Amber Open Source Project

Amber 2 Core Specification

January 2011

Table of Contents

1 Introduction

The Amber 2 Core is a 32-bit RISC CPU with a 3-stage pipeline, a unified instruction & data cache, and a Wishbone interface, capable of 0.8 DMIPS per MHz. Amber is fully compatible with the ARM® v2a instruction set architecture (ISA) and is therefore supported by the GNU toolset.

Register based instructions execute in a single cycle, except for instructions involving multiplication. Load and store instructions require three cycles. The core's pipeline is stalled either when a cache miss occurs, or when the core performs a wishbone access.

This project includes a complete embedded system including the Amber 2 core and a number of peripherals, including UARTs and timers. Amber 2 does not contain a memory management unit (MMU) so it can only run the non-virtual memory variant of Linux. In addition the 2.6 version of Linux does not support the older version of the ISA implemented in the Amber 2 core, so the core is restricted to running versions of the Linux kernel from the 2.4 branch and earlier.

The core was developed in Verilog 2001, and is optimized for FPGA synthesis. For example there is no reset logic, all registers are reset as part of FPGA initialization. The complete system has been tested extensively on the Xilinx SP605 Spartan-6 FPGA board. The full Amber system uses 32% of the XC6SLX45T-3 FPGA LUTs, with the core itself occupying about 20% of the device using the default configuration, and runs at 40MHz. It has also been synthesized to a Virtex-6 device at 80MHz, but not tested on a real Virtex-6 device (yet). The maximum frequency is limited by the execution stage of the pipline which includes a 32-bit barrel shifter, 32-bit ALU and address incrementing logic.

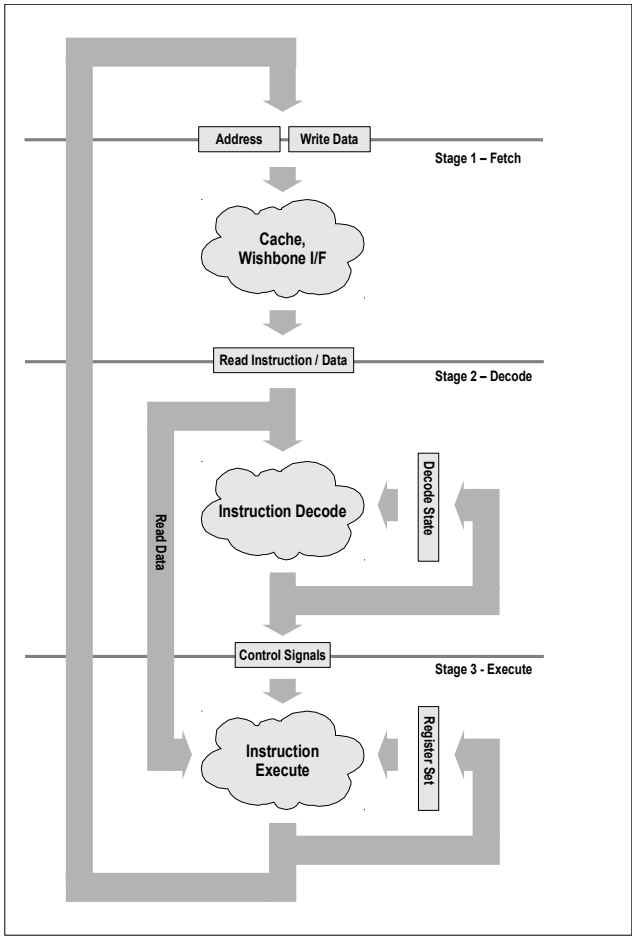
For a description of the ISA, see "Archimedes Operating System - A Dabhand Guide, Copyright Dabs Press 1991".

1.1 Features

- 3-stage pipeline.
- 32-bit Wishbone system bus.
- Unified instruction and data cache, with write through and a read-miss replacement policy. The cache can have 2, 3, 4 or 8 ways and each way is 4kB.
- Multiply and multiply-accumulate operations with 32-bit inputs and 32-bit output in 34 clock cycles using the Booth algorithm. This is a slow, but small multiplier.
- Little endian only, i.e. Byte 0 is stored in bits 7:0 and byte 3 in bits 31:24.

The following diagram shows the data flow through the 3-state core.

Figure 1 - *Amber 2 Core*



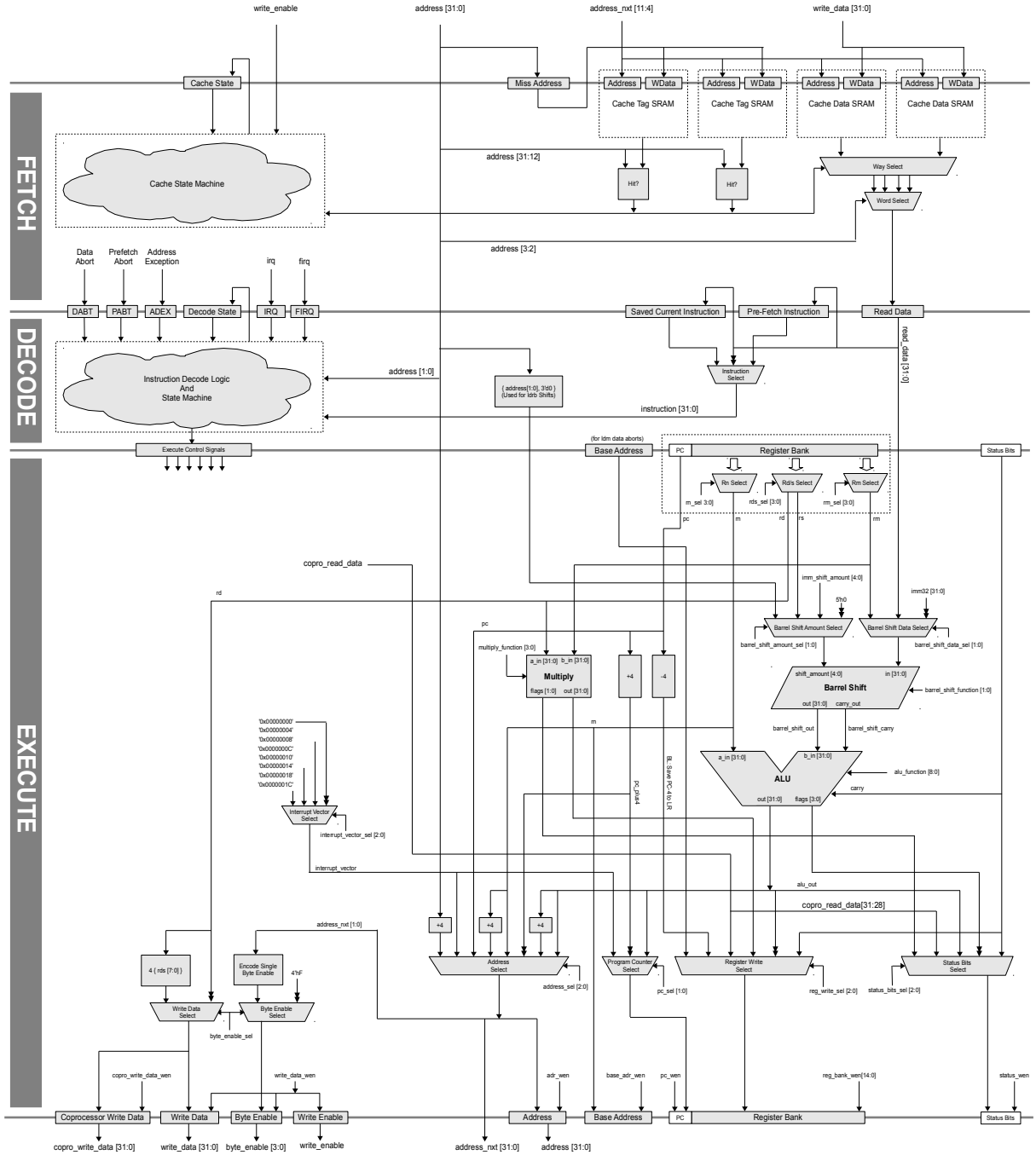
2 Pipeline Architecture

The Amber 2 core has a 3-stage pipeline architecture. The best way to think of the pipeline structure is of a circle. There is no start or end point. The output from each stage is registered and fed into the next stage. The three stages are;

- **Fetch** – The cache tag and data RAMs receive an unregistered version of the address output by the execution stage. The registered version of the address is compared to the tag RAM outputs one cycle later to decide if the cache hits or misses. If the cache misses, then the pipeline is stalled while the instruction is fetched from either boot memory or main memory via the Wishbone bus. The cache always does 4-word reads so a complete cache line gets filled. In the case of a cache hit, the output from the cache data RAM goes to the decode stage. This can either be an instruction or data word.
- **Decode** - The instruction is received from the fetch stage and registered. One cycle later it is decoded and the datapath control signals prepared for the next cycle. This stage contains a state machine that handles multi-cycle instructions and interrupts.
- **Execute –** The control signals from the decode stage are registered and passed into the execute stage, along with any read data from the fetch stage. The operands are read from the register bank, shifted, combined in the ALU and the result written back. The next address for the fetch stage is generated.

The following diagram shows the datapath through the three stages in detail. This diagram closely corresponds to the Verilog implementation. Some details, like the wishbone interface and coprocessor #15 have been left out so as not to overload the diagram completely.

