This Document

This is a snapshot of the zpu/zpu/docs/zpu_arch.html document in CVS.

Several of the links will only work if you have checked out the zpu/zpu tree from opencores CVS. See Download below.

Index

Introduction

The worlds smallest 32 bit CPU with GCC toolchain.

The ZPU is a small CPU in two ways: it takes up very little resources and the architecture itself is small. The latter can be important when learning about CPU architectures and implementing variations of the ZPU where aspects of CPU design is examined. In academia students can learn VHDL, CPU architecture in general and complete exercises in the course of a year.

The current ZPU instruction set and architecture has not changed for the last couple of years and can be considered quite stable. There is a lot of discussion about various modifications to the ZPU architecture in the zylin-zpu mailing list, but currently no actual modifications are planned as the improvements that have been identified are relatively slight(<30% performance/size improvement).

There are a handful of implementations of the ZPU. Most of these usually have some strong points and there is some movement in the direction of consolidating improvements into a few officially recommended ZPU implementations.

For those that are interested in the Zylin ZPU, I recommend joining up on the zylin-zpu mailing list and participating in the discussion there. The zylin-zpu is a friendly place where people of different skills, hardware, software, tools meet to exchange ideas about the ZPU and microprocessor architecture in general.

Sincerely,

Øyvind Harboe
Zylin AS

License

The project includes HDL, GCC toolchain and eCos HAL.

The ZPU has a BSD license for the HDL and GPL for the rest. This allows users to implement any version of the ZPU they want in commercial products, but if improvements are done to the architecture as such, then they need to be contributed back.

Per Jan 1. 2008, Zylin has the Copyright for the ZPU, i.e. Zylin is free to decide that the ZPU shall have a BSD license for HDL + GPL for the rest.

Features

- Small size: (See performance summary)
- Code size 80% of ARM Thumb
- GCC toolchain(GDB, newlib, libstdc+)
- eCos embedded operating system support

Status

- HDL works
- GCC toolchain works
- eCos HAL works

... but there is a long TODO list

Expect churn as we converge onto a shorter list of implementations.

Download source code

The ZPU HDL source code is available as a GIT repository from https://github.com/zylin/zpu.git. You can download the latest sourcecode as a snapshot without installing GIT.

Previously the ZPU repository was hosted as a CVS repository at www.opencores.org, but that ZPU CVS repository is there only for historical reference at this point. Once www.opencores.org grows a GIT hosting service, the plan is to replicate the GIT repository there.

The GCC ZPU toolchain is available from https://github.com/zylin/zpugcc.git. The ZPU GCC toolchain is BIG (over 100 MBytes).

GIT

For more advanced use of GIT, you will need to hit the books and read up on the GIT documentation.

That said, you can ask "silly" newbie questions about GIT on the zylin-zpu mailing list and you should receive some friendly prodding in the right direction w.r.t. finding reading material.

Getting help - mailing list

The place to get help is the zylin-zpu mailing list
To join the list just send an email to zylin-zpu+subscribe@googlegroups.com

The ZPU is an open source project and if you demonstrate that you have made an effort to read the documentation and googled, then you will normally get some help from this list if you ask clear questions.

Architecture

The ZPU is a zero operand, or stack based CPU. The opcodes have a fixed width of 8 bits.

Example:

```
IM 5               ; push 5 onto the stack
LOADSP 20          ; push value at memory location SP+20
ADD                ; pop 2 values on the stack and push the result
```

As can be seen, a lot of information is packed into the 8 bits, e.g. the IM instruction pushes a 7 bit signed integer onto the stack.

The choice of opcodes is intimately tied to the GCC toolchain capabilities.

```
  /* simple program showing some interesting qualities of the ZPU toolchain */
  void bar(int);
  int j;
  void foo(int a, int b, int c)
  {
    a++;
    b+=a;
    j=c;
    bar(b);
  }

foo:
 loadsp 4  ; a is at memory location SP+4
 im 1
 add
 loadsp 12  ; b is now at memory location SP+12
 add
 loadsp 16  ; c is now at memory location SP+16
 im 24    ; ťjť is at absolute memory location 24.
; Notice how the ZPU toolchain is using link-time relaxation
; to squeeze the address into a single no-op
 store
 im 22    ; the fn bar is at address 22
 call
 im 12
 return  ; 12 bytes of arguments + return from fn
```

Instruction set

A base set of instructions must be implemented in RTL, but the rest may be implemented as RTL or as microcode. This allows a tradeoff of core size vs code size and performance.

The instructions that may be implemented in RTL or microcode are referred to as emulated instructions. The microcode is in crt0.s. The implementation determines which instructions run as microcode.

All operations are 32 bit wide.

TODO Is the table broken? Fix it.

Name	Opcode	Description	Definition
BREAKPOINT	00000000	The debugger sets a memory location to this value to set a breakpoint. Once a JTAG-like debugger interface is added, it will be convenient to be able to distinguish between a breakpoint and an illegal(possibly emulated) instruction.	No effect on registers

IM	1xxx xxxx	Pushes 7 bit sign extended integer and sets the a ťinstruction decode interrupt maskť flag(IDIM).
		
		If the IDIM flag is already set, this instruction shifts the value on the stack left by 7 bits and stores the 7 bit immediate value into the lower 7 bits.
		
		Unless an instruction is listed as treating the IDIM flag specially, it should be assumed to clear the IDIM flag.
		
		To push a 14 bit integer onto the stack, use two consecutive IM instructions.
		
		If multiple immediate integers are to be pushed onto the stack, they must be interleaved with another instruction, typically NOP.
			pc <= pc + 1
			idim <= 1
			if (idim=0) then
			sp <= sp - 1;
			for i in wordSize-1 downto 7 loop
			mem(sp)(i) <= opcode(6)
			end loop
			mem(sp)(6 downto 0) <= opcode(6 downto 0)
			else
			mem(sp)(wordSize-1 downto 7) <= mem(sp)(wordSize-8 downto 0)
			mem(sp)(6 downto 0) <= opcode(6 downto 0)
			end if

STORESP	010x xxxx	Pop value off stack and store it in the SP+xxxxx*4 memory location, where xxxxx is a positive integer.	
LOADSP	011x xxxx	Push value of memory location SP+xxxxx*4, where xxxxx is a positive integer, onto stack.	
ADDSP	0001 xxxx	Add value of memory location SP+xxxx*4 to value on top of stack.	
EMULATE	001x xxxx	Push PC to stack and set PC to 0x0+xxxxx*32. This is used to emulate opcodes. See zpupgk.vhd for list of emulate opcode values used. zpu_core.vhd contains reference implementations of these instructions rather than letting the ZPU execute the EMULATE instruction One way to improve performance of the ZPU is to	

		implement some of the EMULATE instructions.	
PUSHPC	emulated	Pushes program counter onto the stack.	
POPPC	0000 0100	Pops address off stack and sets PC	
LOAD	0000 1000	Pops address stored on stack and loads the value of that address onto stack. Bit 0 and 1 of address are always treated as 0(i.e. ignored) by the HDL implementations and C code is guaranteed by the programming model never to use 32 bit LOAD on non-32 bit aligned addresses(i.e. if a program does this, then it has a bug).	
STORE	0000 1100	Pops address, then value from stack and stores the value into the memory location of the address. Bit 0 and 1 of address are always treated as 0	
PUSHSP	0000 0010	Pushes stack pointer.	
POPSP	0000 1101	Pops value off top of stack and sets SP to that value. Used to allocate/deallocate space on stack for variables or when changing threads.	
ADD	0000 0101	Pops two values on stack adds them and pushes the result	
AND	0000 0110	Pops two values off the stack and does a bitwise-and & pushes the result onto the stack	
OR	0000 0111	Pops two integers, does a bitwise or and pushes result	
NOT	0000 1001	Bitwise inverse of value on stack	
FLIP	0000 1010	Reverses the bit order of the value on the stack, i.e. abc->cba, 100->001, 110->011, etc. The raison d'etre for this instruction is mainly to emulate other instructions.	
NOP	0000 1011	No operation, clears IDIM flag as side effect, i.e. used between two consecutive IM instructions to push two values onto the stack.	

