

An Implementation of a Renesas H8/300 Microprocessor with a Cycle-Level Timing Extension

Chen-Chun Huang, Javier Coca, Yashket Gupta, and Stephen A. Edwards
Columbia University
{ch2377,jc2658,yg2153}@columbia.edu, sedwards@cs.columbia.edu

Abstract

We describe an implementation of the Renesas H8/300 16-bit processor in VHDL suitable for synthesis on an FPGA. We extended the ISA slightly to accommodate cycle-accurate timers accessible from the instruction set, designed to provide more precise real-time control.

We describe the architecture of our implementation in detail, describe our testing strategy, and finally show how to build a cross compilation toolchain under Linux.

Contents

1	Introduction	2	4	Testing	9
2	The Datapath	2	4.1	Hardware implementation for testing	9
2.1	Datapath components	2	4.1.1	LED Display Module	9
2.1.1	Accumulator (acc)	2	4.1.2	UAT Module	9
2.1.2	Arithmetic Logic Unit (ALU)	4	4.2	Test programs	10
2.1.3	Bridge (bdg)	4	4.2.1	Bit Counter	10
2.1.4	Concatenator (cnct)	4	4.2.2	Hello Columbia	10
2.1.5	Condition Code Register (ccr)	4	5	Building a cross compiler	10
2.1.6	Instruction Register (IR)	6	6	Conclusions	10
2.1.7	Memory Address (MA)	6	7	Appendix	11
2.1.8	Memory Data (MD)	6	7.1	Signal list for VHDL modules	11
2.1.9	Memory Interface (mem_interface)	6	7.2	ASM chart for the controller	14
2.1.10	Multiplexers RD and RS	7			
2.1.11	Program Counter (PC)	7			
2.1.12	General Purpose Registers (R#)	7			
2.1.13	Temporary Register (tmp)	7			
2.1.14	Timers	7			
2.2	Addressing modes	8			
2.2.1	Register Direct, Rn	8			
2.2.2	Register Indirect, @Rn	8			
2.2.3	Register indirect with displacement	8			
2.2.4	Register indirect with post-inc/pre-dec	8			
2.2.5	Absolute address, @aa:8 or @aa:16	8			
2.2.6	Immediate, #xx:8 or #xx:16	8			
2.2.7	PC-Relative, @(d:8, PC)	8			
2.2.8	Memory Indirect, @@aa:8	8			
2.3	Memory	8			
3	Synthesis on the FPGA	9			

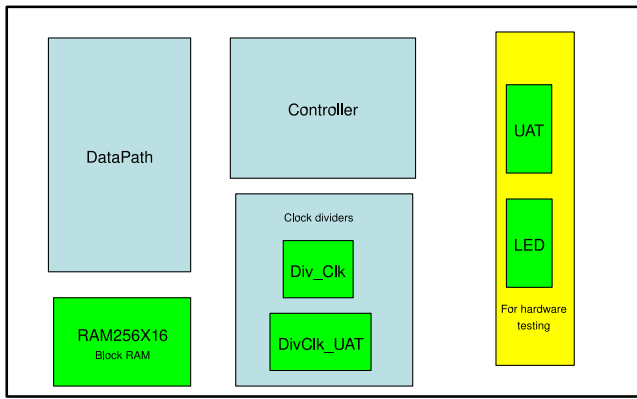


Figure 1: Block diagram

1 Introduction

We designed this sixteen-bit processor based on the Renesas¹ H8/300 to enable cycle-accurate real-time software. We implemented the H8 instruction set and added high-precision timers and added instructions to access them. Specifically, the new instruction waits for a timer to expire, reloads it, and terminates synchronously. This, coupled with the simple architecture of the processor (e.g., no pipeline, no cache) makes execution time predictable. We implemented this processor on an FPGA.

Figure 1 shows the block diagram of our implementation. The main components are the usual datapath and controller. We include an on-chip memory, two clock dividers that reduce the 50 MHz main clock to 25 MHz for the processor core and 9.6 kHz for the “UAT,” a RS-232-style asynchronous serial transmitter we use for debugging. We also include a memory-mapped controller for some LEDs, also for debugging.

2 The Datapath

Figure 2 illustrates the datapath of our processor, which corresponds to the `h8_datapath.vhd` file. The current program counter flows from the PC module, an instance of register16, which is reset to 0, to the `mem_interface` module.

The `mem_interface` receives the 16-bit instruction and sends it to the multiplexers. The instruction from the muxes goes to the ALU (for immediate data) and to the Bridge. The Bridge sends the instruction to the instruction register (IR), which holds it for the controller until the next instruction is fetched. The IR sends the current instruction to the controller, which decodes and executes the instruction.

Data flows in various ways depending on the instruction. Arithmetic and logical operations (add, sub, and, or, etc.) go through the arithmetic logic unit (ALU). The accumulator (ACC) handles shift operations (shift, rotate, etc.). Data transfer, branch, and system control instructions pass through the bridge to various other components.

¹Renesas bought the semiconductor operations of Hitachi, the original developer of the H8, in 2003.

Instructions that use the ALU or ACC all follow a general pattern:

1. Source operands, which can be any general purpose register or the instruction register, flow through a multiplexer.
2. Data from the multiplexers flow to the ALU and ACC.
3. The output from the ALU and ACC flow to a destination register or to the memory data register (MD), which will write it out.

Data transfer, branch, and system control instructions are less regular. The branching or the destination address is stored either in the memory or the general purpose register. For example, for a MOV.B instruction instruction

1. The destination address from a register goes to a multiplexer.
2. The multiplexer output flows to the memory address (MA) multiplexer.
3. The address from the MA mux flows to the MA register.
4. Data from memory flows through the `mem_interface` (MI)
5. Data flows from MI to a multiplexer.
6. From the multiplexer, data is sent to the memory data (MD) register.
7. Data from the MD register is sent to the concatenator (`cnct`) register.
8. Data from Register low/high to a multiplexer.
9. Data from multiplexer to ALU.
10. Data from ALU to `cnct` register.
11. Data from `cnct` to MD register.
12. Data from MD to memory.

2.1 Datapath components

2.1.1 Accumulator (*acc*)

The accumulator (Figure 3) takes the eight-bit `acc_in` input and performs one of the operations in Table 1 based on the `acc_op` input. The `mux_acc` mux selects the input based on `acc_sel`; Table 2 lists the codes, which select among the general-purpose registers. The output of the accumulator is sent to the registers. The accumulator result is also used to update the CCR.