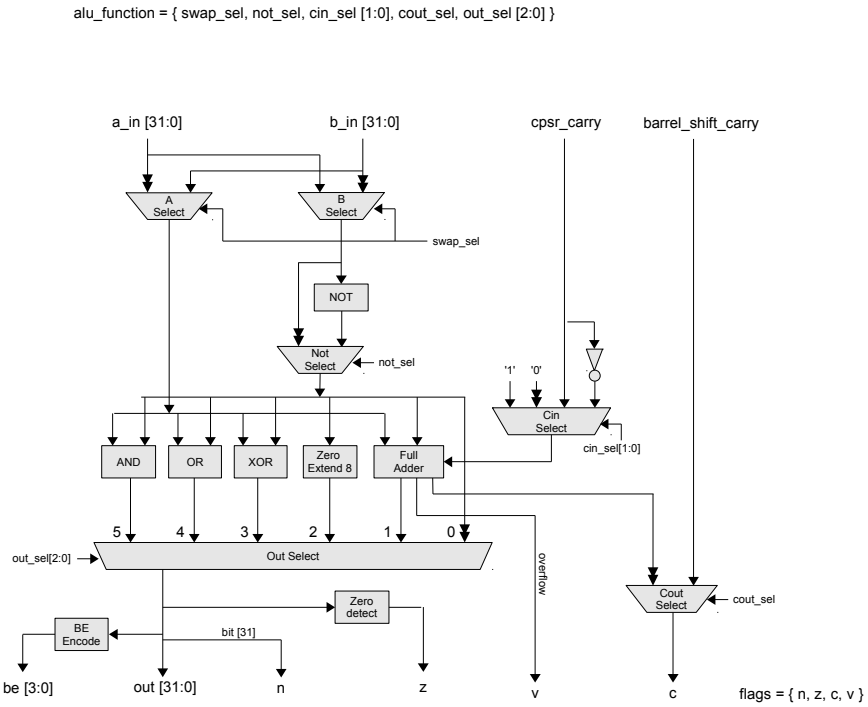
***Figure 2 -** Detailed Pipeline Structure*



2.1 ALU

The diagram below shows the structure of the Arithmetic Logic Unit (ALU). It consists of a set of different logical functions, a 32-bit adder and a mux to select the function.

Figure 3 - *ALU Structure*



The alu_function[6:0] bus in the core is a concatenation of the individual control signals in the ALU. The following table describes these control signals.

Table 1 ALU Function Encoding

Field	Function
swap_sel	Swaps the a and b inputs
not_sel	Selects the NOT version of b
cin_sel[1:0]	Selects the carry in to the full added from { c_in, !c_in, 1, 0 }. Note that bs_c_in is the carry_in from the barrel shifter.
cout_sel	Selects the carry out from { full_adder_cout, barrel_shifter_cout }
out_sel[2:0]	Selects the ALU output from { 0, b_zero_extend_8, b, and_out, or_out, xor_out, full_adder_out }

3 Instruction Set

The following table describes the instructions supported by the Amber 2 core.

Table 2 Amber 2 core Instruction Set

Name	Type	Syntax	Description
adc	REGOP	`adc{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Add with carry** adds two values and the Carry flag.
add	REGOP	`add{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Add** adds two values.
and	REGOP	`and{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**And** performs a bitwise AND of two values.
b	BRANCH	`b{<cond>} <target_address>`	**Branch** causes a branch to a target address.
bic	REGOP	`bic{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Bit** *clear* performs a bitwise AND of one value with the complement of a second value.
bl	BRANCH	`bl{<cond>} <target_address>`	**Branch and link** cause a branch to a target address. The resulting instruction stores a return address in the link register (r14).
cdp	COREGOP	`cdp{<cond>} <coproc>, <opcode_1>, <CRd>, <CRn>, <CRm>, <opcode_2>`	**Coprocessor data processing** tells a coprocessor to perform an operation that is independent of Amber registers and memory. This instruction is not currently implemented by the Amber core because there is no coprocessor in the system that requires it.
cmn	REGOP	`cmn{<cond>}{p} <Rn>, <shifter_operand>`	**Compare negative** compares one value with the twos complement of a second value, simply by adding the two values together, and sets the status flags. If the p flag is set, the pc and status bits are updated directly by the ALU output.
cmp	REGOP	`cmp{<cond>}{p} <Rn>, <shifter_operand>`	**Compare** compares two values by subtracting <shifter operand> from <Rn>, setting the status flags. If the p flag is set, the pc and status bits are updated directly by the ALU output.
eor	REGOP	`eor{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Exclusive OR** performs a bitwise XOR of two values.
ldc	CODTRANS	`lcd{<cond>} <coproc>, <CRd>, <addressing_mode>`	**Load coprocessor** loads memory data from a sequence of consecutive memory addresses to a coprocessor. This instruction is not currently implemented by the Amber core because there is no coprocessor in the system that requires it.
ldm	MTRANS	`ldm{<cond>}<addressing_mode> <Rn>{!}, <registers>`	**Load multiple** loads a non-empty subset, or possibly all, of the general-purpose registers from sequential memory locations. It is useful for block loads, stack operations and procedure exit sequences.
		`ldm{<cond>}<addressing_mode> <Rn>, <registers_without_pc>^`	This version loads User mode registers when the processor is in a privileged mode. This is useful when performing process swaps.
		`ldm{<cond>}<addressing_mode> <Rn>{!}, <registers_and_pc>^`	This version loads a subset, or possibly all, of the general-purpose registers and the PC from sequential memory locations. The status bits are also loaded. This is useful for returning from an exception.
ldr	TRANS	`ldr{<cond>} <Rd>, <addressing_mode>`	**Load register** loads a word from a memory address. If the address is not word-aligned, then the word is rotated left so that the byte addresses appears in bits [7:0] of Rd.
ldrb	TRANS	`ldr{<cond>}b <Rd>, <addressing_mode>`	**Load register byte** loads a byte from memory and zero-extends the byte to a 32-bit word.
mcr	CORTRANS	`mcr{<cond>} <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}`	**Move to coprocessor from register** passes the value of register <Rd> to a coprocessor.
mla	MULT	`mla{<cond>}{s} <Rd>, <Rm>, <Rs>, <Rn>`	**Multiply accumulate** multiplies two signed or unsigned 32-bit values, and adds a third 32-bit value. The least significant 32 bits of the result are written to the destination register.
mov	REGOP	`mov{<cond>}{s} <Rd>, <shifter_operand>`	**Move** writes a value to the destination register. The value can be either an immediate value or a value from a register,

Name	Type	Syntax	Description
			and can be shifted before the write.
mrc	CORTRANS	`mrc{<cond>} <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}`	**Move to register from coprocessor** causes a coprocessor to transfer a value to an Amber register or to the condition flags.
mul	MULT	`mul{<cond>}{s} <Rd>, <Rm>, <Rs>`	**Multiply** multiplies two signed or unsigned 32-bit values. The least significant 32 bits of the result are written to the destination register.
mvn	REGOP	`mvn{<cond>}{s} <Rd>, <shifter_operand>`	**Move not** generates the logical ones complement of a value. The value can be either an immediate value or a value from a register, and can be shifted before the MVN operation.
orr	REGOP	`orr{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Logical OR** performs a bitwise OR of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the OR operation.
rsb	REGOP	`rsb{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Reverse subtract** subtracts a value from a second value.
rsc	REGOP	`rsc{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Reverse subtract with carry** subtracts one value from another, taking account of any borrow from a preceding less significant subtraction. The normal order of the operands is reversed, to allow subtraction from a shifted register value, or from an immediate value.
sbc	REGOP	`sbc{<cond>}{s} <Rd>, <Rn>, <shifter_operand>`	**Subtract with carry** subtracts the value of its second operand and the value of NOT(Carry flag) from the value of its first operand. The first operand comes from a register. The second operand can be either an immediate value or a value from a register, and can be shifted before the subtraction.
stc	CODTRANS	`stc{<cond>} <coproc>, <CRd>, <addressing_mode>`	**Store coprocessor** stores data from a coprocessor to a sequence of consecutive memory addresses. This instruction is not currently implemented by the Amber core because there is no coprocessor in the system that requires it.
stm	MTRANS	`stm{<cond>}<addressing_mode> <Rn>{!}, <registers>`	**Store multiple** stores a non-empty subset (or possibly all) of the general-purpose registers to sequential memory locations. The '!' causes Rn to be updated. The registers are stored in sequence, the lowest-numbered register to the lowest memory address (start_address), through to the highest-numbered register to the highest memory address (end_address).
		`STM{<cond>}<addressing_mode> <Rn>, <registers>^`	This version stores a subset (or possibly all) of the User mode general-purpose registers to sequential memory locations. The registers are stored in sequence, the lowest-numbered register to the lowest memory address (start_address), through to the highest-numbered register to the highest memory address (end_address).
str	TRANS	`str{<cond>} <Rd>, <addressing_mode>`	**Store register** stores a word from a register to memory.
strb	TRANS	`str{<cond>}b <Rd>, <addressing_mode>`	**Store register byte** stores a byte from the least significant byte of a register to memory.
sub	REGOP	`sub{<cond>}{s} <Rd>, <Rn>, <shifter_operand>` `i.e. Rd = Rn - shifter_operand`	**Subtract** subtracts one value from a second value.
swi	SWI	`swi{<cond>} <immed_24>`	**Software interrupt** causes a SWI exception. <immed_24> Is a 24-bit immediate value that is put into bits[23:0] of the instruction. This value is ignored by the Amber core, but can be used by an operating system SWI exception handler to determine what operating system service is being requested.
swp	SWAP	`swp{<cond>} <Rd>, <Rm>, [<Rn>]`	**Swap** loads a word from the memory address given by the value of register <Rn>. The value of register <Rm> is then stored to the memory address given by the value of <Rn>, and the original loaded value is written to register <Rd>. If the same register is specified for <Rd> and <Rm>, this instruction swaps the value of the register and the value at the memory address.
swpb	SWAP	`swp{<cond>}b <Rd>, <Rm>, [<Rn>]`	**Swap Byte** swaps a byte between registers and memory. It loads a byte from the memory address given by the value of register <Rn>. The value of the least significant byte of register <Rm> is stored to the memory address given by

Name	Type	Syntax	Description
			<Rn>, the original loaded value is zero-extended to a 32-bit word, and the word is written to register <Rd>. Can be used to implement semaphores.
teq	REGOP	`teq{<cond>}{p} <Rn>, <shifter_operand>`	**Test equivalence** compares a register value with another arithmetic value. The condition flags are updated, based on the result of logically XORing the two values, so that subsequent instructions can be conditionally executed. If the p flag is set, the pc and status bits are updated directly by the ALU output.
tst	REGOP	`tst{<cond>}{p} <Rn>, <shifter_operand>`	**Test** compares a register value with another arithmetic value. The condition flags are updated, based on the result of logically ANDing the two values, so that subsequent instructions can be conditionally executed. If the p flag is set, the pc and status bits are updated directly by the ALU output.

