `) : sel`. By encoding `` `RR `` and `sxi` into `mux` and `sel` we can implement these two multiplexers in one LUT per bit.

This is not a transformation our synthesis tools will find, but we will certainly want to apply it in a hand-optimized implementation.

All totaled, operand selection requires only about 20 LUTs.

3.9 Arithmetic/logic unit

Now with the two 16-bit operands `a` and `b` in hand, we can perform arithmetic, logic, and right shift operations upon them. (*Compare* is simply a "subtract-discard-result" that establishes condition codes for the conditional branch instruction which follows.)

The add, subtract, logic unit, and shift "units" all operate concurrently upon the two operands. Then a multiplexer selects one of these values as the result of the instruction.

Now let us build the add/subtract unit. Each Virtex 4-LUT has additional dedicated carry-logic, to implement one 1-bit slice of a ripple carry adder. This code

```
wire [`N:0] sumdiff = add ? a + b : a – b;
```

efficiently implements a 16-bit add, 16-bit subtract, and 16-bit mux in a single column of 16 4-LUTs – each $\texttt{sumdiff}_i$ is a function of the four inputs, `add`, a_i, b_i, and $\texttt{carry}_i$.

But our instruction set places additional demands upon our add/subtract unit, and its implementation is more complex. Consider a 64-bit add: `(r4,r3,r2,r1)+=(r8,r7,r6,r5)`:

```
adc r1,r5  ; carry,r1 = r1 + r5
adc r2,r6  ; carry,r2 = r2 + r6 + carry
adc r3,r7  ; carry,r3 = r3 + r7 + carry
add r4,r8  ;       r4 = r4 + r8 + carry
```

Here we must determine and save carry-out from the `adc` instructions, and inject this carry-in to the `adc` or `add` instruction which follows. Ditto for subtracts with carry.

The most straightforward way to code this is:

```
{co,sumdiff,x} = add ? {a,ci} + {b,1'b1}
                     : {a,ci} – {b,1'b1};
```

Unfortunately our synthesis tool doesn't find the potential resource sharing. Instead of generating a compact 17 LUT adder/subtractor, it synthesizes a 17 LUT adder, a 17 LUT subtractor, and a 17 LUT mux. Lesson: be sure to review the output of your synthesis tool!

After much experimenting, the author found that by moving the above expression out into a separate module `addsub`, the synthesis tool successfully infers the 17 LUT construction. It remains to instantiate it:

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

`Addsub` subtracts when the current instruction is one of `sub`, `sbc`, `cmp`, `rsubi`, `rsbci`, or `rcmpi`.

The ALU is also responsible for determining the (hidden) condition codes (z,n,c,v) – zero, negative, carry, and overflow – for the conditional branch instruction that may follow. The zero condition is self evident, and any two's-complement number is negative if its most-significant bit is set. Carry is the carry-out of the most-significant-bit of the add/sub, complemented for subtracts.

Overflow is tricky. An addition overflows if the two most-significant carry-bits differ, that is,

(overflow) `v = c[W]^c[N];`

We already have the most-significant carry-out, `c_W`, but how do we obtain the next-most-significant carry? Since

`sum[N] = a[N]^b[N]^c[N]`

then

`c[N] = sum[N]^a[N]^b[N]`

therefore

`v = c[W]^sum[N]^a[N]^b[N]`

```
// 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
```

We capture these in the condition code vector `{ccz,ccn,ccc,ccv}` as each instruction completes.

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

If the current instruction is `adc`, `sbc`, `adci`, or `rsbci`, we capture the carry-out in the `c` register.

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

With only two functions to compute (`and` and `xor`), the logic unit is trivial. Since each bit of the result $\texttt{log}_i$ is a function of only three inputs ($\texttt{a}_i$, $\texttt{b}_i$, $\texttt{fn}_0$), this synthesizes into 16 LUTs.

```
// logic unit
wire [`N:0] log = fn[0] ? a^b : a&b;
```

The right shift-unit is even simpler, just a relabeling of some wires, total cost 1 LUT:

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

Here `sra` (shift right arithmetic) propagates the most-significant bit of the `a` operand – a right arithmetic shift of a negative number remains a negative number.

3.10 Result multiplexer

The result multiplexer selects a result from among the various resources `sum`, `log`, `sr`, `pc` (`jal`'s link address), and load data.

The simplest way to implement this is to write explicit `?:` or `if`/`else` multiplexers. However, a more frugal implementation will use the abundant TBUF 3-state drivers provided in the FPGA interconnect fabric. These implement the wide n-input multiplexer function "for free", conserving precious interconnect and using no LUTs whatsoever.

```
// 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;
```

Note that in this implementation, the processor's result bus doubles as the system `data` bus. On loads, the system drives `data` with the 16-bit load result.

Also note it is unnecessary to qualify these decoded instruction output enables with `hit`. (Exercise: why?)

3.11 A digression on extensibility

This use of a long-line tri-state bus for results and load data makes it easy to add new functional units and new memory-mapped coprocessors. For example, adding a "population count" instruction to the processor core requires only decoding the pop-count opcode and driving its result:

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

It is just as simple to add a barrel-shifter, parallel multiplier, or multiply-step-assist instruction, etc. if desired.

To add a multi-cycle functional unit (iterative shifter or multiplier, for instance), it is simplest to attach it as a peripheral with memory mapped control registers. Here's how it such a coprocessor might be used

```
lea r1,mult  ; base of mult control regs
sw  r2,0(r1) ; load multiplier
sw  r3,2(r1) ; load multiplicand, start mul
; do something useful while we're waiting
lw  r2,4(r1) ; load product, interlocked
```

3.12 Another technology mapping optimization

We have already gone to some trouble to ensure that the adder/subtractor is implemented in a single column of LUTs.