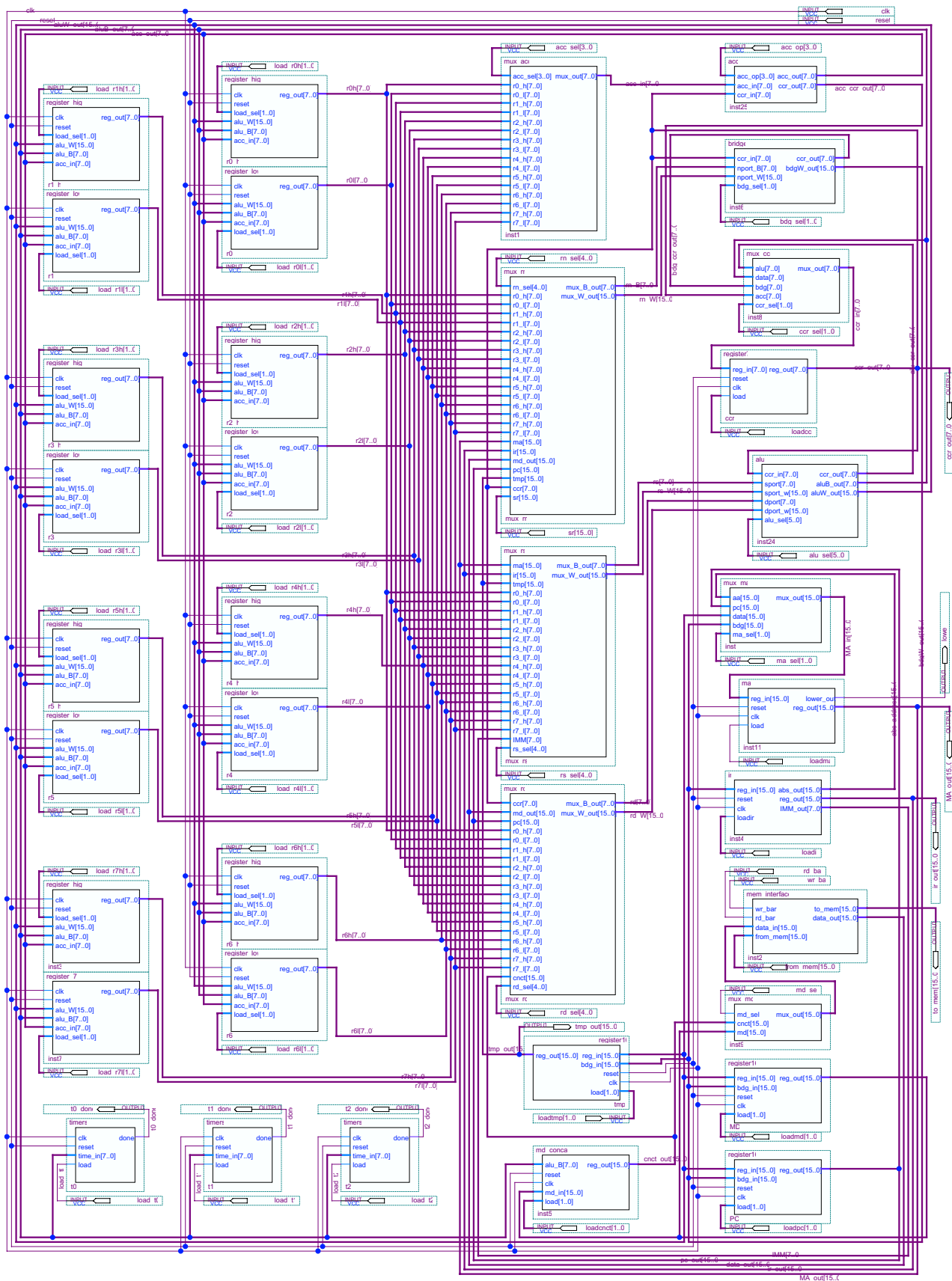


Figure 2: The data path

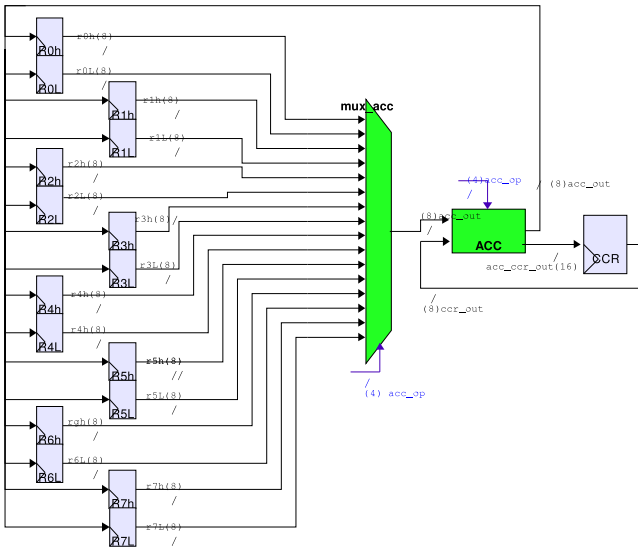


Figure 3: Accumulator schematic

ACCumulator	acc_op	
	"000"	Rotate Left
	"001"	Rotate Right
	"010"	Rot Lft Carry
	"011"	Rot Rht Carry
	"100"	Shift Left
	"101"	Shift Right
	"110"	Shift Log Left

Table 1: Accumulator operations

Mux Acc	acc_sel	
Byte Output <=	Reg0_high	"0000"
	Reg0_low	"0001"
	Reg1_high	"0010"
	Reg1_low	"0011"
	Reg2_high	"0100"
	Reg2_low	"0101"
	Reg3_high	"0110"
	Reg3_low	"0111"
	Reg4_high	"1000"
	Reg4_low	"1001"
	Reg5_high	"1010"
	Reg5_low	"1011"
	Reg6_high	"1100"
Reg6_low	"1101"	

Table 2: Accumulator input selection codes

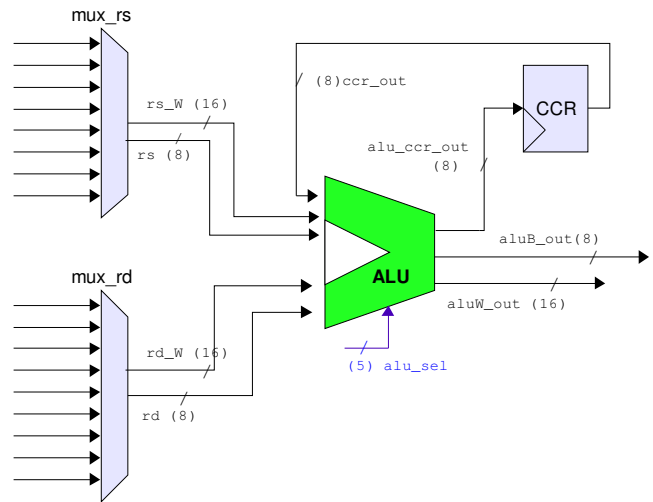


Figure 4: The arithmetic logic unit

2.1.2 Arithmetic Logic Unit (ALU)

The arithmetic logic unit (Figure 4) supports all the 16-bit, 8-bit, and single bit mathematical operations in the processor's ISA. The ALU executes arithmetic operations such as addition, subtraction, and multiplication and logic operations such as AND, OR, and XOR. It also implements the bit modification instructions such as bit-clear and bit-set. Table 3 lists all the operation codes, which come from the controller through the alu_sel signal. The mux_rs and mux_rd multiplexers select the inputs for the ALU. The ALU's output feed almost all the registers through the aluW_out bus for 16-bit results and through the aluB_out bus for 8-bit results. The ALU also modifies the overflow, carry out, zero and negative flags of the CCR. The ALU is also used in a pass-through mode for register-to-register transfers.