4 Instruction Set Encoding

Table 3 Overall instruction set encoding table.

	Type	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data Processing	REGOP	Cond				0	0	**I**	Opcode				**S**	Rn				Rd				shifter_operand											
Multiply	MULT	Cond				0	0	0	0	0	0	**A**	**S**	Rd				Rn				Rs				1	0	0	1	Rm			
Single Data Swap	SWAP	Cond				0	0	0	1	0	**B**	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm			
Single Data Transfer	TRANS	Cond				0	1	**I**	**P**	**U**	**B**	**W**	**L**	Rn				Rd				Offset											
Block Data Transfer	MTRANS	Cond				1	0	0	**P**	**U**	**S**	**W**	**L**	Rn				Register List															
Branch	BRANCH	Cond				1	0	1	**L**	Offset																							
Coprocessor Data Transfer	CODTRANS	Cond				1	1	0	**P**	**U**	**N**	**W**	**L**	Rn				CRd				CP#				Offset							
Coprocessor Data Operation	COREGOP	Cond				1	1	1	0	CP Opcode				CRn				CRd				CP#				CP			0	CRm			
Coprocessor Register Transfer	CORTRANS	Cond				1	1	1	0	CP Opcode			**L**	CRn				Rd				CP#				CP			1	CRm			
Software Interrupt	SWI	Cond				1	1	1	1	Ignored by processor																							
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Where

I_{25} = Immediate form of shifter_operand

L_{24} = Link; Save PC to LR

U_{23} = 1; address = Rn + offset_12

= 0; address = Rn - offset_12

B_{22} = Byte (0 = word)

A_{21} = Accumulate

L_{20} = Load (0 = store)

S_{20} = Update Condition flags

P_{24}, W_{21} : Select different modes of operation

Table 4 Condition Encoding

Condition	Mnemonic extension	Meaning	Condition flag state
4'h0	eq	Equal	Z set
4'h1	ne	Not equal	Z clear
4'h2	cs / hs	Carry set / unsigned higher or same	C set
4'h3	cc / lo	Carry clear / unsigned lower	C clear
4'h4	mi	Minus / negative	N set
4'h5	pl	Plus / positive or zero	N clear
4'h6	vs	Overflow	V set
4'h7	vc	No overflow	V clear
4'h8	hi	Unsigned higher	C set and Z clear

Condition	Mnemonic extension	Meaning	Condition flag state
4'h9	ls	Unsigned lower or same	C clear or Z set
4'h10	ge	Signed greater than or equal	N == V
4'h11	lt	Signed less than	N != V
4'h12	gt	Signed greater than	Z == 0,N == V
4'h13	le	Signed less than or equal	Z == 1 or N != V
4'h14	al	Always (unconditional)	-
4'h15	nv	Never	-

Table 5 RegOp Opcode Encoding

Opcode	Mnemonic extension	Operation	Action
4'h0	**and**	Logical AND	Rd := Rn AND shifter_operand
4'h1	**eor**	Logical XOR	Rd := Rn XOR shifter_operand
4'h2	**sub**	Subtract	Rd := Rn - shifter_operand
4'h3	**rsb**	Reverse subtract	Rd := shifter_operand - Rn
4'h4	**add**	Add	Rd := Rn + shifter_operand
4'h5	**adc**	Add with carry	Rd := Rn + shifter_operand + Carry Flag
4'h6	**sbc**	Subtract with carry	Rd := Rn - shifter_operand - NOT(Carry Flag)
4'h7	**rsc**	Reverse subtract with carry	Rd := shifter_operand - Rn - NOT(Carry Flag)
4'h8	**tst**	Test	Update flags after Rn AND shifter_operand S bit always set
4'h9	**teq**	Test equivalence	Update flags after Rn EOR shifter_operand S bit always set
4'ha	**cmp**	Compare	Update flags after Rn – shifter_operand S bit always set
4'hb	**cmn**	Compare negated	Update flags after Rn + shifter_operand S bit always set
4'hc	**orr**	Logical (inclusive) OR	Rd := Rn OR shifter_operand
4'hd	**mov**	Move	Rd := shifter_operand (no first operand)
4'he	**bic**	Bit clear	Rd := Rn AND NOT(shifter_operand)
4'hf	**mvn**	Move NOT	Rd := NOT shifter_operand (no first operand)

Table 6 RegOp Shifter Operand Encoding

Format	Syntax	25 'I'	11	10	9	8	7	6	5	4	3	2	1	0
32-bit immediate	#<immediate>	1	rotate_imm				imm_8							
Immediate shifts	<Rm>	0	5'h0					2'h0		0	Rm			
	<Rm>, lsl #<shift_imm>	0	shift_imm					shift		0	Rm			
	<Rm>, lsr #<shift_imm>													
	<Rm>, asr #<shift_imm>													
	<Rm>, ror #<shift_imm>													
	<Rm>, rrx	0	5'h0					2'b11		0	Rm			
Register Shifts	<Rm>, lsl <Rs>	0	Rs				0	shift		1	Rm			
	<Rm>, lsr <Rs>													
	<Rm>, asr <Rs>													

Format	Syntax	25 'I'	11	10	9	8	7	6	5	4	3	2	1	0
	<Rm>, ror <Rs>													

Table 7 RegOp Shift Encoding

Condition	Type
2'h0	Logical Shift Left
2'h1	Logical Shift Right
2'h2	Arithmetic Shift Right (sign extend)
2'h3	Rotate Right with Extent (CO -> bit 31, bit 0 -> CO), if shift amount = 0, else Rotate Right

Table 8 RegOp Rotate Immediate Value Encoding

Value	32-bit immediate value
4'h0	{ 24'h0, imm_8[7:0] }
4'h1	{ imm_8[1:0], 24'h0, imm_8[7:2] }
4'h2	{ imm_8[3:0], 24'h0, imm_8[7:4] }
4'h3	{ imm_8[5:0], 24'h0, imm_8[7:6] }
4'h4	{ imm_8[7:0], 24'h0 }
4'h5	{ 2'h0, imm_8[7:0], 22'h0 }
4'h6	{ 4'h0, imm_8[7:0], 20'h0 }
4'h7	{ 6'h0, imm_8[7:0], 18'h0 }
4'h8	{ 8'h0, imm_8[7:0], 16'h0 }
4'h9	{ 10'h0, imm_8[7:0], 14'h0 }
4'h10	{ 12'h0, imm_8[7:0], 12'h0 }
4'h11	{ 14'h0, imm_8[7:0], 10'h0 }
4'h12	{ 16'h0, imm_8[7:0], 8'h0 }
4'h13	{ 18'h0, imm_8[7:0], 6'h0 }
4'h14	{ 20'h0, imm_8[7:0], 4'h0 }
4'h15	{ 22'h0, imm_8[7:0], 2'h0 }

Table 9 Single Data Transfer Offset Encoding

Category	Type	Syntax	25 'I'	24 'P'	23 'U'	22 'B'	21 'W'	20 'L'	11	10	9	8	7	6	5	4	3	2	1	0
Immediate offset / index	Immediate offset	[<Rn>, #+/-<offset_12>]	0	1	-	-	0	-	offset_12											
	Immediate pre-indexed	[<Rn>, #+/-<offset_12>]!	0	1	-	-	1	-	offset_12											
	Immediate post-indexed	[<Rn>], #+/-<offset_12>	0	0	-	-	0	-	offset_12											
	Immediate post-indexed, unprivilaged memory access	[<Rn>], #+/-<offset_12>	0	0	-	-	1	-	offset_12											
Register offset / index	Register offset	[<Rn>, +/-<Rm>]	1	1	-	-	0	-	8'h0									Rm		
	Register pre-indexed	[<Rn>, +/-<Rm>]!	1	1	-	-	1	-	8'h0									Rm		
	Register post-indexed	[<Rn>], +/-<Rm>	1	0	-	-	0	-	8'h0									Rm		
	Register post-indexed, unprivilaged memory access	[<Rn>], +/-<Rm>	1	0	-	-	1	-	8'h0									Rm		
Scaled register offset / index	Scaled register offset	[<Rn>, +/-<Rm>, <shift> #<shift_imm>]	1	1	-	-	0	-	shift_imm					shift		0		Rm		

Category	Type	Syntax	25 'I'	24 'P'	23 'U'	22 'B'	21 'W'	20 'L'	11	10	9	8	7	6	5	4	3	2	1	0
	Scaled register pre-indexed	[<Rn>, +/-<Rm>, <shift> #<shift_imm>]!	1	1	-	-	1	-	shift_imm					shift		0	Rm			
	Scaled register post-indexed	[<Rn>], +/-<Rm>, <shift> #<shift_imm>	1	0	-	-	0	-	shift_imm					shift		0	Rm			
	Scaled register post-indexed, unprivilaged memory access	[<Rn>], +/-<Rm>, <shift> #<shift_imm>	1	0	-	-	1	-	shift_imm					shift		0	Rm			

Where;

Pre-indexed: Address adjusted before access

Post-indexed: Address adjusted after access

I_{25}, P_{24} and W_{21} encode the instruction as shown in the table above.