PUSHSPADD	61	a=sp;
		b=popIntStack()*4;
		pushIntStack(a+b);	

POPPCREL	57	setPc(popIntStack()+getPc());	

SUB	49	int a=popIntStack();
		int b=popIntStack();
		pushIntStack(b-a);	

XOR	50	pushIntStack(popIntStack() ^ popIntStack());	
LOADB	51	8 bit load instruction. Really only here for compatibility with C programming model. Also it has a big impact on DMIPS test. pushIntStack(cpuReadByte(popIntStack())&0xff);	

STOREB	52	8 bit store instruction. Really only here for compatibility with C programming model. Also it has a big impact on DMIPS test.
		
		addr = popIntStack();
		val = popIntStack();
		cpuWriteByte(addr, val);	

LOADH	34	16 bit load instruction. Really only here for compatibility with C programming model. pushIntStack(cpuReadWord(popIntStack()));	

STOREH	35	16 bit store instruction. Really only here for compatibility with C programming model.
		
		addr = popIntStack();
		val = popIntStack();
		cpuWriteWord(addr, val);	

LESSTHAN	36	Signed comparison
		a = popIntStack();
		b = popIntStack();
		pushIntStack((a < b) ? 1 : 0);	

LESSTHANOREQUAL	37	Signed comparison
		a = popIntStack();
		b = popIntStack();
		pushIntStack((a <= b) ? 1 : 0);	

ULESSTHAN	38	Unsigned comparison
		long a;//long is here 64 bit signed integer
		long b;
		a = ((long) popIntStack()) & INTMASK; // INTMASK is unsigned 0x00000000ffffffff
		b = ((long) popIntStack()) & INTMASK;
		pushIntStack((a < b) ? 1 : 0);	

ULESSTHANOREQUAL	39	Unsigned comparison
		long a;//long is here 64 bit signed integer
		long b;
		a = ((long) popIntStack()) & INTMASK; // INTMASK is unsigned 0x00000000ffffffff
		b = ((long) popIntStack()) & INTMASK;
		pushIntStack((a <= b) ? 1 : 0);	

EQBRANCH	55	int compare;
		int target;
		target = popIntStack() + pc;
		compare = popIntStack();
		if (compare == 0)
		{
		setPc(target);
		} else
		{
		setPc(pc + 1);
		}	

NEQBRANCH	56	int compare;
		int target;
		target = popIntStack() + pc;
		compare = popIntStack();
		if (compare != 0)
		{
		setPc(target);
		} else
		{
		setPc(pc + 1);
		}	

MULT	41	Signed 32 bit multiply pushIntStack(popIntStack() * popIntStack());	

DIV	53	Signed 32 bit integer divide.
		a = popIntStack();
		b = popIntStack();
		if (b == 0)
		{
		// undefined
		} pushIntStack(a / b);	

MOD	54	Signed 32 bit integer modulo.
		a = popIntStack();
		b = popIntStack();
		if (b == 0)
		{
		// undefined
		}
		pushIntStack(a % b);	

LSHIFTRIGHT	42	unsigned shift right.
		long shift;
		long valX;
		int t;
		shift = ((long) popIntStack()) & INTMASK;
		valX = ((long) popIntStack()) & INTMASK;
		t = (int) (valX >> (shift & 0x3f));
		pushIntStack(t);	

ASHIFTLEFT	43	arithmetic(signed) shift left.
		long shift;
		long valX;
		shift = ((long) popIntStack()) & INTMASK;
		valX = ((long) popIntStack()) & INTMASK;
		int t = (int) (valX << (shift & 0x3f));
		pushIntStack(t);	

ASHIFTRIGHT	43	arithmetic(signed) shift left.
		long shift;
		int valX;
		shift = ((long) popIntStack()) & INTMASK;
		valX = popIntStack();
		int t = valX >> (shift & 0x3f);
		pushIntStack(t);	

CALL	45	call procedure.
		
		int address = pop();
		push(pc + 1);
		setPc(address);	

CALLPCREL	63	call procedure pc relative
		
		int address = pop();
		push(pc + 1);
		setPc(address+pc);	

EQ	46	pushIntStack((popIntStack() == popIntStack()) ? 1 : 0);	
NEQ	47	pushIntStack((popIntStack() != popIntStack()) ? 1 : 0);	
NEG	48	pushIntStack(-popIntStack());	

Interrupts

The ZPU supports interrupts.

To trigger an interrupt, the interrupt signal must be asserted. The ZPU does not define any interrupt disabling mechanism, this must be implemented by the interrupt controller and controlled via memory mapped IO.

Interrupts are masked when the IDIM flag is set, i.e. with consecutive IM instructions.

The ZPU has an edge triggered interrupt. As the ZPU notices that the interrupt is asserted, it will execute the interrupt instruction. The interrupt signal must stay asserted until the ZPU acknowledges it.

When the interrupt instruction is executed, the PC will be pushed onto the stack and the PC will be set to the interrupt vector address (0x20).

Note that the GCC compiler requires three registers r0,r1,r2,r3 for some rather uncommon operations. These 32 registers are mapped to memory locations 0x0, 0x4, 0x8, 0xc. The default interrupt vector at address 0x20 will load the value of these memory locations onto the stack, call _zpu_interrupt and restore them.

See zpu/hdl/zpu4/test/interrupt/ for C code and zpu/hdl/example/simzpu_interrupt.do for simulation example.

Custom startup code (aka crt0.s)

To minimize the size of an application, one important trick is to strip down the startup code. The startup code contains microcode for emulation of instructions that may never be used by a particular application, or are made redundant because the instructions are implemented in RTL.

The startup code is found in the GCC source code under gcc/libgloss/zpu, but to make the startup code more available, it has been duplicated into zpu/sw/startup

On the TODO list is work to make it easier to reduce code size.

TODO is the following actually useful? if not remove or elaborate.

To minimize startup size, see codesize demo. This is pretty standard GCC stuff and simple enough once you've been over it a couple of times.

Vectors

Address	Name	Description
0x000	Reset	1.When the ZPU boots, this is the first instruction to be executed. 2.The stack pointer is initialised to maximum RAM address
0x020	Interrupt	This is the entry point for interrupts.
0x040-	Emulated instructions	Emulated opcode 34. Note that opcode 32 and opcode 33 are not normally used to emulate instructions as these memory addresses are already used by boot vector, GCC registers and the interrupt vector.

Core Implementations

zpu4 (superseding zpu3) are original work by Øyvind Harboe. All other implementations derive from zpu4.