2.1.3 Bridge (bdg)

The bridge (Figure 5) routes data from mux_rn to the 16-bit special-purpose registers. It can perform several modifications to the inputs, listed in Table 4, but is most commonly used to pass data from general-purpose registers to special-purpose registers. The mux_bdg multiplexer selects the bridge input according to the codes in Table 4. The inputs of the multiplexer are either 8-bit or 16-bit data. The mux concatenates pairs of registers to form 16-bit output.

2.1.4 Concatenator (cnct)

The concatenator (Figure 6) takes the eight-bit output from the ALU and replaces either the low or high byte from the MD register to form a 16-bit result according to the codes in Table 5. The result is sent to memory.

2.1.5 Condition Code Register (ccr)

The 8-bit condition code register (Figure 7) holds flags such as overflow, carry, zero and negative. It is updated by the ALU, ACC, and Bridge, according to the codes in Table 6. It can also be loaded directly from data from the ALU, i.e., rather than from the results of a computation.

alu_sel		alu_sel	
Byte Output <= Zeros	"000000"	0	reg Bit set "101110"
Addition	"000001"	1	reg Bit invert "101111"
Subtract	"000010"	2	reg Bit clear "110000"
Add w/crry	"000011"	3	reg Bit test "110001"
Sub w/crry	"000100"	4	16 bits Bit set "110010"
Increment	"000101"	5	16 bits Bit invert "110011"
Decrement	"000110"	6	16 bits Bit clear "110100"
Daa	"000111"	7	16 bits B inv Str "110101"
Das	"001000"	8	16 bits Bit store "110110"
Compare	"001001"	9	16 bits/reg Bit set "110111"
Twos Cmpl	"001010"	10	16 bits/reg Bit invert "111000"
AND	"001011"	11	16 bits/reg Bit clear "111001"
OR	"001100"	12	
XOR	"001101"	13	
NOT	"001110"	14	Word Output Zeros "010000"
ccr_in	"001111"	15	Addition "010001"
sport Byt	"011100"	28	Subtract "010010"
dport Byt	"011110"	30	Increment "010011"
Bit set	"100000"	32	Decrement "010100"
Bit invert	"100001"	33	Inc by 2 "010101"
Bit clear	"100010"	34	Dec by 2 "010110"
Bit test	"100011"	35	Compare "010111"
Bit inv or	"100100"	36	Multipl "011000"
Bit or	"100101"	37	sport W "011101"
B inv xor	"100110"	38	dport W "011111"
Bit xor	"100111"	39	
B inv and	"101000"	40	

Table 3: ALU operations

rn_sel		rn_sel	
Byte Output	Reg0_high "00000"	0	
	Reg0_low "00001"	1	
	Reg1_high "00010"	2	
	Reg1_low "00011"	3	
	Reg2_high "00100"	4	
	Reg2_low "00101"	5	
	Reg3_high "00110"	6	
	Reg3_low "00111"	7	
	Reg4_high "01000"	8	
	Reg4_low "01001"	9	
	Reg5_high "01010"	10	
	Reg5_low "01011"	11	
	Reg6_high "01100"	12	
	Reg6_low "01101"	13	
	Reg7_high "01110"	14	
	Reg7_low "01111"	15	
	ccr	"11110"	30
Word Output	Reg 0	"10000"	16
	Reg 1	"10001"	17
	Reg 2	"10010"	18
	Reg 3	"10011"	19
	Reg 4	"10100"	20
	Reg 5	"10101"	21
	Reg 6	"10110"	22
	Reg 7	"10111"	23
	md	"11000"	24
	ma	"11001"	25
	bc	"11010"	26

Table 4: Bridge operations

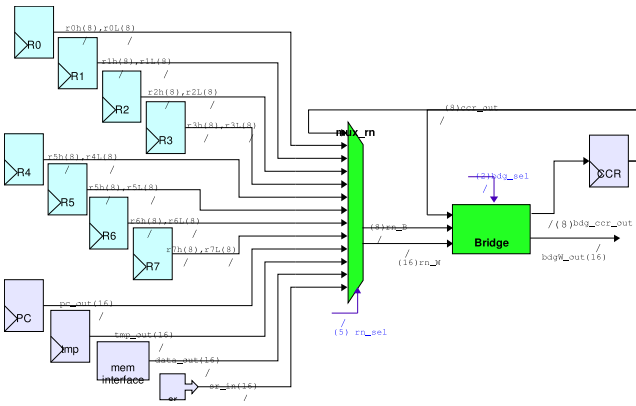


Figure 5: The bridge

Concat		loadcnct	
cnct_out =>	aluB & MD(7 to 0)	"10"	
	MD(15 to 8) & aluB	"11"	

Table 5: Concatenator operations

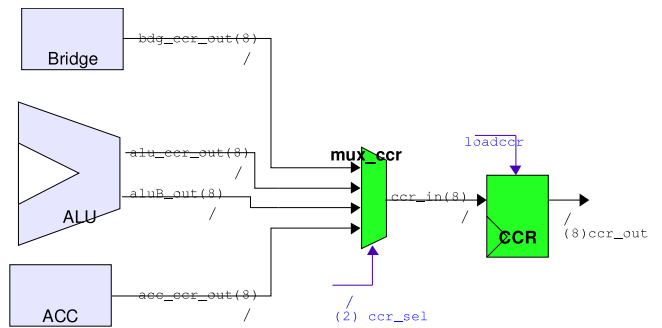


Figure 7: Condition code register schematic

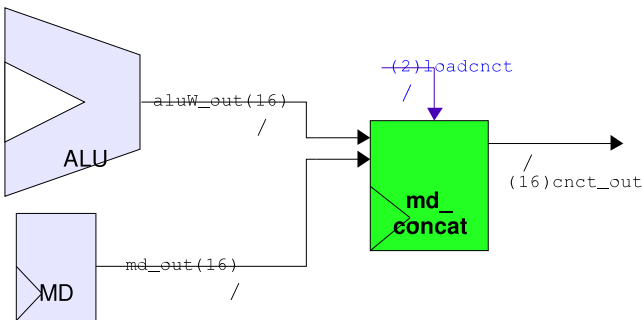


Figure 6: Concatenator schematic

Mux CCR		ccr_sel	
reg_in =>	alu	"00"	
	bdg	"01"	
	acc	"10"	
	data	"11"	