U_{23} = 1; address = Rn + offset_12

= 0; address = Rn – offset_12

B_{22} = 0; data type is 32-bit word

= 1; data type is byte

L_{20} = 1; load

= 0; store

4.1 Load & Store Multiple

Table 10 Index options with ldm and stm

Mode	Stack Load Equivalent	Stack Store Equivalent	Instructions	24 'P'	23 'U'	22 'S'	21 'W'	20 'L'
Increment After (ia)	Full Descending (fd)	Empty Ascending (ea)	ldmia, stmia, ldmfd, stmea	0	1	-	-	-
Increment Before (ib)	Empty Descending (ed)	Full Ascending (fa)	lmdib, stmib, ldmed, stmfa	1	1	-	-	-
Decrement After (da)	Full Ascending (fa)	Empty Descending (ed)	ldmda, stmda, ldmfa, stmed	0	0	-	-	-
Decrement Before (db)	Empty Ascending (ea)	Full Descending (fd)	lmddb, stmdb, ldmea, stmfd	1	0	-	-	-

S_{22}

The S bit for ldm that loads the PC, the S bit indicates that the status bits loaded. For ldm instructions that do not load the PC and all stm instructions, the S bit indicates that when the processor is in a privileged mode, the User mode banked registers are transferred instead of the registers of the current mode. Ldm with the S bit set is unpredictable in User mode.

W_{21}

Indicates that the base register is updated after the transfer.

L_{20}

Distinguishes between Load (L==1) and Store (L==0) instructions.

4.2 Booth's Multiplication Algorithm

Booth's algorithm involves repeatedly adding one of two predetermined values A and S to a product P, then performing a rightward arithmetic shift on P. Let m and r be the multiplicand and multiplier, respectively; and let x and y represent the number of bits in m and r.

1. Determine the values of A and S, and the initial value of P. All of these numbers should have a length equal to (x + y + 1).
    1. A: Fill the most significant (leftmost) bits with the value of m. Fill the remaining (y + 1) bits with zeros.
    2. S: Fill the most significant bits with the value of (−m) in two's complement notation. Fill the remaining (y + 1) bits with zeros.
    3. P: Fill the most significant x bits with zeros. To the right of this, append the value of r. Fill the least significant (rightmost) bit with a zero.
2. Examine the two least significant (rightmost) bits of P.
    1. If they are 01, find the value of P + A. Ignore any overflow.
    2. If they are 10, find the value of P + S. Ignore any overflow.
    3. If they are 00, do nothing. Use P directly in the next step.
    4. If they are 11, do nothing. Use P directly in the next step.
3. Arithmetically shift the value obtained in the 2nd step by a single place to the right. Let P now equal this new value.
4. Repeat steps 2 and 3 until they have been done y times.
5. Drop the least significant (rightmost) bit from P. This is the product of m and r.

Here is the algorithm in C-code form;

```
unsigned int mul ( unsigned int Rm, unsigned int Rs )
{
unsigned int multiply_result_hi, multiply_result_lo, n, booth_bits;

for (n=0;n<33;n++){
    if (n==0) {
        booth_bits          = ((Rs & 1)<<1);
        multiply_result_lo  = Rs;
        if      (booth_bits == 1) { multiply_result_hi = Rm;     }
        else if (booth_bits == 2) { multiply_result_hi = ~Rm + 1;}
        else                      { multiply_result_hi = 0;      }
        }
    else {
        booth_bits         =  multiply_result & 3;
        multiply_result_lo = (multiply_result_lo >>1) | (( multiply_result_hi & 1)<<31);
        multiply_result_hi = (multiply_result_hi >>1) | (multiply_result_hi & 0x80000000);
        if (booth_bits == 1) { multiply_result_hi = multiply_result_hi + Rm;        }
        if (booth_bits == 2) { multiply_result_hi = multiply_result_hi + (~Rm + 1); }
```

```
        }
    }

return multiply_result_lo;
}
```

5 Interrupts

Table 11 Interrupt Types

Interrupt Type	Processor Mode	Address
Reset	Supervisor (svc)	0x00000000
Undefined Instructions	Supervisor (svc)	0x00000004
Software Interrupt (SWI)	Supervisor (svc)	0x00000008
Prefetch Abort (instruction fetch memory abort)	Supervisor (svc)	0x0000000C
Data Abort (data access memory abort)	Supervisor (svc)	0x00000010
Address exception	Supervisor (svc)	0x00000014
IRQ (interrupt)	IRQ (irq)	0x00000018
FIRQ (fast interrupt)	FIRQ (firq)	0x0000001C
-	User (usr)	-

The modes other than User mode are known as privileged modes. They have full access to system resources and can change mode freely. When an exception occurs, the banked versions of r14, the link register, is used to save the pc value and status bits.

6 Registers

Table 12 Register Sets

User (USR)	Supervisor (SVC)	Interrupt (IRQ)	Fast Interrupt (FIRQ)
r0			
r1			
r2			
r3			
r4			
r5			
r6			
r6			
r7			
r8			r8_firq
r9			r9_firq
r10			r10_firq
r11 (fp)			r11_firq
r12 (ip)			r12_firq
r13 (sp)	r13_svc	r13_irq	r13_firq
r14 (lp)	r14_svc	r14_irq	r14_firq
r15 (pc)			

Table 13 Status Bits – Part of the PC

Field	Position	Type	Description
flags	[31:28]	User Writable	{ Negative, Zero, Carry, oVerflow }
I	27	Privileged	IRQ mask, disables IRQs when high
F	26	Privileged	FIRQ Mask, disables FIRQs when high
mode	[1:0]	Privileged	Processor mode 3 - Supervisor 2 - Interrupt 1 - Fast Interrupt 0 - User

7 Cache

The Amber Cache size is optimized to use FPGA Block RAMs. It implements a unified write through, read replace, cache, with a 4-word line size. Lines are replaced on read misses and the way is selected randomly. Each way has 256 lines of 16 bytes. 256 lines x 16 bytes x 2 ways = 8k bytes. The address tag is 20 bits. The cache can be configured with either 2, 3, 4 or 8 ways.

Table 14 Cache Specification

Ways	2	3	4	**8**
Lines per way	256	256	256	256
Words per line	4	4	4	4
Total words	2048	3072	4096	8192
Total bytes	8192	12288	16384	32768
FPGA 9K Block RAMs	8 + 2 = 10	12 + 3 = 15	16 + 4 = 20	32 + 8 = 40