However, we can do better than that. Observe that the core of our ALU (excepting the shift-right circuit) computes

```
result = addsub ? (add ? a+b : a-b)
                : (fn[0] ? a&b : a^b);
```

By encoding

```
op = addsub ? add : fn[0];
```

we obtain

```
result = addsub ? (op ? a+b : a-b)
                : (op ? a&b : a^b);
```

Since each $\texttt{result}_i$ depends solely upon its carry-in plus `addsub`, `op`, $\texttt{a}_i$, and $\texttt{b}_i$, it can be expressed in a single column of LUTs.

It goes beyond the scope of this paper, but this optimization saves 16 LUTs (effectively providing the logic functions `and` and `xor` for free) and saves a bank of 16 TBUFs. This is an important savings in TBUF-constrained designs such as multiprocessors.

To take advantage of this optimization, the `or` and `andn` (and-not) instructions were deleted from the architecture. The author was all too happy to trade off a more expensive `or` emulation sequence (shown earlier) for a 10% smaller core.

3.13 Conditional branches

Whether a conditional branch is taken depends upon the current condition code vector {ccz,ccn,ccc,ccv} and the cond field of the branch instruction.

```
// 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 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
```

Here br is set if the current instruction is a valid, taken branch.

3.14 Address/PC unit

The first instruction to execute is at the reset vector address, i_ad_rst.

After that, the next instruction to execute is either the next sequential instruction, the taken branch target, or the target of the jal jump-and-link.

On a taken branch, pc is incremented by the sign-extended 8-bit branch displacement field disp, multiplied by two.

On a jal, the jump target is the effective address sum formed by the adder. Concurrently the current pc is driven onto the result bus and written back into the destination register.

Otherwise, execution continues with the next sequential instruction. If the current instruction is valid (hit is true), pc is incremented by +2. If not, pc is incremented by zero – in effect, re-fetching the current instruction from the instruction store. This is exactly the right behavior on an i-cache miss.

We can implement this efficiently by resource sharing. Instead of muxing pc+2*disp and pc+2, we can mux the pc increment value and add *that* to pc.

```
// jumps, branches, insn 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;
```

The pcinc multiplexer requires 8 LUTs. On its face, it would appear that pcincd, a 16-bit adder, and the i_ad mux, a 16-bit mux, would each require 16 LUTs to implement. Indeed, that's what our synthesis tool does. However, it is possible to implement circuits of the form o = add ? (a + b) : c in a single LUT per bit, 16 LUTs total. Here again we can save significant resources by explicit technology mapping in a subsequent hand-optimized implementation.

On reset, pc is set to i_ad_rst. Thereafter it is clocked as each valid instruction completes.

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

A new instruction is issued each cycle, unless the current valid instruction is a load or store.In that case, the processor awaits the rdy signal from the system.

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

3.15 Loads and stores

To load/store registers from/to memory, we must determine the effective address, initiate a memory transaction, and await is completion.

The instructions lb, lw, sb, and sw, all use the sum produced by the adder as the effective address of the load or store.

To initiate a memory transaction, the processor drives d_ad, do (valid during stores), and the lw, lb, sw, and sb signals.

```
// 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;
```

All very simple. During loads, the system drives the load data onto the result data bus. During lb (zero-extending load byte) in particular, the system must also drive 8'b0 onto data[15:8].

During stores, dreg, a register selected by the rd field, sources the data out bus do.

And as we saw in the previous section, during a load or store instruction, the processor stalls until `rdy` is asserted.

3.16 Interrupt support

As noted earlier, as an experiment in clean, simple, modular system architecture, interrupts are implemented outside of the processor core itself.

Since certain instruction sequences (those commencing with `imm`, `*cmp*`, `adc*`, or `*sbc*`) are interlocked, we cannot tolerate interrupting them mid-sequence. Therefore, the core provides an interrupt enable output to inform the system that it is safe to insert an interrupt:

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

And this concludes the implementation of our RISC core.

```
 endmodule
```

Hardly rocket science, don't you agree?

3.17 Addsub module

Here is that `addsub` module. This synthesizes to an efficient 17 LUT adder/subtractor.

```
 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
```

4 Interrupts

Interrupts are implemented outside of the processor core.

They're quite simple, really. Since an external agent is feeding instructions to the core, on an interrupt request it can "fib" and insert a `call intr` instruction into the instruction stream, and defer the rest of the work to the interrupt handler software. The handler must return in such as away as to run the interrupted instruction.

The following `gr0041` core layers this interrupt facility on top of the `gr0040` processor core. Before we consider the Verilog source, here are some details.

Assume the start of memory looks like this:

```
addr code  disassembly
0000 0000  jal  r0,0(r0) ; iret: return
0002 1EEE  addi sp,sp,-2 ; intr: make room
0004 60E0  sw   r0,0(sp) ; save ret addr
0006 2005  xor  r0,r0    ; zero r0
; insert your interrupt handler code here
0008 40E0  lw   r0,0(sp) ; reload ret addr
000A 1EE2  addi sp,sp,2  ; release room
000C 90FA  br   0000     ; jump to iret
```

The interrupt handler is at address `0x0002`. The `call intr` instruction is `jal r0,2(r0)`, also `0x0002`. This jumps to `2+r0` (which is, by convention, `0`), e.g. `2+0`, saving the return address in `r0` (temporarily violating the convention).