High on the TODO list is to reduce the number of implementations taking the best from all. For example interrupts are not universally implemented, IO naming is inconsistent and memory architectures differ.

Ultimately we should try to get closer to the opencores coding standard. You can find the document in the opencores cvsroot/common.

For now if you are starting a design, zpu4 or zealot are probably the safest. zealot offers more customization through generics, but lacks interrupts. zpu4 gets more attention. Take your pick.

Performance Summary

TODO fill in performance table for Altera.

Tests are done with the Zealot SoC-System and Xilinx ISE 12.2 with standard settings. For the MachXO2 device Lattice Diamond 3.1 with Synplify Pro I-2013.09L was used.

CORE/Config	Spartan-3	Spartan-3E	Spartan-6	Virtex-5	MachXO2	DMIPS
zpu4 small maxAddrBit=16	591 LUT 389 REG 0 MULT18x18 16 BRAM 90 fmax	626 LUT 389 REG 0 MULT18x18 16 BRAM 100 fmax	639 LUT 372 REG 0 MULT18x18 16 BRAM 100 fmax	561 LUT 391 REG 0 MULT18x18 8 BRAM (RAMB36) 175 fmax	886 LUT4 459 REG 4 EBR 75 fmax	0.5
zpu4 medium	1760 LUT 514 REG 3 MULT18x18 16 BRAM (RAMB16) 75 fmax	1754 LUT 509 REG 3 MULT18x18 16 BRAM (RAMB16) 75 fmax	1162 LUT 481 REG 3 MULT (DSP48A1) 16 BRAM (RAMB16) 80 fmax	1299 LUT 490 REG 3 MULT (DSP48E) 8 BRAM (RAMB36) 125 fmax	2429 LUT4 755 REG 4 EBR 65 fmax	2.6

zpu4 small

Found in zpu/zpu/hdl/zpu4/core/zpu_core_small.vhd

The small ZPU4 implements the minimum instruction set. It is optimized for size and simplicity serving as a reference in both regards.

It uses a RAM (dual port RAM w/read/write to both ports) as data & code storage and is implemented as a simple state machine.

Essentially it has three states:

1. Fetch - starts fetch of next instruction
2. FetchNext - sets up operands for execute cycle
3. Decode - decodes instruction
4. Execute - well.. executes instruction

The tricky bit is that there is a tiny bit of interleaving of states since the BRAM takes a cycle to perform a fetch/store. The above is the normal states the ZPU cycles through unless memory fetch, jumps, etc. take place.

zpu4 medium

Found in zpu/zpu/hdl/zpu4/core/zpu_core.vhd

The medium ZPU4 has a single port memory interface. All data, code and IO is accessed through this memory interface.

It performs better(despite having less memory bandwidth than zpu_core_small.vhd) since it implements many more instructions.

Alvaro's pipelined ZPU

All the rave in the mailing list. TBA.

Zealot

Small found in zpu/zpu/hdl/zealot/zpu_small.vhdl

Medium found in zpu/zpu/hdl/zealot/zpu_medium.vhdl

README found in zpu/zpu/hdl/zealot/0README.txt

The Zealot version of ZPU was contributed by Salvador E. Tropea.

The key features are:

- Includes a very basic PHI I/O synthesizable core. It implements the 64 bits clocks counter (timer), GPIO and the UART. This is enough to run the DMIPS benchmark and a hello world application. I tested the UART @ 9600 bps and @ 115200 bps.
- The ZPU can be customized using generics. It allows the use of more than one core in the same project without problems.
- Implements the lshiftright instruction in hardware, this gives around 10% boost in the DMIPS benchmark (Medium version).
- You can disable various instructions groups and let them to the emulation soft, so you can experiment with various LUTs vs DMIPS configurations (Medium version).
- The medium version provides aprox. 2.6 DMIPS @ 50 MHz and the small 0.5 DMIPS @ 50 MHz.
- Enhanced trace module, it includes the assembler for the executed instruction and can also measure how much stack was consumed during the execution.
- Includes ready to use memory images for a hello world program and the DMIPS benchmark.
- Memory and trace blocks outside ZPU. This provides better modularity.
- Much better documented code than the original version.

Simulation and implementation files are provided. You need 16 kB of BRAMs for the "hello world" example and 32 kB for the DMIPS benchmark. The medium version takes around 1030 slices and 3 multipliers and the small version around 430 slices.

The generics for the Zealot Medium ZPU are:

- **WORD_SIZE** (integer:=32) Data width, only 32 bits are really tested/supported. Adding support for 16 bits should be simple, but the toolchain needs to support it.
- **ADDR_W** (integer:=16) Address bus width memory+I/O space. The MSB selects the address space (1=I/O).
- **MEM_W** (integer:=15) Memory address bus width. It includes program, data and stack sections.
- **D_CARE_VAL** (std_logic:='X') Value used to fill the unsused bits. For simulations this should be '0', for synthesis this is a value that your tools interprets as "don't care". Xilinx tools could get benefit from using 'X'. This is particularly true to assign default values and for unreached cases. Note that I didn't find it useful.
- **MULT_PIPE** (boolean:=false) Enables the multiplication pipeline. This can allow faster clocks but will make the mult instruction slower (more clocks consumed).
- **BINOP_PIPE** (integer range 0 to 2:=0) Enables the pipeline for the -, =, < and <= operations. This can allow faster clocks but will make these instruction slower (more clocks consumed). This value is the amount of extra clocks added.
- **ENA_LEVEL0** (boolean:=true) Enables the hardware implementation of eq, neqbranch, loadb and pushspadd instructions.
- **ENA_LEVEL1** (boolean:=true) Enables the hardware implementation of lessthan, ulessthan, mult, storeb, callpcrel and sub instructions.
- **ENA_LEVEL2** (boolean:=false) Enables the hardware implementation of lessthanorequal, ulessthanorequal, call and poppcrel instructions.
- **ENA_LSHR** (boolean:=true) Enables the hardware implementation of lshiftright instruction.
- **ENA_IDLE** (boolean:=false) Enables the enable_i usage. This signal can hold the CPU in an idle state if after reset this signal remains active. When disabled the enable_i signal isn't used and the idle state is removed.
- **FAST_FETCH** (boolean:=true) This version of the ZPU fetches 4 instructions at ones (32 bits), then they are decoded (2 cycles) and finally executed. The decoded instructions are stored in a "decode cache", the first instruction is immediately moved to the "current instruction" register and a "special instruction" replaces the first slot. This "special instruction" makes the CPU go to the fetch state. When you enable this generic the FSM does the fetch instead of waiting one clock cycle to go to the fetch state. This makes instructions run a little bit faster, but it can cost area and/or frequency.

ZY2000

Found in zpu/zpu/hdl/zy2000/zpu_core.vhd Modified version of zpu4 medium for use with a wishbone bridge.

The ZY2000 is a complete implementation including: ZPU, DRAM, soft-MAC, wishbone bridges, GPIO subsystem, etc. This also included an eCos HAL w/TCP/IP support.

Verilog translation

Found in zpu/wip/ZPU_CORE/src/zpu_core.v

The verilog version of ZPU (zpu4) was contributed by Jurij Kostasenko. No-one appears to be maintaining it, but it should be a useful starting point for further work. There are some useful scripts there.

Implementing your own ZPU

One of the neat things about the ZPU is that the instruction set and architecture is very small and it is easy to implement a ZPU from scratch or modify the existing ZPU implementations.