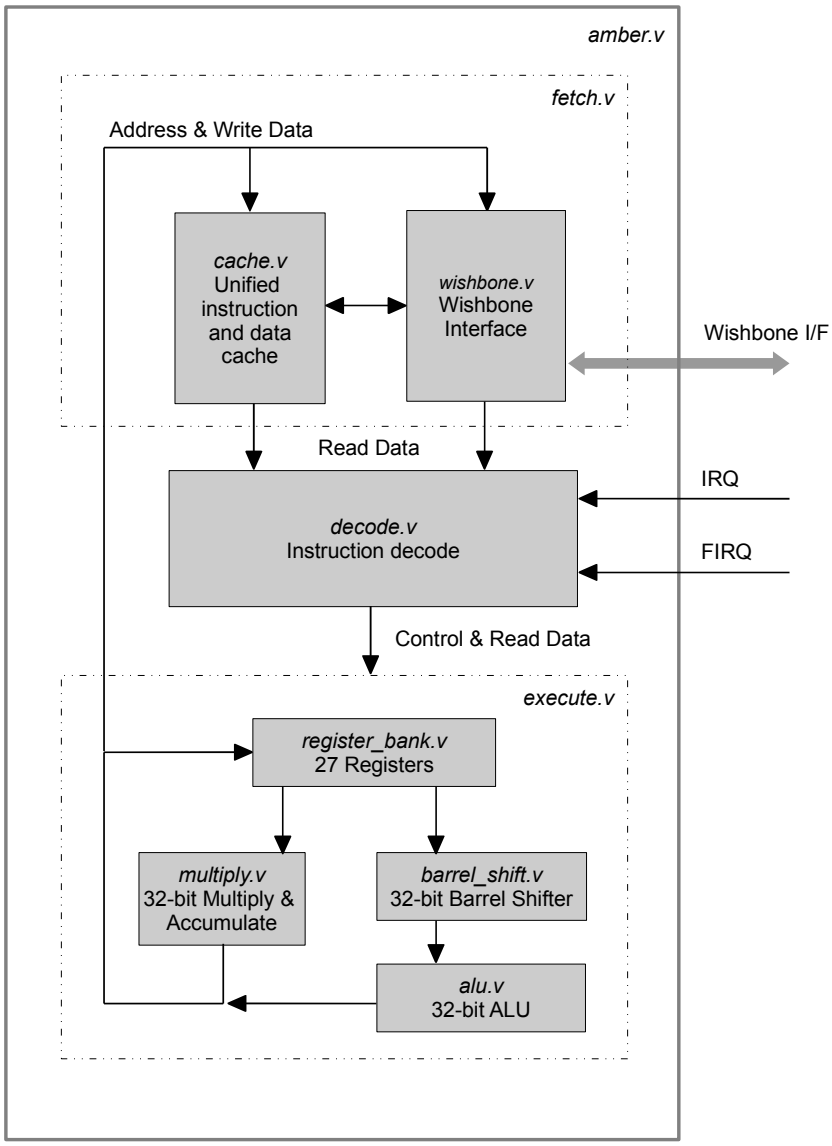
8 Amber Core Source Files

Table 15 Amber 2 Core Source Files

Name	Description
amber_config_definesv	Defines used to configure the amber core. The number of ways in the cache is configurable. Also contains a set of debug switches which enable debug messages to be printed during simulation.
amber_localparams.v	Local parameters used in various amber source files.
wishbone.v	The Wishbone interface connecting the Execute stage and Cache to the rest of the system. Instantiated in Fetch.
alu.v	The arithmetic logic unit. Includes a 32-bit 2's compliment adder/subtractor as well as logical functions such as AND and XOR.
amber_functions.v	Common Verilog functions.
amber.v	Top-level Amber module.
barrel_shifter.v	32-bit barrel shifter instantiated in Execute.
cache.v	Synthesizable cache. Instantiated in Fetch. Cache misses cause the core to stall. The cache then issues a quad-word read on the wishbone bus, starting with the word that missed, and wrapping at the quad-word boundary.
coprocessor.v	Co-processor 15 registers and control signals. Instantiated in Amber.
decode.v	The instruction decode pipeline stage. Instantiated in Amber.
decompile.v	The decompiler. This is a non-synthesizable debug module. It creates the amber.dis file which lists every instruction executed by the core.
execute.v	The execute pipeline stage. Instantiated in Amber. It contains the alu, multiply, and register_bank sub-modules.
fetch.v	The Fetch stage. This contains the Cache and Wishbone interface modules. It is instantiated in Amber.
multiply.v	32-bit 2's compliment multiply and multiply-accumulate unit. Uses the Booth algorithm and takes 34 cycles to complete a signed multiply-accumulate operation but is quite small in logic area.
register_bank.v	Contains all 27 registers r0 to r15 for each mode of operation. Registers are implemented as real flipflops in the FPGA. This allows multiple read and write access to the bank simultaneously.

The following diagram shows the Verilog module structure within the Amber 2 core.

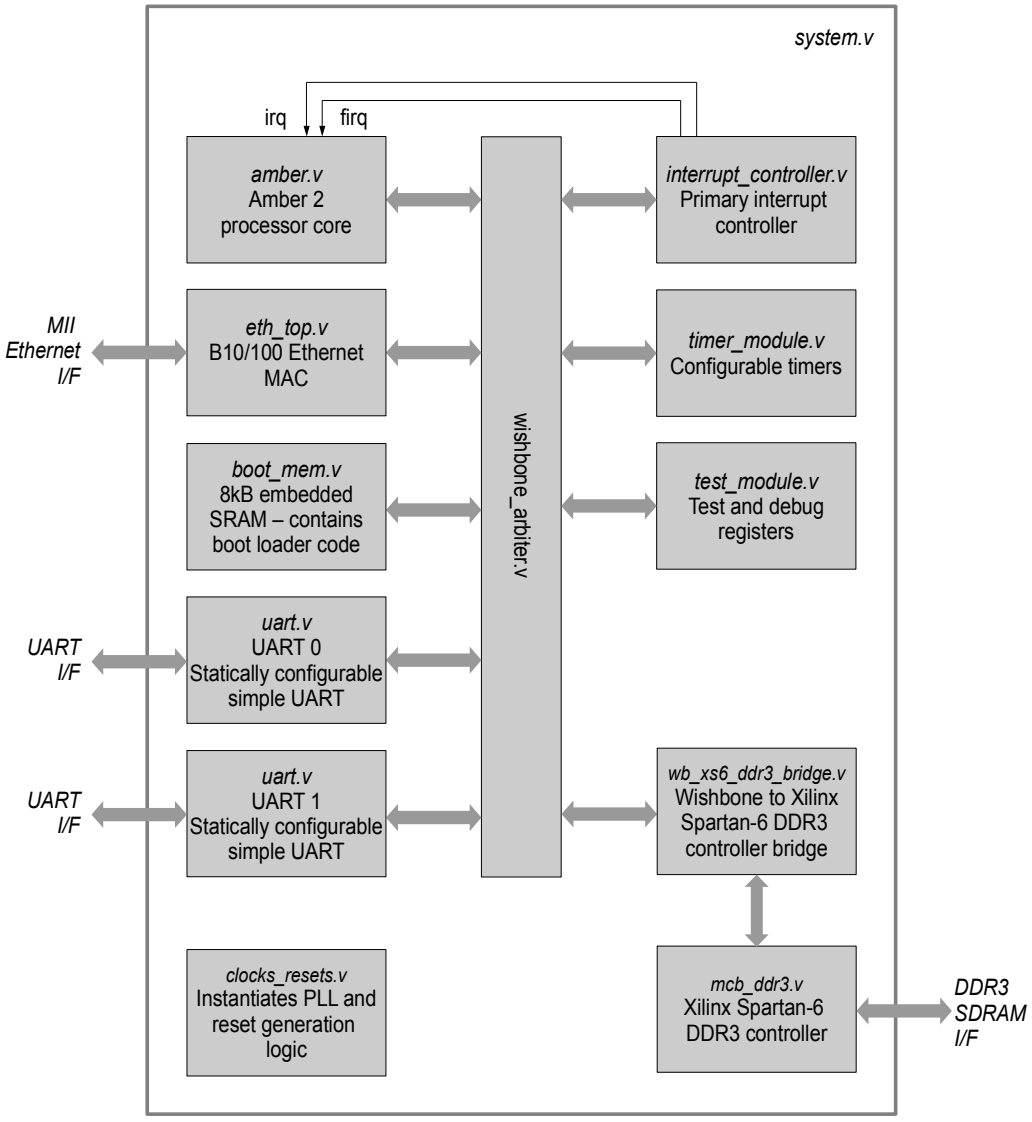
Figure 4 - *Amber 2 Core Verilog Structure*



9 Amber FPGA System

The FPGA system is a complete embedded processor system which included all peripherals needed to run Linux, including UART, timers and an Ethernet (MII) port. The following diagram shows the entire system.

Figure 5 - *Amber FPGA System*



All the Verilog source code was specifically developed for this project with the exception of the following modules;

- *mcb_ddr3.v*. The Xilinx Spartan-6 DDR3 controller was generated by the Xilinx Coregen tool. The files are not included with the project for copyright reasons. It is up to the user to optain the ISE software from Xilinx and generate the correct memory controller. Note that Wishbone bridge modules are included that support both the Xilinx Spartan-6 DDR3 controller and the Virtex-6 controller.

- *eth_top.v*. This module is from the Opencores Ethernet MAC 10/100 Mbps project. The Verilog code is included for convenience. It has not been modified, except to provide a memory module for the Spartan-6 FPGA.

10 Verilog simulations

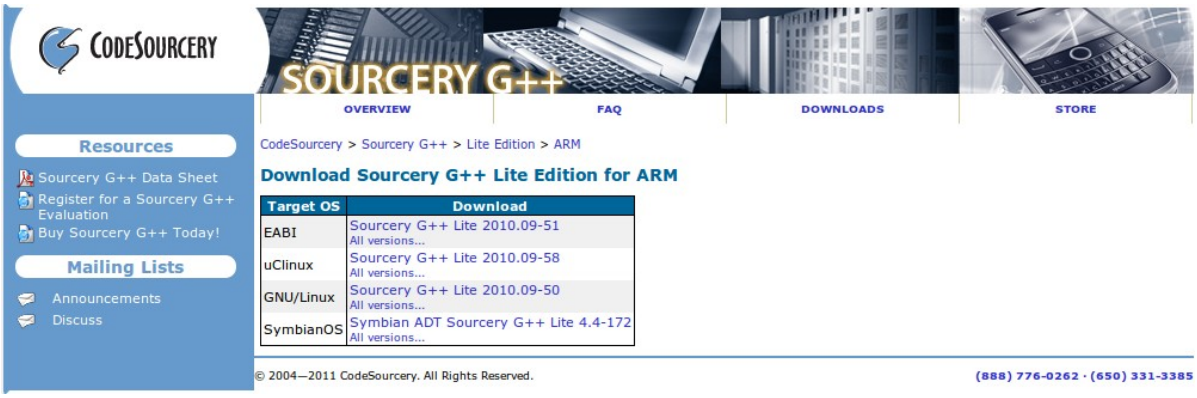
10.1 Required tools

To run simulations you need to have a GNU cross compiler a Verilog simulator installed. I use the Code Sourcery G++ Lite package for GNU and Modelsim SE v6.5 for the Verilog simulator. I also tested the code with the Veritak Verilog simulator running under WINE. To run simulations using the Xilinx library models for RAMs, PLLs etc., you also need to have Xilinx ISE installed.