Table 6: Condition code register operations

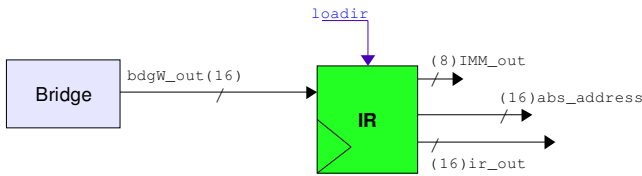


Figure 8: IR pin configuration

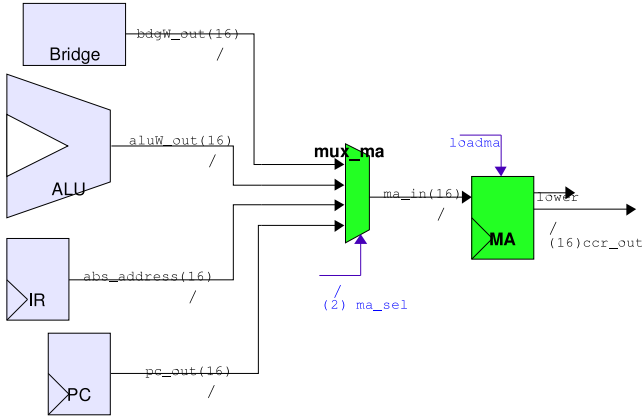


Figure 9: Memory address schematic

2.1.6 Instruction Register (IR)

The 16-bit instruction register (Figure 8) holds the instruction currently being executed and outputs three copies of it:

1. The high byte (bits 15 to 8)
2. The low byte (bits 7 to 0) used for immediate data
3. Absolute address, formed by zero-extending the low byte

2.1.7 Memory Address (MA)

Memory address (Figure 9) is a 16-bit register that is used to supply the read or write address to memory. A multiplexer on the input helps to select among the many addressing modes in the H8. Table 7 lists codes for the four different sources. As we explain at length in the memory section, the least significant bit of the memory address register is always 0.

mux MA		ma_sel	
Mux W Out <=	pc	"00"	
	abs addr	"01"	
	cnct	"10"	
	data	"11"	

Table 7: Memory address operations

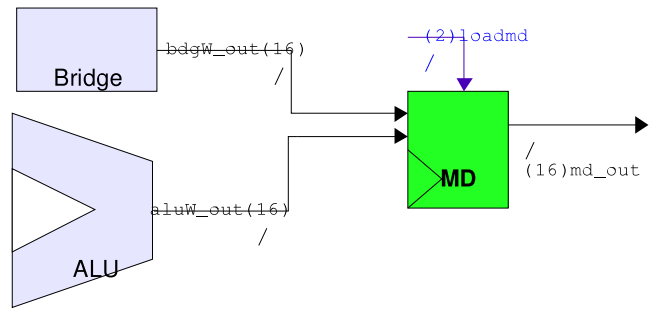


Figure 10: MD pin configuration

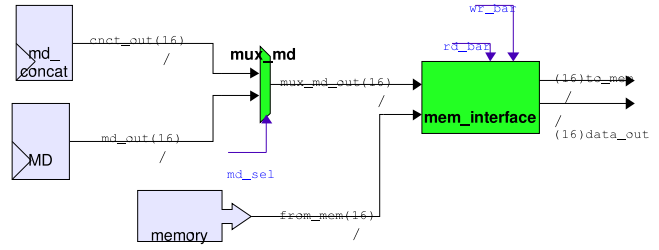


Figure 11: The memory interface unit

2.1.8 Memory Data (MD)

Memory data (Figure 10) is a 16-bit register that holds data to be written or read from memory. It has two inputs selected by the loadmd signal: the Bridge and the ALU. We implemented it with our general-purpose register16 component.

2.1.9 Memory Interface (mem_interface)

The memory interface unit (Figure 11) routes data to and from memory. It operates in three ways:

1. For a write to memory, the data comes from either MD or the Concatenator. The mux_md signal selects the source.
2. For a read from memory, data from memory is routed from the from_mem signal out data_out.
3. When data from MD or md_concat is to be transferred to the data path, the memory interface routes data from mux_md out data_out. Like a memory write, mux_md selects which source will be transferred.

mem interface		write_bar read_bar		mux_md	
data_out <=	mem	w=1	r=0	mux_out <=	md
	data_in	w=1	r=1		cnct
mem <=	data_in	w=0	r=1		
	zzzzz	w=1	r=0		

Table 8: The memory interface unit operations

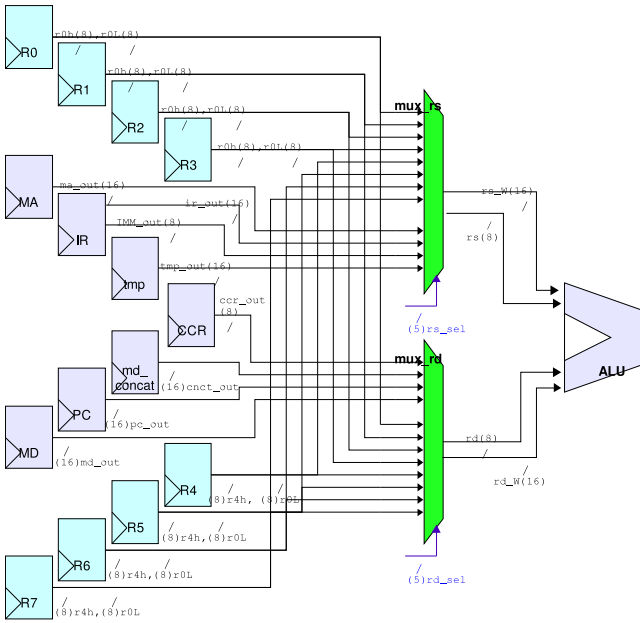


Figure 12: Multiplexer RS and RD schematic

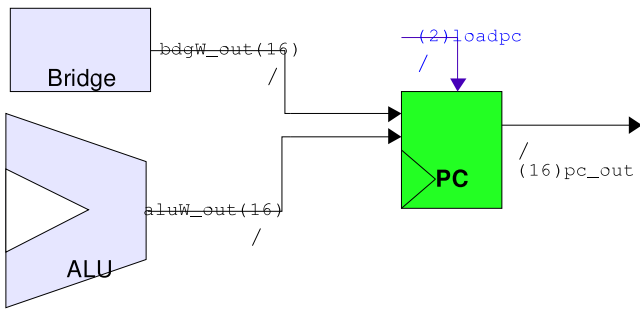


Figure 13: Program counter schematic

2.1.10 Multiplexers RD and RS (mux_rd, mux_rs)

These two multiplexers select the two inputs for the ALU. Figure 12 shows the connections; Table 9 lists the meaning of the control signals. The RS and RD names are taken from the H8/300 programming manual. The outputs are either 8- or 16-bit data—16-bit data comes from concatenating 8-bit register pairs. Note that for simplicity, only one connection from each register is shown in Figure 12.

2.1.11 Program Counter (PC)

The program counter (Figure 13) serves the usual purpose of holding the address of the next instruction. It is implemented using a generic 16-bit loadable register (register 16). The two inputs to this register are the ALU and the Bridge.