Implementing a ZPU can be done without understanding the toolchain in detail, i.e. using exclusively HDL skills and only a rudimentary understanding of standard GCC/GDB usage is sufficient.

A few tips:

- Run zpu_core.vhd or zpu_core_small.vhd and generate an instruction trace from ModelSim or similar. To check that you own implementation is correctly implemented, verify that the instruction trace for the new and old ZPU implementations match. This gives you a simple way to do regression tests as you develop your ZPU.
- To improve performance, you can add more instructions. The EMULATE instructions are optional in HDL since they will be emulated in software if they are not implemented in HDL. This allows you to run the ZPU executables unmodified regardless of which EMULATE instructions you implement.
- Run the DMIPS test to measure your overall performance
- Run the histogram.perl script on the instruction trace to generate histograms of the instructions. Profiling is essential to making the right choices w.r.t. optimization for your application.

Reference Designs

The zpu core is independent of IO and memory architecture. Here are three levels of reference designs a user can refer to in order to get started in their own design, regardless of chosen core.

TODO converge on a single IO structure for core implementations.

TODO re-org CVS to make it easy to keep appropriate SW, RTL(verilog and VHDL) , scripts, verification stuff together.

Minimal (core+RAM)

The minimum design is a zpu core with true dual port RAMs attached. This is handy for size/fmax trial in a particular FPGA, and maybe HDL regression. Maybe not a very useful starting point, unless you can DMA all you IO.

TODO provide FPGA scripts.

TODO provide HDL regression environment.

Basic (core+RAM+UART+Timer)

The minimum design required for hello_world and DMIPS applications. Requires more RAM and a UART (or something) for stdio. This is handy as a starting point for a new users design, and to run DMIPS evaluation, and maybe HDL regression.

TODO provide FPGA scripts.

TODO provide HDL regression environment.

SOC (core+RAM+Wishbone+++)

Large design(s) for one or more chosen eval board. Features dictated by board and available IP.

Common - RAM models

single (1RW), simple dual(1R+1W), true dual(1RW+1RW), and xilinx distributed dual(1RW+1R) RAM models. Parameterized depth / width, and loadable from file. The goal is that ROM be independent of verilog/VHDL implementation of RAM.

TODO RAM model contribution needed. What is in opencore/common is not adequate.

Common - Wishbone

In hdl/wishbone there is an implementation of a wishbone bridge. It was designed to work with ZY2000

TODO make wishbone bridge re-usable with all cores

Common - UART

All self respecting embedded projects should have a debug channel to print stuff to. Typically this is a standard RS232 or UART, but it can also be something more exotic like a DCC JTAG channel.

The point is that characters(bytes) are sent to/from the ZPU via some terminal.

The ZPU defines in the memory map a UART / debug channel. This should be implemented by some suitable debug channel for the device in which the ZPU is implemented.

www.opencores.org has several UART implementations. This is one of the simpler ones: http://www.opencores.org/projects.cgi/web/uart/overview

Implementing your own UART / debug channel

The first thing you need to do is to choose a debug channel for your hardware. This could be a UART, but it doesn't have to be.

Secondly you should write a small HDL module that interface between the ZPU memory map of debug channel to the UART. This should be relatively simple as all you need to do is to let the ZPU query the FIFO in/out for busy flag and allow the ZPU to read/write data to the UART via the memory map.

TODO explicit example with UART from opencores in the above ref designs.

SPI flash controller (read-only)

This is a simple read-only SPI flash controller, with the following characteristics:

- Fast-READ only implementation.
- 32-bit only access

- Fast sequential read access - Uses low-clock approach

Version

The current version is 1.2. This is also the first public version available.

Timing overview

Simple timing overview, with one nonsequential access to address 0x0, followed by a sequential access to address 0x4. This simulation was done with Xilinx tools, after post-routing, and using a ZPU to access the SPI

Image 1: Timing overview

On Image 2, you can see the clock almost perfectly centered on data, when we write to the SPI flash.


Image 2: Issuing commands to the SPI

As you can see from Image 3, I assume the worst-case read delay from SPI (which is 15ns, as you can see from the marker).


Image 3: Reading from the SPI

Usage

Simple description of SPI controller interface:

Symbol	Direction	Bit width	Purpose
adr	Input	24	Address where to read from SPI
dat_o	Output	32	Data read from SPI
clk	Input	1	Input clock. Used for both interface and SPI
ce	Input	1	Chip Enable
rst	Input	1	Asynchronous reset
ack	Output	1	Data valid ACK
SPI_CLK	Output	1	SPI output clock
SPI_MOSI	Output	1	SPI output data from controller to chip
SPI_MISO	Input	1	SPI input data from chip to controller
SPI_SELN	Output	1	SPI nSEL (deselect, active low) signal

License

The Verilog implementation is released under BSD license. See the file itself for more licensing details.

Dowload

Download the Verilog code here: spi_controller.v

Troubleshooting

The current implementation is timed and optimized for myself. Your parameters might not be the same as those I defaulted, so read the code carefully. If you have any issue let me know.

Working with the tools and core

TODO discussion of tools needed and choose some to be supported by project. Need to deal with cygwin vs linux, VHDL vs verilog, open vs closed.... plus language support in simulators is sometimes lacking.

Xilinx ISE webpack is available for windows and linux
Altera Quartus web edition is windows only.
Lattice ispLEVER starter edition is windows only.

None appear to come with a standalone simulator anymore. Not sure if any built in simulators are worth looking at... never have been in the past.

Popular Simulation tools for this kind of project: Modelsim, GHDL, veriwell, cver, icarus, gtkwave... others?

Setup - Linux toolchain

You will need Java installed to run the simulator and some other stuff.

TODO setup.sh script needs to detect linux/cygwin, and should have install path option.

```
$ cd zpu/zpu/sw     # path as appropriate
$ sh setup.sh       # untars the tool chain to ... TODO
$ . env.sh          # puts the tools in you path
```

Setup - Cygwin toolchain

Install Cygwin You will need Java installed to run the simulator and some other stuff.

```
$ cd zpu/zpu/sw     # path as appropriate
$ sh setup.sh       # unzips the tool chain to /tmp/zpu/install/bin
$ . env.sh          # puts the tools in you path
```

GCC to RAM

TODO some of this is generic, some is zpu4 specific. Should move to refdesign section when ref designs exist.

The instructions are stored big endian. That is the first instruction is stored in the most significant byte, and the forth is in the least significant byte.

Generating VHDL BRAM initialization

```
$ zpu-elf-objcopy -O binary hello.elf hello.bin
$ java -classpath ../simulator/zpusim.jar com.zylin.zpu.simulator.tools.MakeRam hello.bin >hello.bram
```

Build another test application for example simulation

Here is how to build a rom image for an application using the zpu/example simulation files.

```
$ cd zpu/roadshow/roadshow/dhrystone
$ sh build.sh
$ cd zpu/hdl/example
$ gcc zpuromgen.c
$ ./a
Usage: ./a binary_file
$ ./a ../../roadshow/roadshow/dhrystone/dhrystone.bin >app.txt
```

Copy and paste app.txt into helloworld.vhd.

TODO need to merge following with above.

The ZPU comes with a standard GCC toolchain and an instruction set simulator. This allows compiling, running & debugging simple test programs. The Simulator has some very basic peripherals defined: counter, timer interrupt and a debug output port.