The handler saves the return address on the stack, and then resets `r0` to `0`. Once interrupt processing is complete, the handler reloads `r0` with the interrupt return address, and branches to address `0x0000`. There, the instruction `jal r0,0(r0)` returns to the interrupted code and (since `pc` is `0x0000`) reloads `r0` with `0` once again.

This is somewhat tricky, but has the strengths that it is quite hardware-frugal, and does not waste a general purpose register as a dedicated interrupt return address register. (In the xr16 FPGA processor [6], `r14` is reserved for this purpose.)

Isn't it expensive to mux `jal r0,2(r0)`, e.g. `0x0002`, into the instruction stream? Doesn't it waste gates, and worse, introduce a delay into the critical path? No! The `gr0000` family is optimized for Virtex, and assumes a block RAM instruction store. Virtex block RAMs have a `.RSTx()` port that forces their output register to `0x0000`. By forcing the block RAM to fetch `insn==0x0000`, and then or'ing that with the value `2'b10`, we can insert our `call intr` instruction at a cost of a couple of gates and little delay.

It is not a coincidence that `jal` is assigned opcode `0`!

Here's the code:

```
 module gr0041(
  clk, rst, i_ad_rst, int_req,
  insn_ce, i_ad, insn, hit, zero_insn,
  d_ad, rdy, sw, sb, do, lw, lb, data);
…
  input  int_req;     // interrupt request
  output zero_insn;   // force insn to 0000

  wire int_en;        // interrupt enabled
  reg int;            // call intr in progress

  // interrupt request rising edge detection
  reg int_req_last, int_pend;
  always @(posedge clk)
    if (rst)
      int_req_last <= 0;
    else
      int_req_last <= int_req;
  always @(posedge clk)
    if (rst)
      int_pend <= 0;
    else if (int)
      int_pend <= 0;
```

```
    else if (int_req && ~int_req_last)
      int_pend <= 1;

  // insert intr at an auspicious time
  wire int_nxt = int_pend & int_en & ~int;
  always @(posedge clk)
    if (rst)
      int <= 0;
    else if (insn_ce)
      int <= int_nxt;

  // on int, fetch 0000 and execute 0002,
  // which is 'jal r0,2(r0)' -- call intr
  assign zero_insn = int_nxt;
  wire [`N:0] insn_int = insn | {int, 1'b0};

  gr0040 p(
   .clk(clk), .rst(rst),
   .i_ad_rst(i_ad_rst),
   .insn_ce(insn_ce), .i_ad(i_ad),
   .insn(insn_int), .hit(hit | int),
   .int_en(int_en),
   .d_ad(d_ad), .rdy(rdy),
   .sw(sw), .sb(sb), .do(do),
   .lw(lw), .lb(lb), .data(data));
 endmodule
```

Here the `zero_insn` output will force the next fetched `insn` to `0x0000` on the next `insn_ce`.

5 XSOC System-on-a-chip

The XSOC system-on-a-chip architecture consists of an instantiation of one or more processor cores, interrupt controllers, memory and peripheral controllers, on-chip instruction/data memories/caches, and peripherals.

In this paper, we build a simple demonstration system, featuring 1 KB of on-chip dual-ported byte-addressable, shared program and data RAM, an on-chip bus, and peripherals – a simple counter/timer and a byte-wide parallel I/O port. The timer is configured to interrupt the processor every 64 clock cycles.

The design may be synthesized with or without the I/O, on-chip bus, and peripherals; either way, you still get the RAM.

XSOC features an on-chip peripheral bus architecture with an *abstract* bus control bus. More on that later. For now, note that these definitions configure the width of the on-chip bus controls.

```
// on-chip peripheral bus defines
`define IO       // on-chip periphs enabled
`define CN   31  // ctrl bus MSB
`define CRAN 7   // control reg addr MSB
`define DN   15  // data bus MSB
`define SELN 7   // select bus MSB
```

5.1 System-on-a-chip interface

Like a standalone MCU, this system's off-chip interface is very simple – clock and reset, and 8-bit parallel inputs and outputs. The system uses on-chip RAM for its program and data storage, saving well over 20 package pins.

```
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
```

In practice, the external `rst` input is often replaced by an on-chip reset-on-configuration `startup` block.

5.2 Embedded processor

The first core in our SoC is the processor itself. All of the processor control signals remain on-chip.

Note the processor reset address is configured to be `0x0020`.

```
  //
  // 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));
```

`Hit` is deasserted on reset because the first instruction fetched on reset is not valid.

5.3 Wait state control

The `rdy` line determines when processor load and store instructions complete.

Stores to on-chip RAM complete in the same cycle as they are issued, but loads from on-chip RAM must first wait for the data to be read out on the next `clk` rising edge. In this case, `rdy` is held off until `loaded` goes true in the second cycle of the load.

```
  //
  // rdy (wait state) control
  //
  reg loaded; // load data in bram out regs
  always @(posedge clk)
    if (rst)
      loaded <= 0;
    else if (insn_ce)
      loaded <= 0;
    else
      loaded <= (lw|lb);
```

The signal `io_nxt` is asserted if the current load or store is to a memory-mapped I/O address. In the present design, all addresses `0x8000-0xFFFF` are considered to be I/O accesses.