2.1.12 General Purpose Registers (R#)

The general purpose registers (Figure 14) are sixteen 8-bit registers grouped as eight 16-bit registers (ROL, ROH, R1L, ..., R8L, R8H). The load_reg signal controls from where each register is loaded according to the codes in Table 10.

mux_rd			mux_rs			
	rd_sel			rs_sel		
Byte Output	Reg0_high	"00000"	0	Reg0_high	"00000"	
	Reg0_low	"00001"	1	Reg0_low	"00001"	
	Reg1_high	"00010"	2	Reg1_high	"00010"	
	Reg1_low	"00011"	3	Reg1_low	"00011"	
	Reg2_high	"00100"	4	Reg2_high	"00100"	
	Reg2_low	"00101"	5	Reg2_low	"00101"	
	Reg3_high	"00110"	6	Reg3_high	"00110"	
	Reg3_low	"00111"	7	Reg3_low	"00111"	
	Reg4_high	"01000"	8	Reg4_high	"01000"	
	Reg4_low	"01001"	9	Reg4_low	"01001"	
	Reg5_high	"01010"	10	Reg5_high	"01010"	
	Reg5_low	"01011"	11	Reg5_low	"01011"	
	Reg6_high	"01100"	12	Reg6_high	"01100"	
	Reg6_low	"01101"	13	Reg6_low	"01101"	
	Reg7_high	"01110"	14	Reg7_high	"01110"	
	Reg7_low	"01111"	15	Reg7_low	"01111"	
Word Output	ccr	"11011"	27	tmp_IMM	"11010"	
	md High By	"11100"	28	IMM	"11011"	
	md Low By	"11101"	29			
	cnct_IMM	"11110"	30			
				Word Output	Reg 0	"10000"
					Reg 1	"10001"
					Reg 2	"10010"
					Reg 3	"10011"
					Reg 4	"10100"
					Reg 5	"10101"
				Reg 6	"10110"	
				Reg 7	"10111"	
				ma	"11000"	
				ir	"11001"	
				md	"11000"	
				"00"& IMM	"11100"	

Table 9: Multiplexer RS and RD operations

As the H8/300 programming manual suggests, register 7 is typically used as the stack pointer.

2.1.13 Temporary Register (tmp)

The temporary register (Figure 15) can save temporary results from either the ALU or the Bridge. It is implemented using a generic register 16.

2.1.14 Timers

The three timers (Figure 16) are 8-bit registers that count down and are reloaded when they reach zero to implement our cycle-accurate timing extension.

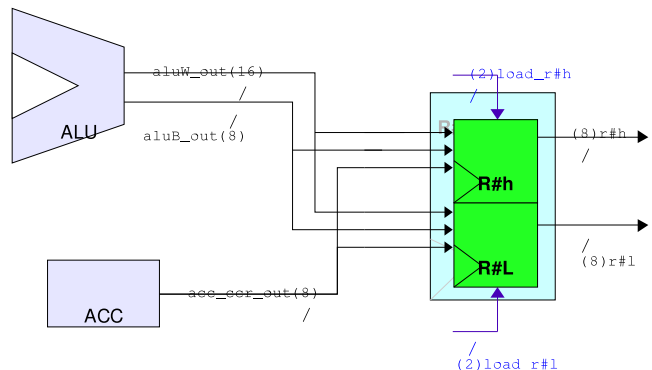


Figure 14: General purpose register schematic

Registers High / Low	load_reg	
reg_in =>	alu BYTE	"01"
	alu WORD	"10"
	acc	"11"

Table 10: General purpose register operations

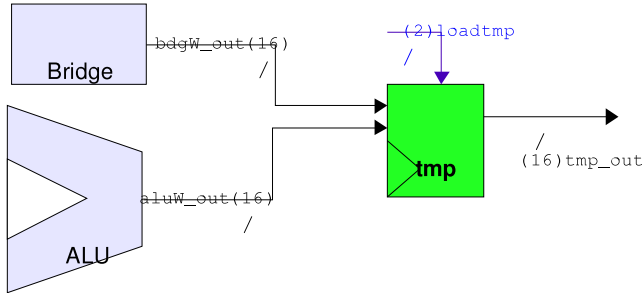


Figure 15: Temporary register schematic

2.2 Addressing modes

Our implementation supports the eight addressing modes of the H8/300, but their details differ slightly because we employ a 16-bit data bus instead of the 8-bit bus on an actual H8.

2.2.1 Register Direct, Rn

The register field of the instruction specifies an 8- or 16-bit general register containing the operand. The mux_select signal comes from the instruction itself to select one or a pair of the general-purpose registers.

2.2.2 Register Indirect, $@Rn$

The register field of the instruction specifies from which 16-bit general register the address of the operand is to be taken. This address is passed to the MA. Data is fetched from the memory and sent to the ALU.

2.2.3 Register indirect with displacement, $@(d:16, Rn)$

This mode is like register indirect but adds the contents of a second instruction word to the contents of a 16-bit register to form the address.

The controller first fetches the offset from the word after the instruction by moving the contents of the PC to the MA. The offset is fetched into the mem_interface, which forwards it to the MD.

Once the offset is in the MD, the ALU adds the two 16-bit operands (one general-purpose register and the MD) to generate the operand address, which is sent again to the MA to fetch the operand.

2.2.4 Register indirect with post-increment or pre-decrement, $@Rn+$ or $@-Rn$

Like register indirect, the register field of the instruction specifies which 16-bit general register holds the address of

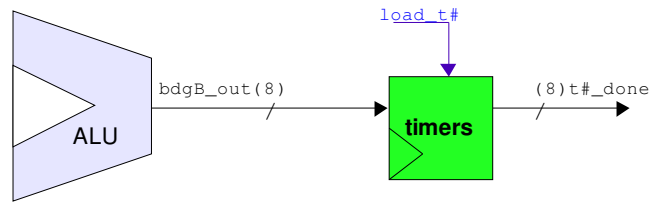


Figure 16: Timer schematic

the operand. The difference is that the address is either incremented after or decremented before being returned to the source register. Such modes conveniently implement a stack.

2.2.5 Absolute address, $@aa:8$ or $@aa:16$

The instruction specifies the absolute address of the operand in memory. The $@aa:8$ mode zero-extends an 8-bit absolute address to access memory in the range H'0000 to H'00FF. Eight-bit absolute addresses are in the instruction itself, so the value is taken from the IR and passed through the mux_ma.

A sixteen-bit absolute address appears after the instruction in memory, so the PC must be loaded into the MA, the address fetched from the instruction stream into the mem_interface, and finally the result passed back to the MA.

2.2.6 Immediate, $\#xx:8$ or $\#xx:16$

Eight-bit immediate data is in the instruction itself, so it is taken from the IR. Sixteen-bit immediate data comes from the next word in the instruction stream, so the PC is loaded into the MA, the word fetched, and this becomes the immediate data.

2.2.7 PC-Relative, $@(d:8, PC)$

This mode is used to generate branch addresses. An 8-bit value in the lower byte of the instruction is sign-extended and added to the new program counter (i.e., beyond the current instruction). The ALU adds these values and passes them back to the PC.

2.2.8 Memory Indirect, $@@aa:8$

The lower byte of the instruction code specifies an 8-bit absolute address from H'0000 to H'00FF (0 to 255). A word is fetched from this address and becomes the target of the branch (this mode is only used by the JSR or JMP instructions).