Hello world example

The ZPU toolchain comes with newlib & libstdc++ support which means that many C/C++ programs can be compiled without modification.

```
$ cd zpu/sw/helloworld
$ zpu-elf-gcc -Os -phi hello.c -o hello.elf -Wl,--relax -Wl,--gc-sections
or ? TODO which one
$ zpu-elf-gcc -phi hello.c -o hello.elf
$ zpu-elf-size hello.elf
```

HDL simulation (ZPU4)

TODO some of this is generic, some is zpu4 specific. Should move to refdesign section when ref design exists.

For new users you will also find scripts in the zealot area that may be useful.

You'll find a working simulation script in hdl/example/simzpu_small.do and hdl/example_medium/simzpu_medium.do, which show simulation of the small(zpu_core_small.vhd) and medium sized ZPU(zpu_core.vhd). hdl/example/simzpu_interrupt.do shows use of interrupts.

When implementing the ZPU, copy the following files and modify them to your needs:

1. hdl/example/zpu_config.vhd - set up RAM size here
2. hdl/example/helloworld.vhd - dual port BRAM implementation.

Obviously you must also connect the ZPU to the rest of your IO subsystem. IO is memory mapped(read/write) in the ZPU.

Running example simulation

The hdl/example directory has a simulation written for Xilinx WebPack ModelSim. From the ModelSim command prompt:

1. cd c:/<installfolder>/hdl/example
2. do zpusim_small.do

After running the hello world simulation (see zpusim.do), two files are written to the hdl/example directory:

1. log.txt - contains the "Hello world!" text written to the debug channel/simplified UART.
2. trace.txt - a trace file for the CPU. The instruction set simulator has the capability of taking this file as input in order to verify that the HDL implementation matches the instruction set simulator. When a mismatch is found, the GDB debugger will break. Very handy for debugging custom ZPU implementations.

GDB simulation

1. cd zpu/sw/helloworld
2. Launch the simulator from a seperate bash shell:

   java -classpath ../simulator/zpusim.jar -Xmx512m com.zylin.zpu.simulator.Phi 4444

3. Launch GDB:

   ../install/bin/zpu-elf-gdb hello.elf

4. Connect to target, load and run application:

   ```
   (gdb) target remote localhost:4444
   (gdb) load
   (gdb) continue
   ```

Simulator

The ZPU simulator is integrated into the Zylin Embedded CDT plugin to ease debugging of ZPU applications:

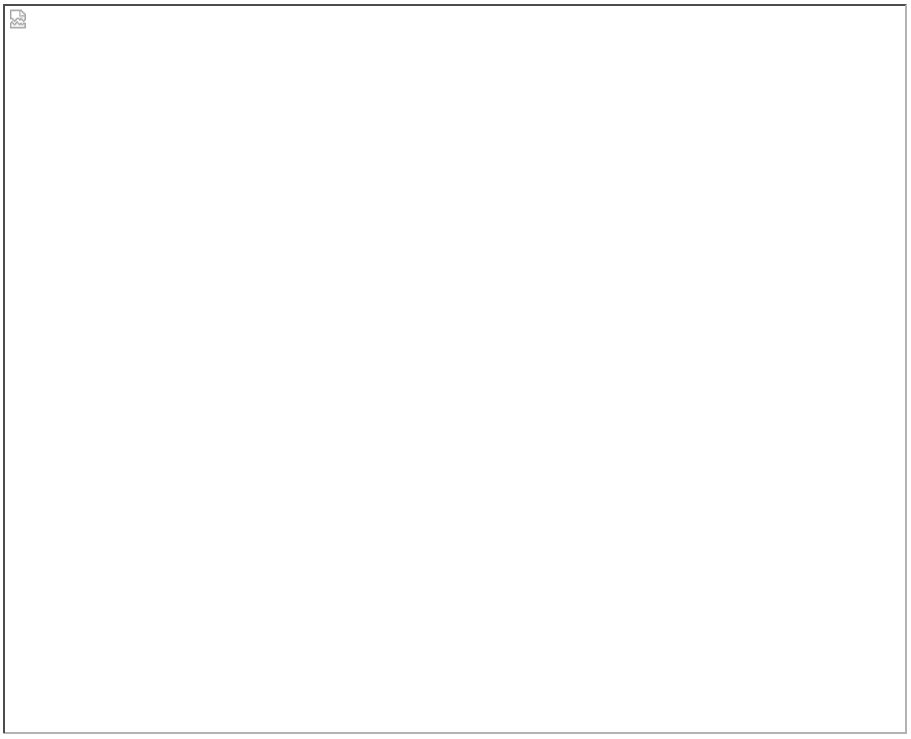
http://www.zylin.com/embeddedcdt.html

The ZPU simulator has many features besides debugging an application:

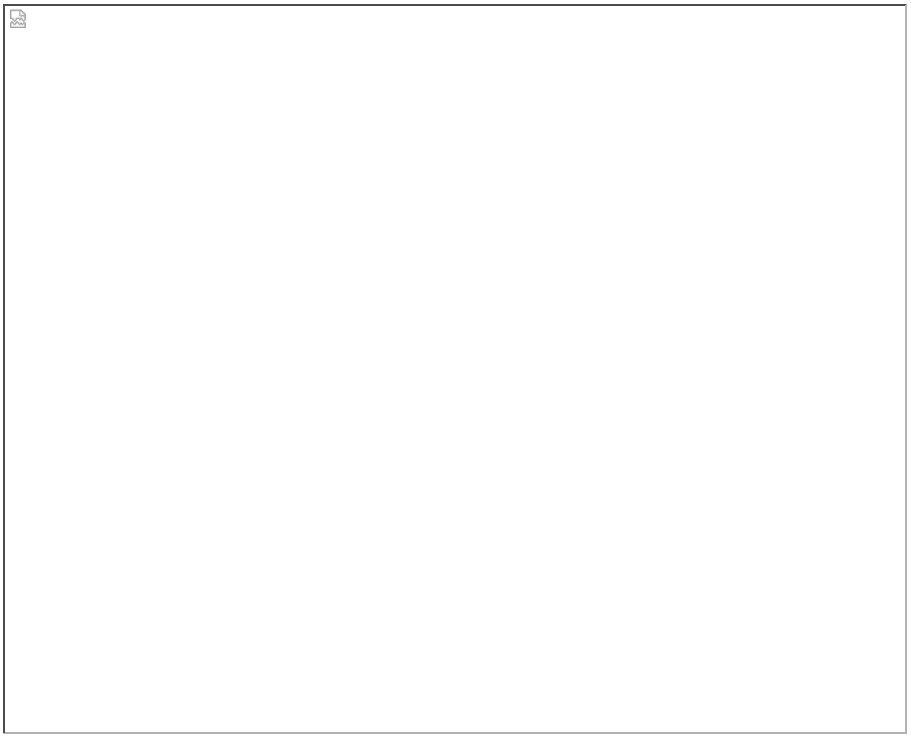
- taking output from simulation(e.g. ModelSim) and matching that against the Java simulator, thus making it much easier to debug HDL implementations and also getting real world timing information
- can generate gprof output
- generate various statistics

The plugin is still pretty rough around the edges, and needs to get GUI support for enabling the ModelSim trace input feature.