The easiest way to install the GNU tool chain is to download a ready made package. Code Sourcery provides a free one. To download the Sourcery package, go to this page
http://www.codesourcery.com/sgpp/lite/arm

and click on download the current release. This brings up the following page;

Figure 6 - *Code Sourcery GNU Download*



Select the **GNU/Linux** version and then the **IA32 GNU/Linux** Installer on the next page. Once the package is installed, add the following to your .bashrc file, where the PATH is set to where you install the Code Sourcery GNU package.

```
# Change /proj/amber to where you saved the amber package on your system
export AMBER_BASE=/proj/amber/trunk

# Change /opt/Sourcery to where the package is installed on your system
PATH=/opt/Sourcery/bin:${PATH}

# AMBER_CROSSTOOL is the name added to the start of each GNU tool in
# the Sourcery bin directory. This variable is used in the Makefiles to use
# the correct tool to compile code for the Amber core
export AMBER_CROSSTOOL=arm-none-linux-gnueabi

# Xilinx ISE installation directory
# This should be configured for you when you install ISE.
# But check that is has the correct value
# It is used in the run script
export XILINX=/opt/Xilinx/11.1/ISE
```

10.1.1 GNU Tools Usage

It's important to remember to use the correct switches with the GNU tools to restrict

the ISA to the set of instructions supported by the Amber 2 core. The switches are already set in the Makefiles included with the Amber 2 core. Here are the switches to use with gcc;

```
-march=armv2a -mno-thumb-interwork
```

These switches specify the correct version of the ISA, and tell the compiler not to create bx instructions. Here is the switch to use with ld;

```
--fix-v4bx
```

This switch converts any bx instructions (which are not supported) to mov pc, lr. Here is an example usage;

```
arm-none-linux-gnueabi-gcc -c -Os -march=armv2a -mno-thumb-interwork -ffreestanding
   -I../include   -c -o boot-loader.o boot-loader.c
arm-none-linux-gnueabi-gcc -I../include    -c -o start.o start.S
arm-none-linux-gnueabi-gcc -c -Os -march=armv2a -mno-thumb-interwork -ffreestanding
   -I../include   -c -o crc16.o crc16.c
arm-none-linux-gnueabi-gcc -c -Os -march=armv2a -mno-thumb-interwork -ffreestanding
   -I../include   -c -o xmodem.o xmodem.c
arm-none-linux-gnueabi-gcc -c -Os -march=armv2a -mno-thumb-interwork -ffreestanding
   -I../include   -c -o elfsplitter.o elfsplitter.c
arm-none-linux-gnueabi-ld -Bstatic -Map boot-loader.map  --strip-debug --fix-v4bx -o boot-
   loader.elf -T sections.lds boot-loader.o start.o crc16.o xmodem.o elfsplitter.o
   ../mini-libc/printf.o ../mini-libc/libc_asm.o ../mini-libc/memcpy.o
arm-none-linux-gnueabi-objcopy -R .comment -R .note boot-loader.elf
../tools/amber-elfsplitter boot-loader.elf > boot-loader.mem
../tools/amber-memparams.sh boot-loader.mem boot-loader_memparams.v
arm-none-linux-gnueabi-objdump -C -S -EL boot-loader.elf > boot-loader.dis
```

A full list of switches for gcc can be found here;
http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/ARM-Options.html#ARM-Options

10.2 Running Simulations

A script to run tests is included, $AMBER_BASE/hw/tools/run.sh. This script calls a Makefile to compile the test, then calls the modelsim tools to compile and run the simulation, as follows;

1. vlib to create a modelsim library work directory, if not already created
2. vlog to compile the Verilog source code
3. vsim to run the simulation, either in command line mode or with the full user interface, depending on the switches passed into the run script.

For ease of use, create a link to this script in the sim directory;

```
cd $AMBER_BASE/hw/sim
ln -s ../tools/run.sh run
```

Then to run a test, simply type run with the test name, e.g.

```
run add
```

To get a list of the switches that run understands, type;

```
run -h
Usage:
run <test_name> [-a] [-g] [-d] [-t] [-s] [-v]
 -h : Help
 -a : Run hardware tests (all tests in $AMBER_BASE/hw/tests)
 -g : Use Modelsim GUI
 -d <cycle number to start dumping>: Create vcd file
 -t <cycle number to start dumping>: Create vcd file and terminate
 -s : Use Xilinx Spatran6 Libraries (slower sim)
 -v : Use Xilinx Virtex6 Libraries (slower sim)
```

10.3 Simulation output files

10.3.1 Disassembly Output File

The disassembly file, amber.dis, is generated by default during a simulation. It is located in the $AMBER_BASE/hw/sim directory.

To turn off generation of this file, comment the line where AMBER_DECOMPILE is defined in $AMBER_BASE/hw/vlog/amber/amber_config_defines.v.

Here is an example of the dissassembly output file. The first column gives the time that the instruction was executed. The time is specified in sys_clk ticks. The second column gives the address of the instruction being executed and the next column gives the instruction. If an instruction is not executed because of a conditional execution code, this is marked with a – character in front of the instruction. For load and store instructions, the actual memory access is given below the instruction. This is the complete listing for the add test.

```
      264         0:   mov     r1,  #3
      267         4:   mov     r2,  #1
      270         8:   add     r3,  r1,  r2
      273         c:   cmp     r3,  #4
      276        10:  -movne   r10, #10
      279        14:  -bne     b4
      282        18:   mov     r4,  #0
      285        1c:   mov     r5,  #0
      288        20:   add     r6,  r5,  r4
      291        24:   cmp     r6,  #0
      294        28:  -movne   r10, #20
      297        2c:  -bne     b4
      300        30:   mov     r7,  #0
      303        34:   mvn     r8,  #0
      306        38:   add     r9,  r7,  r8
      309        3c:   cmn     r9,  #1
      312        40:  -movne   r10, #30
      315        44:  -bne     b4
      318        48:   mvn     r1,  #0
      321        4c:   mov     r2,  #0
      324        50:   add     r3,  r1,  r2
      327        54:   cmn     r3,  #1
      330        58:  -movne   r10, #40
      333        5c:  -bne     b4
      336        60:   mvn     r4,  #0
      339        64:   mvn     r5,  #0
      342        68:   add     r6,  r4,  r5
      345        6c:   cmn     r6,  #2
      348        70:  -movne   r10, #50
      351        74:  -bne     b4
      354        78:   mvn     r7,  #0
      357        7c:   mvn     r8,  #254
      360        80:   add     r9,  r7,  r8
      363        84:   cmn     r9,  #256
      366        88:  -movne   r10, #60
```

```
      369        8c:  -bne     b4
      372        90:   ldr     r1, [pc, #60]
      377              read    addr d4, data 7fffffff
      381        94:   mov     r2,  #1
      384        98:   adds    r3,  r1,  r2
      387        9c:  -bvc     b4
      390        a0:   ldr     r0, [pc, #48]
      395              read    addr d8, data 80000000
      399        a4:   cmp     r0,  r3
      402        a8:  -movne   r10, #70
      405        ac:  -bne     b4
      408        b0:   b       c0
      410              jump    from b0 to c0, r0 80000000, r1 7fffffff
      417        c0:   ldr     r11, [pc, #8]
      422              read    addr d0, data f0000000
      426        c4:   mov     r10, #17
      429        c8:   str     r10, [r11]
      432              write   addr f0000000, data 00000011, be f
```

10.3.2 VCD Output File

The VCD dump file is $AMBER_BASE/hw/sim/sim.vcd.

The VCD dump file, sim.vcd, is useful for debugging very long simulations where you just want to get waves for a period of time around where a bug is occurring. Usually with long simulations you can not dump the entire simulation because the dump file would get too large. You can use the free waveform viewer, gtkwave, to view this file.

To create a VCD output file, you can use the -d or -t switch with the run script. Alternatively you can uncomment the defines AMBER_DUMP_VCD and AMBER_DUMP_START in the file $AMBER_BASE/hw/vlog/system/system_config_defines.v.

For example,

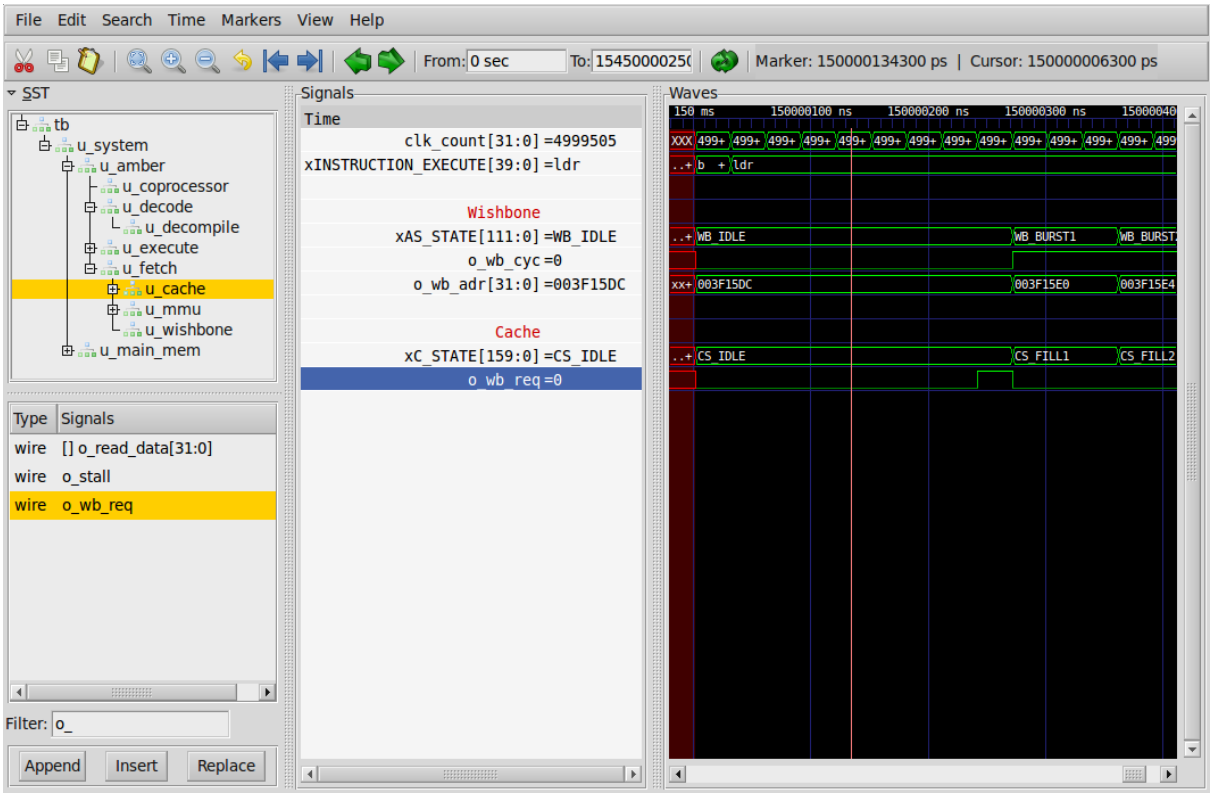
```
run linux -t 5000000
```

This runs the test vmlinux, and turns on waveform dumping at system clock tick 5,000,000. The dumping continues for a few thousand cycles and then the test terminates with a fatal error. The lenght of time the dumping runs for is configured with the AMBER_DUMP_LENGTH define in $AMBER_BASE/hw/vlog/system/system_config_defines.v.