2.3 Memory

We implemented a 16-bit wide memory system. Since the H8 has byte-addressable memory, we implemented a mask where the H8 sees the 16-bit RAM as individual bytes.

To implement such a behavior, the LSB of the memory address inside the data path is used to designate what part of the word the upper 8-bits or the lower 8-bits is been accessed. When the LSB is 1, the lower part (bit 7 to bit 0) of the word is accessed, with a 0 the higher part (bit 15 to bit 8) of the word is accessed. Because of this lower bit implementation the memory can only be accesses in the even

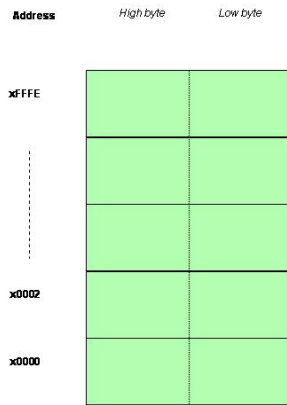


Figure 17: The concatenator

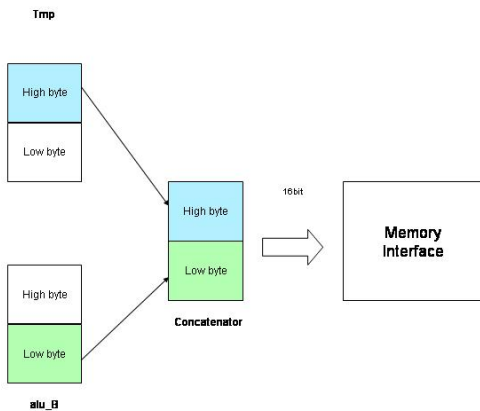


Figure 18: The memory interface

addresses reducing the available space for storage by half. Given the fact that all the accesses to memory must be done in words (16-bits) we created a component that concatenates two 8-bit data. This component is called the concatenator (CNCT, Figure 17).

The CNCT is responsible for creating proper data to be stored in memory and has a direct connection to the memory interface. Depending on the part of the 16-bit data (high byte or low byte) being modified, the CNCT combines the modified byte with the unmodified byte. If the lower part of the word (bit 7 to bit 0) is to be modified, we save the high part of the word (bit 15 to bit 8) in a temporary register (tmp). When the low byte is ready to be written back, it will be concatenated with the unchanged high part in the CNCT.

The memory is connected to the Datapath through a component called the memory interface (MI, Figure 18). This is a bidirectional component that routes data to and from memory. When the memory interface is receiving data from the

memory, the data_out output is used for the data to be connected with the all the other components through the muxes. For a write, the data_in input is sent to the memory output. The data that is feed into data_in can be from the registers MD or the CNCT.

3 Synthesis on the FPGA

Our hardware platform, an XSB-300E board from XESS corporation, has a single Xilinx Spartan IIE FPGA containing 300K system gates. This FPGA has sixteen 4K-bit block RAMs. We only used one of the sixteen RAMs for testing. To configure a block RAM to work in “write-first” mode, we followed the specific coding style from the Xilinx manual.

Coding The combinational logic required another trick. Our design consists of many case statements. If we did not carefully include every case and a default, the synthesis tool would generate priority decoders and latches. Careful encoding of multiplexer controls and parallel condition checks also reduce any extraneous logic.

According to synthesis report generated by Xilinx ISE 6.3, the longest paths passed through register files and multiplexers with the highest fan-outs. Much could be done to improve the performance of our implementation.

4 Testing

We did extensive simulation to check each instruction is consistent with the specification in the H8 programmer’s manual. After testing each instruction in isolation, we started testing programs.

Initial testing was done with simulation; for testing on the FPGA, we implemented an LED module (Figure 19) to display the data bus and memory address. This was cumbersome and time consuming.

Finally, we implemented a serial transmitter, the UAT, which could be made to send program output through a serial port under program control.

4.1 Hardware implementation for testing

A LED module and a UAT were implemented to provide debugging capabilities for our processor.

4.1.1 LED Display Module

We used the LED display module (Figure 20) for hardware testing. This module controls three LED displays on the XSB board. Memory input/output busses, address, and control states are connected to this module. The data trigger is set in the controller and it is easy to modify trigger conditions. Two seven-segment LEDs display the memory address or data. Their role is set by the switches on the XSB board.

4.1.2 UAT Module

UAT (Universal asynchronous transmitter), unlike UART, it only transmits data. A nine-bit shift register which operates at a clock frequency of 9600Hz provided by Clock Divider module fetches parallel data from memory data bus and outputs to serial port when controller issues the trigger with particular memory address.

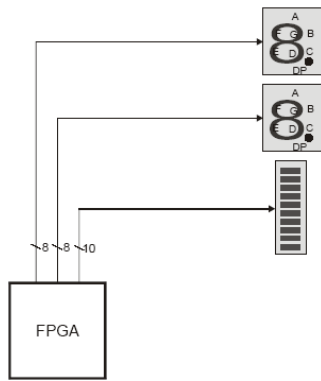


Figure 19: LEDs on the XSB board

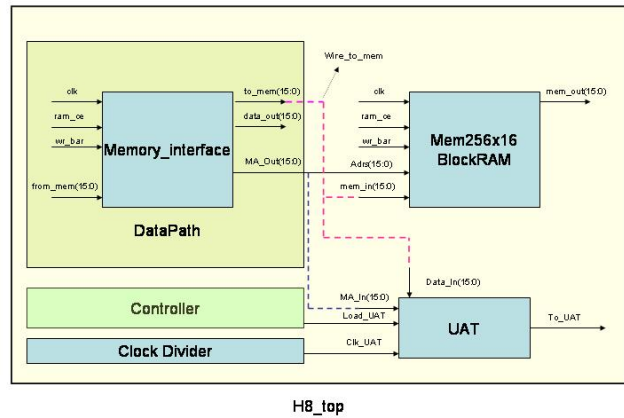


Figure 21: Block diagram of UAT

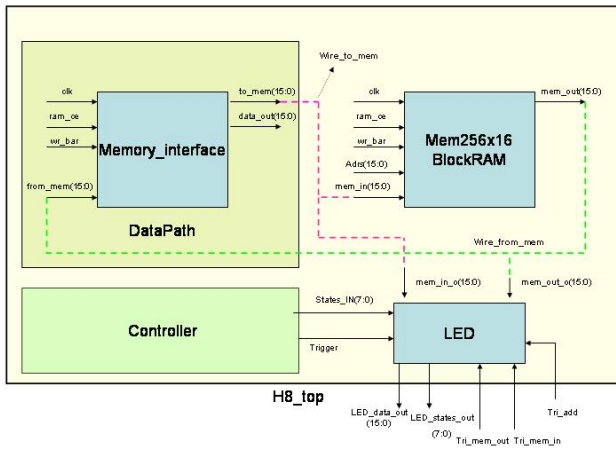


Figure 20: Block diagram of LED display module

4.2 Test programs

4.2.1 Bit Counter

This counts the number of 1s and 0s in data. It is easier to test the result in simulation; on the FPGA we had the program send its output out the serial port.