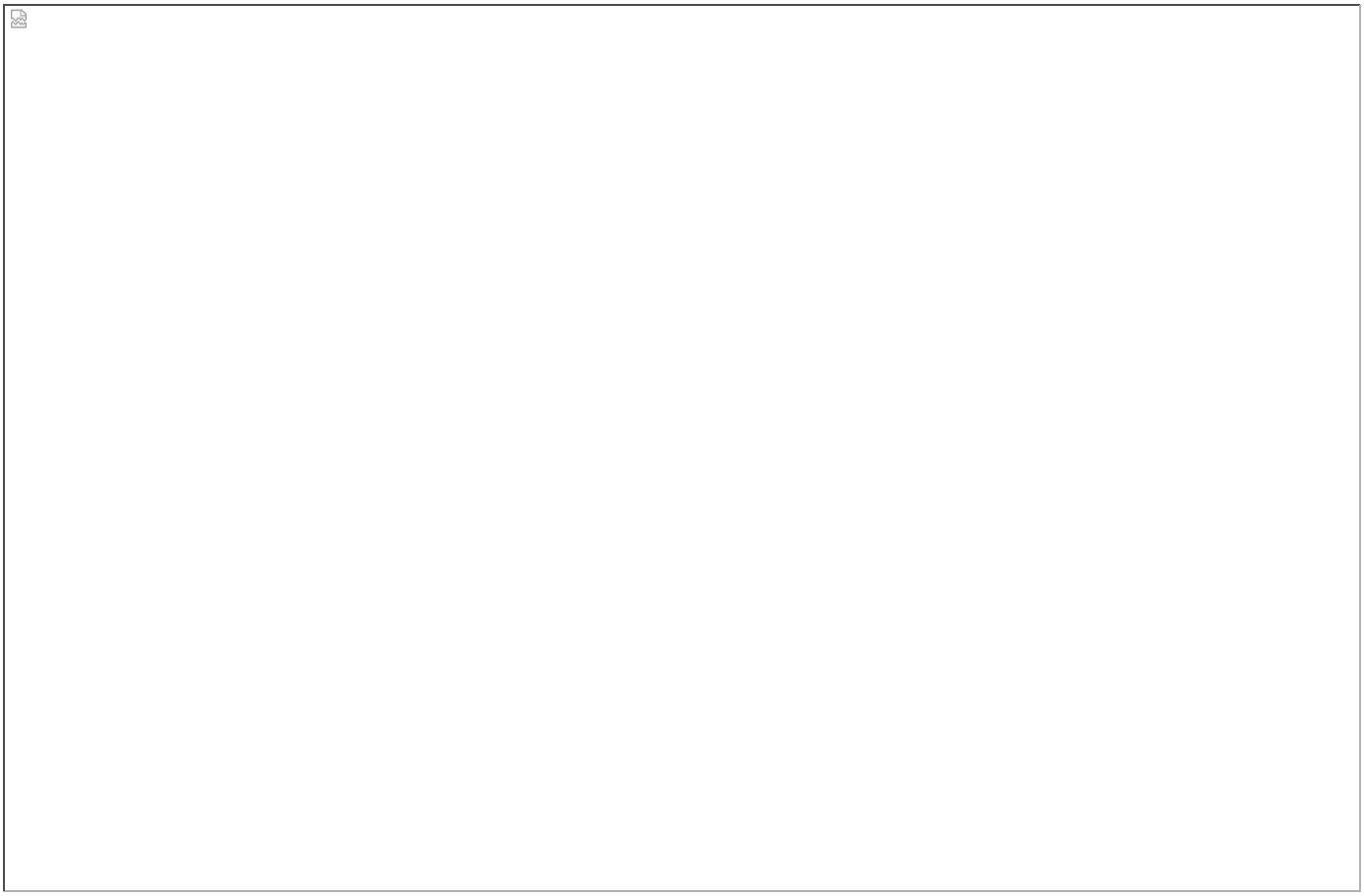
Compiling ZPU application



Setting up the simulator



Choosing ZPU executable



Debug session

Misc

TODO Stuff that could probably find a better home.

Speeding up the ZPU

There are two aspects of speeding up the ZPU: making it perform better for a particular application and toying around with the ZPU architecture.

Performance tips

1. Profile. Create a small sample and run in a simulator that is as close to the real deployment as possible. zpu4/core/histogram.perl is a script that will tell you which instructions take the most time.
2. Using the profile output, decide on which emulated instructions that it makes sense to implement in HDL for your particular application. Modifying zpu_core_small.vhd is not particularly hard. Most instructions can be transliterated into zpu_core_small.vhd from zpu_core.vhd without too much problem.
3. The memory subsystem may well turn out to be where you should concentrate your efforts.

Toying around with the architecture

Again: profile 90% of the time and spend the remaining 10% tinkering with the architecture.

- There is a DMIPS program you can use to measure the performance of the ZPU in lieu of profiling a real application. The latter is obviously a superior solution.
- Again: use histogram.perl to figure out which instructions you should add in HDL.
- Tinker a bit with Fmax to find the maximum speed rating for your design.
- zpu_core_small.vhd should be ca. 1 DMIPS and zpu_core.vhd should yield about 5-10 DMIPS before adding instructions runs out of steam.

If you need to get ca. 20-50 DMIPS out of the ZPU you will have to write a heavily pipelined architecture with caches(if you are running against DRAM). This is *tricky*, but some proof of concept work was done to show 20 DMIPS w/the ZPU(the actual result was discarded since it was not complete and contained fatal flaws).

Achieving above 50-100 DMIPS with the current ZPU architecture is probably a non-starter and a more conventional RISC design makes more sense here.

The unique advantages of the ZPU is size in terms of HDL & code size.

Optimizing for code size

The ZPU toolchain produces highly compact code.

1. Since the ZPU GCC toolchain supports standard ANSI C, it is easy to stumble across functionality that takes up a lot of space. E.g. the standard printf() function is a beast. Some compilers drop e.g. floating point support from the printf() function and thus boast a "smaller" printf() when in fact they have a non-standard printf(). newlib has a standard printf() function and an alternative iprintf() function that works only on integers.
2. The ZPU ships with default startup code that works across various configurations of the ZPU, so be warned that there is some overhead that will not occur in the final application(anywhere between 1-4kBytes).
3. Compilation and linker options matter. The ZPU benefits greatly from the "-Wl,--relax -Wl,--gc-sections" options which is not used by all architectures(e.g. GCC ARM does not implement/need -Wl,--relax).