(If the design is configured without the on-chip I/O bus, `io_nxt` is a constant `0` and most of the subsequent peripheral I/O support is automatically optimized away.)

```
`ifdef IO
  wire io_nxt = d_ad[`AN];
`else
  wire io_nxt = 0;
`endif
```

To keep the processor cycle time from growing without bound as each new peripheral is added to the system, the design holds off all loads and stores to memory mapped I/O locations until the second cycle of the access. A valid I/O access is denoted by the `io` signal.

```
  reg io; // peripheral I/O access underway
  always @(posedge clk)
    if (rst)
      io <= 0;
    else if (insn_ce)
      io <= 0;
    else
      io <= io_nxt;
```

Finally we come to the global `rdy` signal. If the access is to on-chip RAM, the access is ready if it is a store, or if it is a load and the data has already been loaded from RAM (e.g. second cycle of access). If the access is to a peripheral control register, the access is ready when the specific peripheral signals it is ready.

```
  wire io_rdy;
  assign rdy = ~io_nxt&~((lw|lb)&~loaded) |
               io&io_rdy | rst;
```

5.4 Embedded RAM

Recall that this entire design has been optimized for, and made possible by, Virtex dual-port block RAMs. Now we put them to work.

Each block RAM stores 4096 bits, and has two independently configurable ports with configurable access widths. The only restriction on port usage is to not write data on one port while accessing the same data on another.

In this design, we use two Virtex block RAMs, `ramh` and `raml` (RAM high-byte and low-byte), each configured with two independent 512x8-bit ports, for a total of 1 KB of shared RAM. We use the A port on both RAMs as the two byte read-only instruction fetch port, and the B port on both RAMs as the two byte read-write data load/store port. Self-modifying code notwithstanding, no port contention should occur.

(It would also be possible to configure a single block RAM with dual 256x16 ports, one for instructions, one for data, but since Virtex block RAMs lack byte-write-enables, this would complicate byte data stores, requiring read-modify-write cycles. Using two byte-wide RAMs is much simpler.)

Our `gr0041` CPU core assumes that instructions are sourced by high-speed block RAM. Late in each cycle, the next-instruction address `i_ad` is determined, shortly before `clk` rises. On interrupt, the `zero_insn` signal is also asserted. The RAMs' A ports latch this address and shortly thereafter deliver the two selected `insn` bytes (or `0x0000` if `zero_insn` is set).

For loads and stores, the data address `d_ad` is presented to both RAMs' B ports. Since memory is byte-addressable, the design must ensure that byte stores, output from the core on `do[7:0]`, are only written to that byte's block RAM.

On `sw` (store word), `do[15:8]` is written to `ramh` and `do[7:0]` is written to `raml`.

On `sb` (store byte) to an even address, `do[7:0]` is written to `ramh`, since GR0000 is a big-endian architecture. On `sb` to an odd address, `do[7:0]` is written to `raml`.

For loads, `di[15:8]` is loaded from `ramh` and `di[7:0]` from `raml`. For `lw` (load word) specifically, `di[15:0]` drives `data[15:0]`.

For all `lb` (load byte) instructions, whether accessing on-chip RAM or memory-mapped peripheral I/O control registers, `data[15:8]` is driven to `8'b0`, since `lb` is zero-extending.

For `lb` from an even RAM address, `di[15:8]` drives `data[7:0]`. For `lb` from an odd RAM address, `di[7:0]` drives `data[7:0]`.

```
  //
  // 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(
   .RSTA(zero_insn), .WEA(1'b0),
   .ENA(insn_ce), .CLKA(clk),
   .ADDRA(i_ad[9:1]), .DIA(8'b0),
   .DOA(insn[7:0]),
   .RSTB(rst), .WEB(l_we),
   .ENB(1'b1), .CLKB(clk),
   .ADDRB(d_ad[9:1]), .DIB(do[7:0]),
   .DOB(di[7:0]));

  // load data outputs
  wire w_oe = ~io & lw;
  wire l_oe = ~io & (lb&d_ad[0] | lw);
  wire h_oe = ~io & (lb&~d_ad[0]);
  assign data[15:8] = w_oe ? di[15:8] : 8'bz;
  assign data[7:0]  = l_oe ? di[7:0]  : 8'bz;
  assign data[7:0]  = h_oe ? di[15:8] : 8'bz;
  assign data[15:8] = lb   ? 8'b0     : 8'bz;
```

A unique feature of FPGA block RAM (as compared to traditional embedded MCU RAM) is that it is initialized at configuration time – and thus can act as a "boot RAM". Implementation-wise, it is most convenient to specify initial program code and data separately in a separate constraint file.

This embedded block RAM is just the thing to use when your embedded application is modest. By keeping the program on-chip, you save dozens of I/O pads and the CV^2F power they dissipate.

If the embedded system requires more storage for code or data, it is straightforward to use further block RAMs. If the program is larger than available block RAM, it is still possible to use the block RAM as an instruction cache. The instruction tags can themselves be stored in the same block RAM (using the other port), or in a separate block RAM, or even in separate distributed (LUT) RAM.

Similarly, it is possible to use the embedded RAM as a data cache. Here design issues include the write policy (write-back or write-through) and the allocate policy (allocate on write-miss or not). In current devices it is a challenge to implement single-cycle stores with a write-back d-cache because a write to a line may overwrite dirty data from another address that is occupying that line. It may be necessary to first read out the dirty data before overwriting the line or a part of it.

5.5 On-chip peripheral bus architecture

The title of this paper is "Designing a *Simple FPGA-Optimized* RISC CPU and System-on-a-Chip". While there are many strengths to the two emerging industry-standard on-chip buses, AMBA and CoreConnect, they are neither simple nor FPGA-optimized, and you'll probably never live to see their annotated Verilog source code in a paper such as this.

Instead we're going to use a bus that is simple, extensible, and very efficient in its use of programmable logic.

The goals of the on-chip peripheral bus are to enable robust and easy reuse of peripheral cores, and to help prepare for an ecology of interoperable cores to come.

An on-chip bus designer must consider two design communities: core *users* and core *designers*. The former will be more numerous, the latter more experienced. If there are to be ease-of-use tradeoffs, they should be made in favor of the core users.