The signals included in the VCD dump file are specified in the file $AMBER_BASE/hw/vlog/tb/dumpvcd.v.

The following diagram shows a screen grab of the GTKWave viewer looking at signals 5,000,000 cycles into a simulation..

Figure 7 - *GTKWave waveform viewer*



10.4 Tests

The following list of tests are used to verify the correct operation of all the instructions, interrupts, the cache and peripherals. The tests are written in assembly. Several of the tests were added when a specific bug was found while debugging the core. To run a specific test, simply use run <test-name>, e.g.

```
cd $AMBER_BASE/hw/sim
run barrel_shift
```

Each test generates pass or fail when it completes;

```
# ++++++++++++++++++++
# Passed barrel_shift
# ++++++++++++++++++++
```

To run the complete test suite;

```
cd $AMBER_BASE/hw/sim
run -a
```

Once the run is complete look at the output file hw-tests.log in the $AMBER_BASE/hw/sim/ directory to check the results. All tests should pass.

The following table describes each test. The source files for these tests are in the

directory $AMBER_BASE/hw/tests.

Table 16 Amber Core Verification Tests

Name	Description
adc	Tests the adc instruction. Adds 3 32-bit numbers using adc and checks the result.
addr_ex	Tests an address exception interrupt. Sets the pc to 0x3ffffc and executes a nop. The pc then increments to 0x4000000 triggering an address exception.
add	Tests the add instruction. Runs through a set of additions of positive and negative numbers, checking that the results are correct. Also tests that the 's' flag on the instruction correctly sets the condition flags.
barrel_shift_rs	Tests the barrel shift operation with a mov instruction, when the shift amount is a register value. Test that shift of 0 leaves Rm unchanged. Tests that a shift of > 32 sets Rm and carry out to 0.
barrel_shift	Tests the barrel shift operation with a mov instruction when the shift amount is an immediate value.Tests lsl, lsr and ror.
bcc	Tests branch on carry clear.
bic_bug	Test added to catch specific bug with the bic instruction. The following instruction stored the result in r3, instead of r2 tst r2, r0, lsl r3 bicne r2, r2, r0, lsl r3
bl	Test Branch and Link instruction. Checks that the correct return address is stored in the link register (r14).
cache1	Contains a long but simple code sequence. The entire sequence can fit in the cache. This sequence is executes 4 times, so three times it will execute from the cache. Test passes if sequence executes correctly.
cache2	Tests simple interactin between cached data and uncached instruction accesses.
cache3	Tests that the cache can write to and read back multiple times from 2k words in sequence in memory - the size of the cache.
cacheable_area	Tests the cacheable area co-processor function.
cache_flush	Tests the cache flush function. Does a flush in the middle of a sequence of data reads. Checks that all the data reads are correct.
cache_swap_bug	Tests the interaction between a swap instruction and the cache. Runs through a main loop multiple times with different numbers of nop instructions before the swp instruction to test a range of timing interactions between the cache state machine and the swap instruction.
cache_swap	Fills up the cache and then does a swap access to data in the cache. That data should be invalidated. Check by reading it again.
change_mode	Tests teq, tst, cmp and cmn with the p flag set. Starts in supervisor mode, changes to Interrupt mode then Fast Interrupt mode, then supervisor mode again and finally User mode.
change_sbits	Change status bits. Tests movs where the destination register is r15, the pc. Depending on the processor mode and whether the s bit is set or not, some or none of the status bits will change.
ddr31	Word accesses to random addresses in DDR3 memory. The test creates a list of addresses in an area of boot_mem. It then writes to all addresses with data value equal to address. Finally it reads back all locations checking that the read value is correct.
ddr32	Tests byte read and write accesses to DDR3 memory.
ddr33	Test back to back write-read accesses to DDR3 memory.
ethmac_mem	Tests wishbone access to the internal memory in the Ethernet MAC module.
ethmac_reg	Tests wishbone access to registers in the Ethernet MAC module.
ethmac_tx	Tests ethernet MAC frame transmit and receive functions and Ethmac DMA access to hiboot mem. Ethmac is put in loopback mode and a packet is transmitted and received.
firq	Executes 20 FIRQs at random times while executing a small loop of code. The interrupts are triggered using a ransom timer. Test checks the full set of FIRQ registers (r8 to r14) and will only pass if all interrupts are handled correctly.
flow_bug	The core was illegally skipping an instruction after a sequence of 3 conditional not-execute instructions and 1 conditional execute instruction.
hiboot_mem	Tests wishbone read and write access to hi (non-cachable) boot SRAM.
inflate_bug	A load store sequence was found to not execute correctly.

Name	Description
irq	Tests running a simple algorithm to add a bunch of numbers and check that the result is correct. This algorithm runs 80 times. During this, a whole bunch of IRQ interrupts are triggered using the random timer.
ldm1	Tests the standard form of ldm.
ldm2	Tests ldm where the user mode registers are loaded whilst in a privileged mode.
ldm3	Tests ldm where the status bits are also loaded.
ldm4	Tests the usage of ldm in User Mode where the status bits are loaded. The s bit should be ignored in User Mode.
ldr	Tests ldr and ldrb with all the different addressing modes.
ldr_atr_pc	Tests lrd and str of r15.
mla	Tests the mla (multiply and accumulate) instruction.
mlas_bug	Bug with Multiply Accumulate. The flags were gettting set 1 cycle early.
movs_bug	Tests a movs followed by a sequence of ldr and str instructions with different condition fields.
mul	Tests the mul (multiply) instruction.
sbc	Tests the 'subtract with carry' instruction by doing 3 64-bit subtractions.
stm1	Tests the normal operation of the stm instruction.
stm2	Test jumps into user mode, loads some values into registers r8 - r14, then jumps to FIRQ and saves the user mode registers to memory.
strb	Tests str and strb with different indexing modes.
sub	Tests sub and subs.
swi	Tests the software interrupt – swi.
swp_lock_bug	Bug broke an instruction read immediately after a swp instruction.
swp	Tests swp and swpb.
uart_reg	Tests wishbone read and write access to the Amber UART registers.
uart_rxint	Tests the UART receive interrupt function. Some text is sent from the test_uart to the uart and an interrupt generated.
uart_rx	Tests the UART receive function.
uart_tx	Uses the tb_uart in loopback mode to verify the transmitted data.
undefined_ins	Tests Undefined Instruction Interrupt. Fires a few unsupported floating point unit (FPU) instructions into the core. These cause undefined instruction interrupts when executed.

10.5 C Programs

In addition to the short assembly language tests, some longer programs written in C are included with the Amber system. These can be used to further test and verify the system, or as a basis to develop your own applications.

The source code for these programs is in $AMBER_BASE/sw.

10.5.1 Boot Loader

This is located in $AMBER_BASE/sw/boot-loader. It can be run in simulation as follows;

```
cd $AMBER_BASE/hw/sim
run boot-loader
```

The simulation output looks like the following;

```
# do run.do
# Test boot-loader, log file boot-loader.log
# Load boot memory from ../../sw/boot-loader/boot-loader.mem
# Read in 1868 lines
# Amber Boot Loader v20101201135034
# Commands
# l                           : Load elf file
# d <start address> <num bytes> : dump mem
# h                           : Print help message
# j                           : Execute loaded elf, jumping to 0x8000
# p <address>                 : print ascii mem until first 0
# r <address>                 : read mem
# s                           : Core status
# w <address> <value>         : write mem
# x                           : Load and execute elf in 1 step
#
#   Example
# > d 100 200
#   Dumps 512 bytes from 0x100
#
# r 0  0x00000035
# r 1  0x00001a72
# r 2  0x00000000
# r 3  0x00000002
# r 4  0x00000010
# r 5  0xdeadbeef
# r 6  0xdeadbeef
# r 7  0xdeadbeef
# r 8  0xdeadbeef
# r 9  0xdeadbeef
# r10  0xdeadbeef
# r11  0xdeadbeef
# r12  0x00001c86
# r13  0x60000654
# sp   0x07ffff88
# pc   0x60000650
#
# ------------------------------------------------------------------------
# Amber Core
#        > User         FIRQ         IRQ          SVC
# r0       0x00000001
# r1       0x00001b5c
# r2       0x00000000
# r3       0x00000000
# r4       0x00000010
# r5       0xdeadbeef
# r6       0xdeadbeef
# r7       0xdeadbeef
# r8       0xdeadbeef   0xdeadbeef
# r9       0xdeadbeef   0xdeadbeef
# r10      0x00000011   0xdeadbeef
# r11      0xf0000000   0xdeadbeef
# r12      0x00001c86   0xdeadbeef
# r13      0x07ffffc0   0xdeadbeef   0xdeadbeef   0xdeadbeef
# r14 (lr) 0x60000660   0xdeadbeef   0xdeadbeef   0xdeadbeef
# r15 (pc) 0x00001148
#
# Status Bits: N=0, Z=1, C=1, V=0, IRQ Mask 0, FIRQ Mask 0, Mode = User
# ------------------------------------------------------------------------
#
# ++++++++++++++++++++
# Passed boot-loader
# ++++++++++++++++++++
```

The boot loader is used to download longer applications onto the FPGA development board via the UART port and using Hyper Terminal on a host Windows PC.