4.2.2 Hello Columbia

This test uses a number of arithmetic operations to set the ASCII equivalent of “Hello Columbia” in the registers. Once the values are set, the data is sent from the registers out the serial port to be displayed. If successful, we observe “Hello Columbia” on the terminal.

5 Building a cross compiler

We built a cross-compiler for the H8/300 using gcc. The main challenge was finding appropriate versions of the GNU binutils, gcc, and newlib packages that would handle the H8/300 environment. The steps to do this are

1. Create two directories in your home: cross-compiler and h8-compiler.

2. Download the following files to the cross-compiler directory:

```
binutils-2.17_sh_h8_mi6c_v0603.tar.bz2
gcc-4.2-20060812_sh_h8_m16c_v0603.tar.bz2
newlib-1.14.0_sh_h8_m16c_v0603.tar.bz2
```

3. Build binutils

```
cd cross-compiler
tar xjf binutils-2.17_sh_h8_mi6c_v0603.tar.bz2
# Unpacking creates a binutils-2.17/ directory
mkdir build-binutils
cd build-binutils
./binutils-2.17/configure \
  --prefix=/home/user/../../h8-compiler \
  --target=h8300-elf
gmake CFLAGS="-O2 -fomit-frame-pointer" all
gmake install
export PATH=$PATH:~/h8-compiler/bin
```

4. Build gcc with newlib

```
tar xjf gcc-4.2-20060812_sh_h8_m16c_v0603.tar.bz2
# Unpacking creates a gcc-4.2-20060812/ directory
tar xzf newlib-1.14.0_sh_h8_m16c_v0603.tar.bz2
# Unpacking creates a newlib-1.14.0/ directory
cd gcc-4.2-20060812
ln -s ~/cross-compiler/newlib-1.14.0/newlib .
mkdir build-gcc
cd build-gcc
~/cross-compiler/gcc-4.2/configure \
  --prefix=/home/user/../../h8-compiler \
  --target=h8300-elf \
  --enable-languages=c \
  --with-newlib
gmake CFLAGS="-O2 -fomit-frame-pointer" all
gmake install
```

The above steps will create h8 tool chain for the elf format in the h8-compiler directory.

6 Conclusions

We tested assembly programs for our H8/300. We synthesized the design on the Xilinx Spartan IIE FPGA, we were able to run the design at 25 MHz, the memory was a 256x16 bit BRAM (we used one of sixteen such RAMs

on the FPGA). We implemented a serial transmitter that displayed results on a 9600 baud terminal.

We were not able to test C programs on the H8 because we were not able to merge the ELF files generated by our compilation chain. This is the obvious next step.

7 Appendix

7.1 Signal list for VHDL modules

H8_Top.vhd		
Signal	Type	Description
clk	Input	System clock (50MHz)
reset	Input	System reset
LED_in	Input	Manual LED Trigger
LED_out	Input	Manual LED Trigger
LED_add	Input	Manual LED Trigger
sr (15:0)	Input	TBD
Switches (3:0)	Input	TBD
UAT_out	Output	To serial port
data_out (15:0)	Output	Data to LEDs
state_out (7:0)	Output	Data to LEDs
DClk_UAT_OUT	Output	Test output

ram256x16.vhd		
Signal	Type	Description
clk	Input	Divided clock (25MHz)
wr_bar	Input	Memory write
ram_ce	Input	Memory chip enable
adrs (15:0)	Input	Memory address
mem_in (15:0)	Input	Data of memory write
mem_out (15:0)	Output	Data of memory read

register7.vhd		
Signal	Type	Description
clk	Input	Divided clock (25MHz)
reset	Input	System reset
Load	Input	Register load selection
reg_in (7:0)	Input	Data from CCR
reg_out (7:0)	Output	Register output

register16.vhd (TMP)		
Signal	Type	Description
clk	Input	Divided clock (25MHz)
reset	Input	System reset
Load (1:0)	Input	Register load selection
bdg_in (15:0)	Input	Load selection
reg_in (15:0)	Input	Word data from ALU
reg_out (15:0)	Output	Register output

Div_Clk.vhd		
Signal	Type	Description
clk	Input	System clock (50 MHz)
reset	Input	System reset
DClk	Output	Divided clock

H8_datapath.vhd		
Signal	Type	Description
clk	Input	Divided clock (25MHz)
reset	Input	System reset
acc_op (3:0)	Input	ACC operation control
alu_sel (5:0)	Input	ALU operation control
bdg_sel (1:0)	Input	Bdg select control
ma_sel (1:0)	Input	MA select control
md_sel	Input	MD select control
ccr_sel (1:0)	Input	CCR select control
loadccr	Input	Register7 load control
loadtmp (1:0)	Input	Register16 load control
loadir	Input	IR load control
loadma	Input	MA load control
loadmd (1:0)	Input	Register16 load control
loadcnct (1:0)	Input	Md_concnct control
loadpc (1:0)	Input	Register16 load control
rd_sel (4:0)	Input	Mux_rd select control
rs_sel (4:0)	Input	Mux_rs select control
rn_sel(4:0)	Input	Mux_rn select control
load_r0h (1:0)	Input	Register load control
load_r0l (1:0)	Input	Register load control
load_r1h (1:0)	Input	Register load control
load_r1l (1:0)	Input	Register load control
load_r2h (1:0)	Input	Register load control
load_r2l (1:0)	Input	Register load control
load_r3h (1:0)	Input	Register load control
load_r3l (1:0)	Input	Register load control
load_r4h (1:0)	Input	Register load control
load_r4l (1:0)	Input	Register load control
load_r5h (1:0)	Input	Register load control
load_r5l (1:0)	Input	Register load control
load_r6h (1:0)	Input	Register load control
load_r6l (1:0)	Input	Register load control
load_r7h (1:0)	Input	Register load control
load_r7l (1:0)	Input	Register load control
load_t0	Input	Load timer0
load_t1	Input	Load timer1
load_t2	Input	Load timer2
rd_bar	Input	Memory read
wr_bar	Input	Memory write
from_mem (15:0)	Input	Data from memory
sr (15:0)	Input	TBD
ir_out (15:0)	Output	IR to controller
tmp_out (15:0)	Output	TMP to controller
ccr_out (7:0)	Output	CCR to controller
t0_done	Output	Timer0 done
t1_done	Output	Timer1 done
t2_done	Output	Timer2 done
lower	Output	LSB of memory address
MA_out (15:0)	Output	Memory address
to_mem (15:0)	Output	Data to memory