Since FPGAs are so malleable, and since FPGA SoC design is so new, we need an interface that can evolve to address new requirements, without invalidating existing designs.

With these considerations in mind, I borrowed some ideas from the software engineering world, and defined an *abstract control signal bus*: all of the common control signals are collected into an opaque bus named `ctrl`. In addition, I/O addresses are decoded into the peripheral select vector `sel`, and the ready signals from each peripheral are in `per_rdy`. For simplicity, the processor's result bus `data` is shared as the on-chip peripheral data bus. Now let's see how this on-chip bus architecture is applied.

5.6 Using a peripheral core

The top-level `soc` module establishes `ctrl`, `sel`, `per_rdy`, and `data`.

To add a peripheral core to a design, a designer need only instantiate the core, connect `ctrl`, `data`, some `sel[`*`i`*`]`, some `per_rdy[`*`i`*`]`, perhaps an `int_req`, plus any core-specific inputs and outputs, of course. Address assignment and decoding is implicit with the choice of `sel[`*`i`*`]`.

(A software tool to configure peripherals, addresses, interrupts, and so forth, driven from a system configuration specification file, would be another reasonable approach. However the present design is simpler and adequate for our purposes here.)

Let's contrast this ease-of-use with that of interfacing to a traditional peripheral IC. Each IC has its own idiosyncratic set of control signals, I/O register addresses, chip selects, byte read and write strobes, ready, interrupt request, etc. They don't call it "glue logic" for nothing.

5.7 Implementing a peripheral core

Of course, we can't just sweep all the complexity under the rug. Each peripheral core must decode `ctrl` and recover the specific control signals it needs – clock, reset, byte write enables and output enables and so forth. This is done with the

ctrl_dec module. Each instance of ctrl_dec combines the ctrl bus and a particular sel signal to derive peripheral-specific control signals.

Using ctrl_dec and our on-chip tri-state data bus, the typical bus interface overhead per peripheral is just one or two CLBs and perhaps a column of TBUFs.

Earlier I promised easy extensibility. How does control signal abstraction help? So long as we 'version' soc's ctrl_enc and ctrl_dec together, and retain the same ctrl_dec inputs and outputs, we can make arbitrary changes to the ctrl-encoded bus protocols without invalidating any existing design source code. Neither core uses nor core designs are affected. Just resynthesize and go.

And to add new bus features, with new bus control outputs, we need only provide a new ctrl_dec2 for use by new peripheral core designs.

5.8 Peripheral bus implementation

Returning to the soc module, let's see how all this is done.

```
`ifdef IO
  //
  // on-chip peripheral bus
  //
```

First, for stores to peripherals on the data bus, soc must drive the store data outputs do onto the bus. Just as with the embedded RAM above, byte addressing throws us a curve.

Sw to I/O drives data[15:0] with do[15:0]. However, sb drives both data[15:8] and data[7:0] with the same byte value do[7:0].

```
  // peripheral data bus store data outputs
  wire swsb = sw | sb;
  assign data[7:0]  = swsb ? do[7:0]  : 8'bz;
  assign data[15:8] = sb   ? do[7:0]  : 8'bz;
  assign data[15:8] = sw   ? do[15:8] : 8'bz;
```

Next soc must encode the ctrl and sel buses.

At the start of the second cycle of the access, just as io becomes valid, we capture d_ad in io_ad, the I/O access address. Io_ad is used throughout the peripheral I/O subsystem. Why? After all, io_ad will never differ from d_ad when io is valid. The answer is this is the most straightforward way to keep the I/O subsystem, including peripherals, off the clk-to-clk critical path. This simplifies the lot of the static timing analyzer, which is used heavily during the timing-driven place-and-route phase of the FPGA design implementation. Remember that d_ad is only valid late in the first cycle of a load or store, so adding further decoding and use of d_ad downstream in the I/O subsystem could hurt the minimum cycle time. This way, only the io_ad register setup time could contribute to the critical path.

```
  // control, sel bus encoding
  reg [`AN:0] io_ad;
  always @(posedge clk) io_ad <= d_ad;
```

We'll study the ctrl_enc module next.

```
  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));
```

Io_rdy is simply the or-reduction of each sel line element-wise-gating each per_rdy line. Unassigned per_rdy lines are treated as 0 by my synthesis tool, so valid accesses to selected peripherals should be OK. An invalid access to an unassigned peripheral address might hang, though – a weakness of the current implementation.

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

5.9 Instantiating on-chip peripherals

Now for the two peripherals. The counter/timer is assigned sel[0], e.g. addresses 0x8000-0x80FF and other aliases. Here we elide an explicit interrupt controller, and instead assign the processor's sole int_req input to the timer.

By default, the counter/timer operates in timer mode. It counts every clock cycle that its i input port is set, so here it counts each cycle. Each time it counts up from the initial count 0xFFC0 to overflow at 0x0000, it asserts int_req, and resets the count to 0xFFC0. This has the effect of interrupting the processor every 64 cycles.

```
  //
  // 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));
```

The 8-bit parallel input/output port is assigned sel[1], addresses 0x8100-0x81FF.

Both peripherals assert their rdy signal on the appropriate element of the per_rdy vector.

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

Of course, if the no-IO configuration applies, no peripheral can raise int_req.

```
`else
  assign int_req = 0;
`endif
```

This concludes the soc module.

```
 endmodule
```

5.10 Control bus encoder

The control bus encoder ctrl_enc establishes the abstract ctrl bus and the sel bus. Ctrl and sel each partially decode the I/O access. The two parts ctrl and sel[*i*] are subsequently brought together in each peripheral$_i$'s instance of ctrl_dec.