10.5.2 Hello World

This is located in $AMBER_BASE/sw/hello-world. It can be run in simulation as follows;

```
cd $AMBER_BASE/hw/sim
run hello-world
```

This is a very simple example of a stand alone C program. The printf function it uses

is contained in $AMBER_BASE/sw/mini-libc, so that it can run on an FPGA without access to a real libc library file.

10.5.3 Ethmac Test

This is located in $AMBER_BASE/sw/ethmac-test. This is designed to be loaded and run by the boot loader. It is a simple test of the Ethernet MAC module in loopback mode. It can be run in simulation as follows;

```
cd $AMBER_BASE/hw/sim
run ethmac-test
```

11 FPGA Synthesis

A script is provides that performs synthesis of the system to a Xilinx Spartan-6 or Virtex-6 FPGA. To use this script you must have Xilinx ISE installed. I have tested it with ISE v11.5.

To use the script,

```
cd $AMBER_BASE/hw/fpga/bin
make new
```

The script performs the following steps

1. Compiles the boot loader program in $AMBER_BASE/sw/boot-loader, to make sure the latest version goes into the FPGA boot SRAM ram blocks.
2. Runs xst to synthesize the Verilog file $AMBER_BASE/hw/vlog/system/system.v and everything inside it.
3. Runs ngbbuild to create the map-compatible top-level netlist.
4. Runs map to do placement.
5. Runs par to do routing.
6. Runs bitgen to create an FPGA bitfile in the bitfile directory.
7. Runs trce to do timing analysis on the finished FPGA.

The Spartan-6 FPGA is the default. To compile for the Virtex-6 FPGA, set VIRTEX6=1 on the command line, e.g.

```
cd $AMBER_BASE/hw/fpga/bin
make new VIRTEX6=1
```

If the par step fails (timing or area constrains not met), you can rerun map and par with a different seed. Simply call the Makefile again without the new switch. The Makefile will automatically increment the seed, e.g.

```
cd $AMBER_BASE/hw/fpga/bin
make
```

The system clock speed is configured within the FPGA Makefile, $AMBER_BASE/hw/fpga/bin/Makefile. To change it, change the value of AMBER_CLK_DIVIDER in that file. The system clock frequency is equal to the PLL's VCO clock frequency divided by AMBER_CLK_DIVIDER. By default it is set to 40MHz for Spartan-6 and 80MHz for Virtex-6.

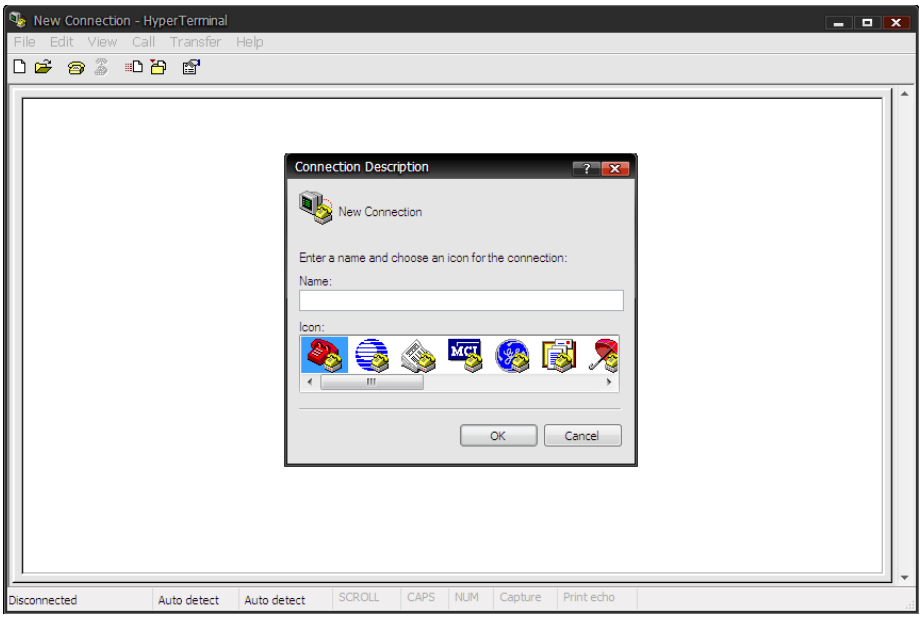
12 Using Boot-Loader

If you have a development board with a UART connection to a PC you can use boot-loader to download and run applications on the board. I have tested this with the Xilinx SP605 development board. It provides a UART connection via a USB port on the board.

12.1 Configure HyperTerminal

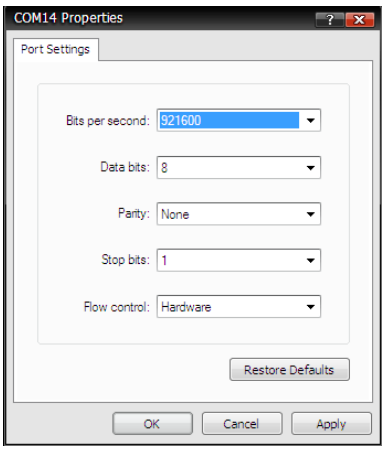
Run HyperTerminal on the PC. This is a free application supplied with Windows. In Windows XP it is available on the Start Menu under All Programs -> Accessories -> Communications -> HyperTerminal

When HyperTerminal starts it brings up the new connection dialogue. The first screen of this dialogue asks you to name the connection. Name it anything you like.



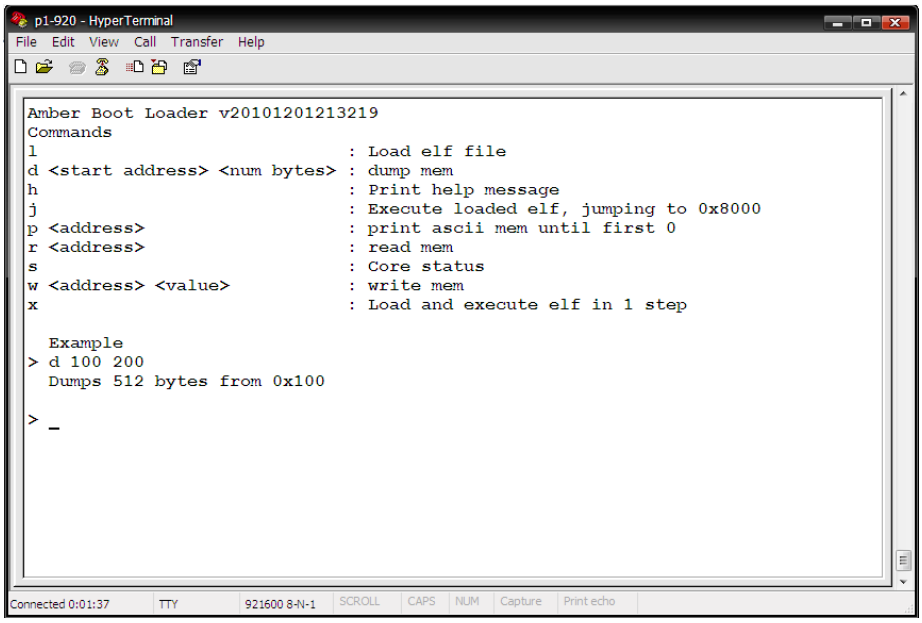
On the next screen select the com port. This will depend on your PC. For me the correct port is COM14. Check the documentation for your FPGA board.

The final screen sets the connection speed and type. Select 921600 bits per second, 8 data bits, no parity bits, 1 stop bit and hardware flow control. This is the default speed of the UART and is configured within the Verilog code. It can be changed prior to FPGA synthesis.



12.2 Configure the FPGA

Load the bitfile into the FPGA on the development board. This is usually done using Xilinx iMPACT. Once the FPGA is configured the boot loader will print some messages via the UART interface onto the HyperTerminal screen. You should see the following;



You can now load and run applications using the boot loader. To load the ethmac-test application, type 'x' and hit return. This puts the boot loader in a loop waiting to receive a file. Next select the Transfer-> Send File menu item on HyperTerminal. Then click on browse and select the ethmac-test.elf file in $AMBER_BASE/sw/ethmac-test. The elf file in generated when you run make in that directory. The file downloads to the board and runs.

13 License

All source code provided in the Amber package is release under the following license terms;

```
Copyright (C) 2010 Authors and OPENCORES.ORG

This source file may be used and distributed without
restriction provided that this copyright statement is not
removed from the file and that any derivative work contains
the original copyright notice and the associated disclaimer.

This source file is free software; you can redistribute it
and/or modify it under the terms of the GNU Lesser General
Public License as published by the Free Software Foundation;
either version 2.1 of the License, or (at your option) any
later version.

This source is distributed in the hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the implied
warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU Lesser General Public License for more
details.

You should have received a copy of the GNU Lesser General
Public License along with this source; if not, download it
from http://www.opencores.org/lgpl.shtml

 Author(s):
     - Conor Santifort, csantifort.amber@gmail.com
```