Small code example

```
zpu-elf-gcc -Os -abel smallstd.c -o smallstd.elf -Wl,--relax -Wl,--gc-sections
zpu-elf-size small.elf

$ zpu-elf-size small.elf
text data bss dec hex filename
2845 952 36 3833 ef9 small.elf
```

Even smaller code example

If the ZPU implements the optional instructions, the RAM overhead can be reduced significantly.

```
zpu-elf-gcc -Os -abel crt0_phi.S small.c -o small.elf -Wl,--relax -Wl,--gc-sections -nostdlib
zpu-elf-size small.elf

$ zpu-elf-size small.elf
text data bss dec hex filename
56 8 0 64 40 small.elf
```

Installing eCos build tools

```
tar -xjvf ecossnapshot.tar.bz2
tar -xjvf repository.tar.bz2
tar -xjvf ecostools.tar.bz2
# run this every time you open the shell
export PATH=$PATH:`pwd`/ecos-install
export ECOS_REPOSITORY=`pwd`/ecos/packages:`pwd`/repository
```

Compiling eCos tests

```
ecosconfig new phi default
ecosconfig tree
make
cd kernel/current
make tests
```

Code size ZPU

```
$ zpu-elf-size *
   text    data     bss     dec     hex filename
  15761    1504   12060   29325    728d bin_sem0
  16907    1512   14436   32855    8057 bin_sem1
  17105    1524   30032   48661    be15 bin_sem2
  17186    1512   14436   33134    816e bin_sem3
  18986    1500   12036   32522    7f0a clock0
  15812    1504   13236   30552    7758 clock1
  25095    1972   13224   40291    9d63 clockcnv
  16437    1500   13224   31161    79b9 clocktruth
  15762    1504   12060   29326    728e cnt_sem0
  17124    1512   14436   33072    8130 cnt_sem1
  35947    1564   22512   60023    ea77 dhrystone
  16428    1500   13228   31156    79b4 except1
  15751    1504   12052   29307    727b flag0
  19145    1512   15624   36281    8db9 flag1
  20053    1516  102908  124477   1e63d fptest
  15998    1496   12092   29586    7392 intr0
  16080    1496   12200   29776    7450 kalarm0
  15327    1496   12036   28859    70bb kcache1
  15549    1496   13224   30269    763d kcache2
  18291    1500   12260   32051    7d33 kclock0
  16231    1500   13232   30963    78f3 kclock1
  16572    1496   13228   31296    7a40 kexcept1
  15618    1496   12060   29174    71f6 kflag0
  19287    1500   15624   36411    8e3b kflag1
  16887    1516   15628   34031    84ef kill
  16186    1496   12128   29810    7472 kintr0
  19724    1504   14516   35744    8ba0 klock
  18283    1500   14592   34375    8647 kmbox1
  15539    1496   12064   29099    71ab kmutex0
  16524    1504   15664   33692    839c kmutex1
  18272    1712   20348   40332    9d8c kmutex3
  18682    1608   20352   40642    9ec2 kmutex4
  15619    1496   14412   31527    7b27 ksched1
  15567    1496   12060   29123    71c3 ksem0
  17063    1500   14436   32999    80e7 ksem1
  15504    1496   13228   30228    7614 kthread0
  16167    1496   14412   32075    7d4b kthread1
  18281    1512   14580   34373    8645 mbox1
  20611    1508   14940   37059    90c3 mqueue1
  15672    1504   12064   29240    7238 mutex0
```

```
  16678    1516   15664   33858    8442 mutex1
  17694    1508   16868   36070    8ce6 mutex2
  18203    1720   20344   40267    9d4b mutex3
  16352    1508   14428   32288    7e20 release
  15890    1500   14412   31802    7c3a sched1
  44196    1612  286332  332140   5116c stress_threads
  17891    1524   16864   36279    8db7 sync2
  16943    1512   15644   34099    8533 sync3
  15467    1496   13064   30027    754b thread0
  16134    1496   14420   32050    7d32 thread1
  17560    1512   15636   34708    8794 thread2
  16279    1500   24028   41807    a34f thread_gdb
  17051    1504   20376   38931    9813 timeslice
  17146    1504   21564   40214    9d16 timeslice2
  37313    1512  422380  461205   70995 tm_basic
```

Code size ARM (non-thumb)

Thumb does not compile out of the box w/AT91 EB40a for which this test was made.

```
$ arm-elf-size *
   text    data     bss     dec     hex filename
  25204     692   16976   42872    a778 bin_sem0
  26644     700   22096   49440    c120 bin_sem1
  26996     712   55584   83292   1455c bin_sem2
  27008     700   22100   49808    c290 bin_sem3
  28992     688   16944   46624    b620 clock0
  25456     692   19532   45680    b270 clock1
  34572    1160   19520   55252    d7d4 clockcnv
  26224     688   19508   46420    b554 clocktruth
  25204     692   16976   42872    a778 cnt_sem0
  26888     700   22108   49696    c220 cnt_sem1
  44180     752   27416   72348   11a9c dhrystone
  26088     688   19520   46296    b4d8 except1
  25236     692   16968   42896    a790 flag0
  29532     700   24668   54900    d674 flag1
  29508     704  109652  139864   22258 fptest
  25932     684   17016   43632    aa70 intr0
  25824     684   17112   43620    aa64 kalarm0
  24728     684   16956   42368    a580 kcache1
  25168     684   19512   45364    b134 kcache2
  28112     688   17168   45968    b390 kclock0
  25976     688   19524   46188    b46c kclock1
  26372     684   19512   46568    b5e8 kexcept1
  25140     684   16968   42792    a728 kflag0
  29824     688   24660   55172    d784 kflag1
  26896     704   24656   52256    cc20 kill
  26088     684   17028   43800    ab18 kintr0
  30812     692   22176   53680    d1b0 klock
  28504     688   22260   51452    c8fc kmbox1
  24984     684   16984   42652    a69c kmutex0
  26504     692   24704   51900    cabc kmutex1
  28792     900   34892   64584    fc48 kmutex3
  29264     796   34896   64956    fdbc kmutex4
  25240     684   22084   48008    bb88 ksched1
  25044     684   16968   42696    a6c8 ksem0
  26988     688   22100   49776    c270 ksem1
  25028     684   19512   45224    b0a8 kthread0
  25996     684   22080   48760    be78 kthread1
  28552     700   22252   51504    c930 mbox1
  31324     696   22612   54632    d568 mqueue1
  25108     692   16980   42780    a71c mutex0
  26464     704   24700   51868    ca9c mutex1
  27624     696   27280   55600    d930 mutex2
  28596     908   34884   64388    fb84 mutex3
  26156     696   22100   48952    bf38 release
  25460     688   22084   48232    bc68 sched1
  56356     828   45892  103076   192a4 stress_threads
  27900     712   27288   55900    da5c sync2
  26760     700   24692   52152    cbb8 sync3
  24924     684   19356   44964    afa4 thread0
  25868     684   22084   48636    bdfc thread1
  27452     700   24680   52832    ce60 thread2
  26136     688   42704   69528   10f98 thread_gdb
  27212     692   34916   62820    f564 timeslice
  52728     700  123332  176760   2b278 tm_basic
```

Phi memory map

TODO This probably belongs in the refdesign section. For now leaving it here because zealot refers to it. Not sure what else uses it.

The ZPU architecture does not define a memory map as such, but the GCC + libgloss + ecos hal library uses the memory map below. "Phi" is just a three letter word for the particular memory layout below that came about while developing the ZPU.