Ctrl_enc decodes io_addr and establishes I/O bus byte-enables for both output enables oe[] and write enables we[] for up to four byte lanes, anticipating a 32-bit peripheral bus.

```
`ifdef IO

 module ctrl_enc(
  clk, rst, io, io_ad, lw, lb, sw, sb,
  ctrl, sel);

  input  clk;
  input  rst;
  input  io;
  input  [`AN:0] io_ad;
  input  lw, lb, sw, sb;
  output [`CN:0] ctrl;
  output [`SELN:0] sel;
```

Since the on-chip bus can be up to 32-bits wide – four byte "lanes" – a byte- or 16-bit-wide peripheral will reside on a subset of lanes (data[7:0] or data[15:0]).

```
  // on-chip bus abstract control bus
  wire [3:0] oe, we;
  assign oe[0] = io & (lw | lb);
  assign oe[1] = io & lw;
  assign oe[2] = 0;
  assign oe[3] = 0;
  assign we[0] = io & (sw | sb);
  assign we[1] = io & sw;
  assign we[2] = 0;
  assign we[3] = 0;
  assign ctrl={oe,we,io_ad[`CRAN:0],rst,clk };

  assign sel[0] = io & (io_ad[11:8] == 0);
  assign sel[1] = io & (io_ad[11:8] == 1);
  assign sel[2] = io & (io_ad[11:8] == 2);
  assign sel[3] = io & (io_ad[11:8] == 3);
  assign sel[4] = io & (io_ad[11:8] == 4);
  assign sel[5] = io & (io_ad[11:8] == 5);
  assign sel[6] = io & (io_ad[11:8] == 6);
  assign sel[7] = io & (io_ad[11:8] == 7);
 endmodule
```

All very straightforward; note that some signals are validated by io.

5.11 Control bus decoder

Both core users and core designers must treat the specific contents of ctrl as subject to change without notice – because it is. Instead, core users are oblivious to ctrl specifics, and core designers embed an instance of the ctrl_dec control bus decoder into their cores, to obtain the specific control signals they need.

Recall that ctrl and sel each partially decode the I/O access. The two parts ctrl and sel[*i*] are subsequently brought together in each peripheral$_i$'s instance of ctrl_dec to derive its fully decoded I/O byte enables.

```
module ctrl_dec(
 ctrl, sel, clk, rst, oe, we, ad);

 input  [`CN:0] ctrl;// abstract control bus
 input  sel;         // peripheral select
 output clk;         // clock
 output rst;         // reset
 output [3:0] oe;    // byte output enables
 output [3:0] we;    // byte wire enables
 output [`CRAN:0] ad;// ctrl reg addr

 wire [3:0] oe_, we_;
 assign { oe_, we_, ad, rst, clk } = ctrl;
 assign oe[0] = sel & oe_[0];
 assign oe[1] = sel & oe_[1];
 assign oe[2] = sel & oe_[2];
 assign oe[3] = sel & oe_[3];
 assign we[0] = sel & we_[0];
 assign we[1] = sel & we_[1];
 assign we[2] = sel & we_[2];
 assign we[3] = sel & we_[3];
endmodule
```

For maximum flexibility, even the bus clock and reset regime is abstracted into the ctrl bus.

6 Peripherals

This section briefly describes the two peripheral cores used in our example system-on-a-chip.

6.1 8-bit parallel I/O port

This core is trivial. Given ctrl and sel, ctrl_dec provides clock and byte write and output enables. This core uses only we[0] and oe[0], the enables for data[7:0]. Unused

enables are optimized away by the synthesis and/or place-and-route software.

`Pario` never needs to insert wait states. It drives `rdy` with its `sel` input – if it is selected, it is ready (done this cycle).

On a byte write, `pario` latches `data[7:0]` to `o[7:0]`. On a read, it drives `i[7:0]` onto `data[7:0]`.

```
// 8-bit parallel I/O peripheral
module pario(ctrl, data, sel, rdy, i, o);
 input  [`CN:0] ctrl;
 inout  [`DN:0] data;
 input  sel;
 output rdy;
 input  [7:0] i;
 output [7:0] o;
 reg    [7:0] o;

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

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

6.2 16-bit counter/timer

Besides the on-chip bus interface signals, `timer` has two additional inputs, `i` and `cnt_init`.

In timer mode, `i` is a counter clock enable – the counter is incremented as clock rises when `i` is true. In counter mode, `timer` counts `clk` synchronized rising-edge transitions on `i`. `Cnt_init` defines the initial count value, loaded on reset and reloaded on count overflow.

If enabled, `timer` can signal an interrupt request `int_req` when the count overflows. This request remains asserted until the processor writes to the interrupt reset control register.

The two control registers are read-write. Register 0 is the configuration register. Bit 0 enables interrupts on overflow. Bit 1 determines whether `timer` operates as a counter (0) or timer (1).

Register 1 is the interrupt request register. It is set to 1 on counter overflow, and is reset on write.

```
// 16-bit timer/counter peripheral
module timer(
 ctrl, data, sel, rdy, int_req, i, cnt_init);

 input  [`CN:0] ctrl;
 inout  [`DN:0] data;
 input  sel;
 output rdy, int_req;
 input  i;
 input  [15:0] cnt_init;

 wire clk, rst;
 wire [3:0] oe, we;
 wire [`CRAN:0] ad;
 ctrl_dec d(.ctrl(ctrl), .sel(sel),
   .clk(clk), .rst(rst), .oe(oe),
   .we(we), .ad(ad));
 assign rdy = sel;

 // CR#0: counter control register
 // * resets to non-interrupting timer
 // * readable
 // * bit 0: int_en: interrupt enable
 // * bit 1: timer: 1 if timer, 0 if counter
 reg timer, int_en;
 always @(posedge clk)
   if (rst)
     {timer,int_en} <= {2'b11};
   else if (we[0] & ~ad[1])
     {timer,int_en} <= data[1:0];

 // tick counter when:
 // * timer mode:   i (enabled) on clk
 // * counter mode: i rising edge on clk
 reg i_last;
 always @(posedge clk) i_last <= i;
 wire tick = (timer&i | ~timer&i&~i_last);

 // counter/timer
 reg [15:0] cnt;
 wire [15:0] cnt_nxt;
 wire v; // overflow (wrap-around to 0)
 assign {v,cnt_nxt} = cnt + 1;
 always @(posedge clk) begin
   if (rst)
     cnt <= cnt_init;
   else if (tick) begin
     if (v)
       cnt <= cnt_init;
     else
       cnt <= cnt_nxt;
   end
 end

 // CR#1: interrupt request register
 // * resets to no-request
 // * readable
 // * bit 0: interrupt request
 // * cleared on writes to CR#1
 // * set on counter overflow with int_en
 reg int_req;
 always @(posedge clk)
   if (rst)
     int_req <= 0;
   else if (we[0] && ad[1])
     int_req <= 0;
   else if (tick && v && int_en)
     int_req <= 1;
```

```
  // read CR#0 or CR#1
  assign data[1:0]
        = oe[0] ? (ad[1]==0 ? {timer,int_en}
                            : int_req)
                : 2'bz;

 endmodule

`endif
```

And that's all there is to our whole system-on-a-chip.

7 Results

Using Synplicity Synplify, this design synthesizes in a few seconds, and using Xilinx Alliance 3.1i, it is mapped, placed, and routed in less than a minute.

The complete system occupies 2 block RAMs, 257 4-LUTs, 71 flip-flops, and 130 TBUFs. The design consumes just 16% of the logic resources of a Spartan-II-50, one of the smallest devices in that product family. It has a minimum cycle time under 27 ns, disappointingly shy of 40 MHz in a Spartan-II-5 speed grade. Of that 27 ns, 14 ns are spent in logic and 13 ns in routing signals between logic outputs and inputs. That's push-button synthesis for you – quick and easy, but you can often lose some quality of results.

In comparison, the author has also placed-and-routed a highly optimized version of this design sans IO. Certain optimizable structures, cited earlier, are hand-technology-mapped. The datapath is floorplanned using RLOCs to build RPMs (relationally-placed macros), to reduce routing delays. This design yields 50 MHz in the same part and speed grade. It is floorplanned as 8 rows by 6 columns of CLBs, and fits in less than 200 logic cells (less than 50 CLBs).

This figure depicts eight gr0040 cores implemented in a single 16x24 CLB XCV50E, the smallest member of the Xilinx Virtex-E family. (Virtex-E is a Virtex derivative with additional columns of block RAM). Just as with our `soc` above, each of the eight processors in this design has a private 1 KB embedded program/data RAM.

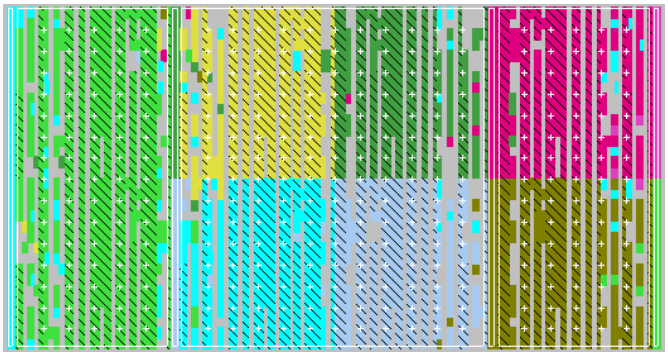

Figure 1: 8 gr0040's in an XCV50E

8 Comparisons

Let's compare the gr0040 core with published data for other FPGA processor cores: the Xilinx KCPSM 8-bit MCU [7], gr0050 (a hypothetical 32-bit stretch of gr0000), the pipelined xr16 RISC [6], the 16- and 32-bit Altera Nios RISC cores [8-9], and the ARC configurable RISC core in its "basecase" configuration (in a 2S150-*6*) [10].

This is certainly an apples-to-oranges comparison, since these cores each support a vastly different set of features, and the instruction sets differ. The only thing they may have in common is they are all (perhaps excepting KCPSM) designed to host integer C code.

For area units, we use logic cells, which correspond to one 4-LUT and one FF. (Xilinx "logic cells" are comparable to Altera "logic elements".) For frequency, where possible, we use the published frequency in the slowest (cheapest) speed grade of the device.

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

Table 1: Approximate core sizes and speeds

Since this is such an apples-to-oranges comparison, there is little one should conclude from this data, except perhaps that there does not appear to be a correlation between core size and clock speed.

9 Software development tools

One of the barriers to entry for would-be custom processor designers is software tools support. Even if it is no longer "rocket science" to design a new processor, maybe it *is* rocket science to obtain even a minimal C compiler tools chain that targets the new instruction set.

Fortunately there are two excellent retargetable free C compilers, GCC [11] and lcc [12].

GCC is the gold standard for embedded system compilers. It is used to build several free OSs and RTOSs. It is accompanied by various C runtime libraries, assemblers, linkers, librarians, and debuggers. Unfortunately it is huge and sprawling and complex.

Lcc on the other hand, is small and simple. It is accompanied by an excellent textbook that describes its inner workings.

The author (formerly a compiler developer) implemented his first lcc custom target in a single day. The big drawback of lcc is it is not GCC – it is incapable of compiling most of the interesting open source software, including GCC's C runtime libraries, because this code tends to make use of either C++ and/or GCC extensions.

For many purposes, lcc is more than adequate.

For this project, the author ported lcc to gr0040. This involved creating a new machine description, derived from the pre-existing xr16 machine description, by changing only 40 lines of code. Here's a typical change:

```
< reg: ADDI2(reg,reg)  "add r%c,r%0,r%1\n" 1
---
> reg: ADDI2(reg,reg) \
       "?mov r%c,r%0\nadd r%c,r%1\n" 1
```

This is an instruction template that describes how to add two arbitrary general purpose registers `r%0` and `r%1` and store the sum in register `r%c`. On xr16, there is a three-operand `add`. On `gr0000`, there is not. Instead, this gr0000 template says to (optionally) move `r%0` to `r%c` (if `r%c` is not `r%0`) and then add `r%1` to `r%c`.

Similarly, the gr0040 assembler and instruction set simulator were derived from the xr assembler/simulator. Here the changes were more extensive, totaling approximately 400 new lines of code.

10 Conclusion

It is possible for mere mortals to build a compact, reasonably fast embedded processor, and even a complete system-on-a-chip, in a small fraction of a small FPGA, if the processor and system are designed to make the best use of the FPGA.

11 Appendix A: ram16x16d module

Here's the 16x16-bit dual-port RAM implementation. It overrides Synplify synthesis, directly instantiating sixteen `RAM16X1D` primitives in an 8 row by 1 column RPM (relationally placed macro). For simulation, it uses a simpler behavioral model.

```
 module ram16x16d(clk, we, wr_ad, ad,d,wr_o,o)
    /* synthesis syn_hier="hard"*/;
  input  clk;          // write clock
  input  we;           // write enable
  input  [3:0] wr_ad;  // write port addr
  input  [3:0] ad;     // read port addr
  input  [15:0] d;     // write data in
  output [15:0] wr_o;  // write port data out
  output [15:0] o;     // read port data out

`ifdef synthesis
  RAM16X1D r0(
    .A0(wr_ad[0]), .A1(wr_ad[1]),
    .A2(wr_ad[2]), .A3(wr_ad[3]),
    .DPRA0(ad[0]), .DPRA1(ad[1]),
    .DPRA2(ad[2]), .DPRA3(ad[3]),
    .D(d[0]), .SPO(wr_o[0]), .DPO(o[0]),
    .WCLK(clk), .WE(we))
      /* synthesis xc_props="RLOC=R7C0.S0" */;
…
  RAM16X1D r15(
    .A0(wr_ad[0]), .A1(wr_ad[1]),
    .A2(wr_ad[2]), .A3(wr_ad[3]),
    .DPRA0(ad[0]), .DPRA1(ad[1]),
    .DPRA2(ad[2]), .DPRA3(ad[3]),
    .D(d[15]), .SPO(wr_o[15]), .DPO(o[15]),
    .WCLK(clk), .WE(we))
      /* synthesis xc_props="RLOC=R0C0.S1" */;

`else /* !synthesis */
  reg [15:0] mem [15:0];

  reg [4:0] i;
  initial begin
        for (i = 0; i < 16; i = i + 1)
      mem[i] = 0;
  end

  always @(posedge clk) begin
    if (we)
      mem[wr_ad] = d;
  end
  assign o    = mem[ad];
  assign wr_o = mem[wr_ad];
`endif
 endmodule
```

12 References

[1] J. Gray, "FPGA CPU News", www.fpgacpu.org/.

[2] J. Gray, "Building a RISC System in an FPGA: Part 1: Tools, Instruction Set, and Datapath; Part 2: Pipeline and Control Unit Design; Part 3: System-on-a-Chip Design", Circuit Cellar Magazine, #116-118, March-May 2000, www.fpgacpu.org/xsoc/cc.html.

[3] J. Gray, "The XSOC Project Kit", March 2000, www.fpgacpu.org/xsoc/.

[4] Xilinx Spartan-II web site, www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Spartan-II

[5] J. Gray, "The Myriad Uses of Block RAM", Oct. 1998, www.fpgacpu.org/usenet/bb.html.

[6] "The xr16 Processor Core", www.fpgacpu.org/xsoc/xr16.html.

[7] K. Chapman, "XAPP213: 8-Bit Microcontroller for Virtex Devices ", Oct. 2000, www.xilinx.com/xapp/xapp213.pdf

[8] "Nios Soft Core Embedded Processor Data Sheet", June 2000, www.altera.com/document/ds/dsexcnios.pdf.

[9] C. Snyder, “FPGA Processor Cores Get Serious”, Microprocessor Report, Sept. 18, 2000.

[10] “ARC 32-Bit Configurable RISC Processor” data sheet, July 2000, www.xilinx.com/products/logicore/alliance/arc/risc_processor.pdf.

[11] “GCC Home Page”, www.gnu.org/software/gcc/.

[12] C. Fraser and D. Hanson, *A Retargetable C Compiler: Design and Implementation*, Benjamin Cummings, 1995. www.cs.princeton.edu/lcc.

13 Legalese

14 Revision history

3/08/01: Added Appendix A detailing the implementation of the 16x16 dual-port RAM register file.

12/07/00: Removed `or` and `andn` from architecture; new technology mapping optimization does adder/subtractor/logic in one column of LUTs.

11/27/00: DesignCon2001 final draft.

10/21/00: Incomplete draft #2.