Address	Type	Name	Description
0x080A0000	Write	ZPU enable	Bit [31:1] Not used Bit [0] Enable ZPU operations 0 ZPU is held in Idle mode 1 ZPU running
0x080A0004	Read/ Write	GPIO data	Bit [31:0] input data 31:0 Bit [31:0] output data 31:0
0x080A0008	Read/ Write	GPIO direction	Bit [31:0] data direction 31:0 0 output 1 input (default)

0x080A000C	Read/ Write	ZPU Debug channel / UART to ARM7 TX **NOTE! ZPU side**	Bit [31:9] Not used Bit [8] TX buffer ready (valid on ready) 0 TX buffer not ready (full) 1 TX buffer ready Bit [7:0] TX byte (valid on write)
0x080A0010	Read	ZPU Debug channel / UART to ARM7 RX **NOTE! ZPU side**	Bit [31:9] Not used Bit [8] RX buffer data valid 0 RX buffer not valid 1 RX buffer valid Bit [7:0] RX byte (when valid)
0x080A0014	Read/ Write	Counter(1)	Bit [0] Reset counter (valid for write) 0 N/A 1 Reset counter Bit [1] Sample counter (valid for write) 0 N/A 1 Sample counter Bit [31:0] Counter bit 31:0
0x080A0018	Read	Counter(2)	Bit [31:0] Counter bit 63:32
0x080A0020	Read / Write	Global_Interrupt_mask	Bit [31:1] Not used Bit [0] Global intr. Mask 0 Interrupts enabled 1 Interrupts disabled
0x080A0024	Write	UART_INTERRUPT_ENABLE	Bit [31:1] Not used Bit [0] Debug channel / UART RX interrupt enable 0 Interrupt disable 1 Interrupt enable
0x080A0028	Read Write	UART_interrupt	Bit [31:1] Not used Bit [0] Debug channel / UART RX interrupt pending (Read) 0 No interrupt pending 1 Interrupt pending Bit [0] Clear UART interrupt (Write) 0 N/A 1 Interrupt cleared
0x080A002C	Write	Timer_Interrupt_enable	Bit [31:1] Not used Bit [0] Timer interrupt enable 0 Interrupt disable 1 Interrupt enable
0x080A0030	Read / Write	Timer_interrupt	Bit [31:2] Not used Bit [0] Timer interrupt pending (Read) 0 No interrupt pending 1 Interrupt pending Bit [1] Reset Timer counter (Write) 0 N/A 1 Timer counter reset Bit [0] Clear Timer interrupt (Write) 0 N/A 1 Interrupt cleared
0x080A0034	Write	Timer_Period	Bit [31:0] Interrupt period (write) Number of clock cycles between timer interrupts **NOTE!** The timer will start at Timer_Periode value and count **down** to zero, and generate an interrupt

.0x080A0038	Read	Timer_Counter	Bit [31:0] Timer counter (read)

TODO

TODO list

- fix the TODO in this doc that are just doc fixes
- organize the TODO list by priority and assign responsibility... if there are takers.
- converge on a single IO for core implementations.
- fill in performance table for Altera and Lattice.
- re-org CVS to make it easy to keep appropriate SW, RTL(verilog and VHDL) , scripts, verification stuff together. separation of tools, core, common, and ref design
- provide FPGA scripts.
- provide HDL regression environment.
- RAM model contribution needed. What is in opencore/common is not adequate.
- make wishbone bridge re-usable with all cores
- explicit example with UART from opencores in the above ref designs.
- discussion of tools needed and choose some to be supported by project. Need to deal with cygwin vs linux, VHDL vs verilog, open vs closed.... plus language support in simulators is sometimes lacking.
- setup.sh script needs to detect linux/cygwin, and should have install path option.
- shaping up the www.opencores.org pages.
- BSD and GPL licenses in the appropriate places.
- Currently there exists some pages at http://www.zylin.com/zpu.htm that explains about the ZPU. According to OpenCores policy this information should be moved to www.opencores.org. Patches gratefully accepted to do so!
- eCos HAL could be less RAM hungry
- Needs GDB stub support in eCos
- Could do with a Verilog implementation(ca. 600 lines to translate)
- Make little endian throughout. Currently instructions are stored big endian, loadb and storeb are big endian, but the data bus is treated as little endian. Creates some problems in type conversion.

Repository Re-org

I am proposing the following structure for the repository. It follows somewhat the way I've organized this document with seperation of core, common, and three SOC ref designs. New users go straight to the SOC that best matches their needs.

```
zpu/bin         # scripts and toolchain?  Want toolchain installed with project.  Tidier when working in multi user / multi project environment
zpu/doc         #
zpu/core/rtl    # RTL for the various core implementations.
zpu/core/sw     # crt0.s ?
zpu/common/rtl  # Re-use RTL such as RAM and UART
zpu/common/sim  # Re-use RTL and tools for regresion testing
zpu/common/sw   # ?
zpu/soc/minimal # Three levels of ref designs described above
       /basic
       /board
zpu/soc/*/rtl   # top level, arbiter, etc
zpu/soc/*/sw    # helloworld, dmips, etc. makefile/ROMS
zpu/soc/*/sim   # regression test area. makefile/scripts
zpu/soc/*/fpga  # syn and par area. makefile/scripts
zpu/tools       # zip/tarball of tool chains, simulator
```

Not sure where ecos fits.

Next generation ZPU

Based on feedback here is a list of a tenuous "consensus" for the next generation of the ZPU with some tentative ideas on implementation.

Goals

1. Reduce minimum code size footprint, i.e. BRAM code overhead. Non-trivial usable applications in 4kBytes of BRAM (single BRAM block).
2. Reduce minimum FPGA logic footprint by 20% or more. Goal <300 LUT for 32 bit ZPU
3. Weed out unnecessary ZPU variations and merge in useful features to a few recommeneded ZPU implementations.
4. Will someone be willing to contribute a heavily pipelined ZPU? Performance goal of 10 DMIPS w/DRAM & cache. This ZPU could run a TCP/IP stack with relevant performance to compete with stripped down ARM7 type systems.

GCC changes

The GCC changes planned are 100% backwards compatible with default options. However, a raft of options will be added to disable functionality so as to allow study and experimentation with the ZPU architecture.

1. Add options that allow defining single entry for all unknown instructions. Precisely how unknown instructions are handled will be defined by the HDL implementation. Currently the GCC backend places relatively strict limitations on how unknown/emulated instructions are handled. This will allow HDL implementations to have sparser instruction set support. Also this can allow sparse implementations of emualted instructions. This is especially important to reduce minimal BRAM requirements for small applications.
2. GCC needs 4 "hard" registers. These are today mapped to memory. GCC will allow specifying what address to use or alternatively not to use memory mapped hard registers at all.
3. Strip away unused instructions from GCC and add options to GCC for not emitting more advanced instructions. This will e.g. convert MULT/DIV into function calls to libgcc and thus make it easier to determine that microcode is not needed.

Floating point support

The ZPU does not currently have floating point support. Feedback from users indicates that single precision floating point support for addition, multiplication and float-to-integer convesion would be useful for small ZPU programs that sit in a tight control loop. Essentially the ZPU is then measuring something, doing a few calculations and then modifying the control signal.

Such control loops can be written in fixed point math, but that adds to the engineering effort and reduces clarity of the software implementation and the performance will probably be worse than for a hardware floating point version.

Pipelined floating point module

Design needs to be nailed down. **Goals:**

- 32 bit single precision floating point
- FADD => add two floats
- FMULT => multiply two floats
- FINT => convert float to int

The problem is divided into two:

1. One top level VHDL module for each of the operations above.
2. Integration into ZPU's are a separate problem that will not be addressed in this project.
3. add a memory mapped coprocessor interface to the above. This yields an example of a coprocessor which can be used for any custom calculations and allows interest to be gauged.

Throughput:

1. pipelined design where throughput is one operation per cycle with a fixed number of cycles delay.
2. there is no flow control or enable signal.

GCC support is not hard, but modifying GCC should considered after interest in this feature beyond a coprocessor has been gauged.

VHDL module interface

Patches anyone???