controller.vhd		
Signal	Type	Description
clk	Input	Divided clock (25MHz)
reset	Input	System clock
ir_out (15:0)	Input	IR from datapath
tmp_out (15:0)	Input	TMP from datapath
ccr_out (7:0)	Input	CCR from datapath
t0_done	Input	Timer0 done
t1_done	Input	Timer1 done
t2_done	Input	Timer2 done
halt_current	Input	Halt request
int_request	Input	Interrupt request
switches (3:0)	Input	TBD
c_state (7:0)	Output	States output
acc_op (3:0)	Output	Acc control
acc_sel (3:0)	Output	Acc select control
alu_sel (5:0)	Output	Alu control
bdg_sel (1:0)	Output	Bdg control
ma_sel (1:0)	Output	Ma control
md_sel	Output	Mux_md control
ccr_sel (1:0)	Output	Mux_ccr control
loadccr	Output	Register7 load control
loadtmp (1:0)	Output	Register16 load control
loadir	Output	IR load control
loadma	Output	MA load control
loadmd (1:0)	Output	Register16 load control
loadcnct (1:0)	Output	Md_concnct control
loadpc (1:0)	Output	Register16 load control
rd_sel (4:0)	Output	Mux_rd control
rs_sel (4:0)	Output	Mux_rs control
rn_sel (4:0)	Output	Mux_rn control
load_r0h (1:0)	Output	Register load control
load_r0l (1:0)	Output	Register load control
load_r1h (1:0)	Output	Register load control
load_r1l (1:0)	Output	Register load control
load_r2h (1:0)	Output	Register load control
load_r2l (1:0)	Output	Register load control
load_r3h (1:0)	Output	Register load control
load_r3l (1:0)	Output	Register load control
load_r4h (1:0)	Output	Register load control
load_r4l (1:0)	Output	Register load control
load_r5h (1:0)	Output	Register load control
load_r5l (1:0)	Output	Register load control
load_r6h (1:0)	Output	Register load control
load_r6l (1:0)	Output	Register load control
load_r7h (1:0)	Output	Register load control
load_r7l (1:0)	Output	Register load control
load_t0	Output	Load timer0
load_t1	Output	Load timer1
load_t2	Output	Load timer2
rd_bar	Output	Read memory
wr_bar	Output	Write memory
ram_ce	Output	Memory enable
load_uat	Output	Load UAT
trigger_LED	Output	State Trigger to LED

acc.vhd		
Signal	Type	Description
clk	Input	System clock
reset	Input	System reset
acc_op (3:0)	Input	ACC operation selection
acc_in (7:0)	Input	Byte data from mux_acc
ccr_in (7:0)	Input	CCR data from register7
acc_out (7:0)	Output	ACC output
ccr_out (7:0)	Output	Modified CCR output

alu.vhd		
Signal	Type	Description
ccr_in (7:0)	Input	CCR data from register7
sport (7:0)	Input	Byte data from mux_rd
sport_w (15:0)	Input	Word data from mux_rd
dport (7:0)	Input	Byte data from mux_rs
dport_w (15:0)	Input	Word data from mux_rs
alu_sel (5:0)	Input	ALU operation selection
ccr_out (7:0)	Output	ALU CCR output
aluB_out (7:0)	Output	ALU byte output
aluW_out (15:0)	Output	ALU word output

bridge.vhd		
Signal	Type	Description
ccr_in (7:0)	Input	CCR data from register7
nport_B (7:0)	Input	Byte data from mux_rn
nport_W (15:0)	Input	Word data from mux_rn
bdg_sel (1:0)	Input	Mux select control
ccr_out (7:0)	Output	Bridge CCR out
bedW_out (15:0)	Output	Bridge word output

ir.vhd		
Signal	Type	Description
clk	Input	Divided clock (25MHz)
reset	Input	System reset
loadir	Input	Load selection
reg_in (15:0)	Input	Word data from Bridge
abs_out (15:0)	Output	ABS address
reg_out (15:0)	Output	IR output
IMM_out (7:0)	Output	Immediate data

LED.vhd		
Signal	Type	Description
clk	Input	Divided clock (25MHz)
reset	Input	System reset
Trigger	Input	Trigger from controller
Tri_mem_in	Input	Manual trigger
Tri_mem_out	Input	Manual trigger
Tri_add	Input	Manual trigger
States_IN (7:0)	Input	States from controller
mem_in (15:0)	Input	Memory write data
mem_out (15:0)	Input	Memory read data
mem_add (15:0)	Input	Memory address
LED_data_out (15:0)	Output	Data to LED display
LED_state_out (7:0)	Output	Data to LED display

md_concat.vhd		
Signal	Type	Description
clk	Input	Divided clock (25MHz)
reset	Input	System reset
alu_B (7:0)	Input	Byte data from ALU
md_in (15:0)	Input	Memory data
Load (1:0)	Input	Load selection
reg_out (15:0)	Output	Concat output

mem_interface.vhd		
Signal	Type	Description
wr_bar	Input	Write memory
rd_bar	Input	Read memory
data_in (15:0)	Input	Data to memory
from_mem (15:0)	Output	Data from memory

mux_acc.vhd		
Signal	Type	Description
acc_sel (3:0)	Input	Mux select control
r0h (7:0)	Input	Low byte of register1
r0l (7:0)	Input	High byte of register2
r1h (7:0)	Input	Low byte of register2
r1l (7:0)	Input	High byte of register3
r2h (7:0)	Input	Low byte of register3
r2l (7:0)	Input	High byte of register4
r3h (7:0)	Input	Low byte of register4
r3l (7:0)	Input	High byte of register5
r4h (7:0)	Input	Low byte of register5
r4l (7:0)	Input	High byte of register6
r5h (7:0)	Input	Low byte of register6
r5l (7:0)	Input	High byte of register7
r6h (7:0)	Input	Low byte of register7
r6l (7:0)	Input	Low byte of register1
r7h (7:0)	Input	High byte of register2
r7l (7:0)	Input	Low byte of register2
mux_out (7:0)	Output	Mux output

mux_ccr.vhd		
Signal	Type	Description
alu (7:0)	Input	CCR out from ALU
data (7:0)	Input	Byte data from ALU
bdg (7:0)	Input	BDG out from ACC
acc (7:0)	Input	CCR out from ACC
ccr_sel (1:0)	Input	Mux select control
mux_out (7:0)	Output	Mux output

mux_ma.vhd		
Signal	Type	Description
aa (15:0)	Input	ABS address from IR
pc (15:0)	Input	PC value from TMP
data (15:0)	Input	Word data from ALU
bdg (15:0)	Input	Word data from BDG
ma_sel (1:0)	Input	Mux select control
ma_out (15:0)	Output	Memory address output

mux_md.vhd		
Signal	Type	Description
md_sel	Input	Mux select control
cnct (15:0)	Input	Word data from CNCT
mux_out (15:0)	Output	Memory data output

mux_rd.vhd		
Signal	Type	Description
ccr (7:0)	Input	CCR data from register7
md_out (15:0)	Input	Memory data
pc (15:0)	input	PC value from TMP
r0h (7:0)	Input	Low byte of register1
r0l (7:0)	Input	High byte of register2
r1h (7:0)	Input	Low byte of register2
r1l (7:0)	Input	High byte of register3
r2h (7:0)	Input	Low byte of register3
r2l (7:0)	Input	High byte of register4
r3h (7:0)	Input	Low byte of register4
r3l (7:0)	Input	High byte of register5
r4h (7:0)	Input	Low byte of register5
r4l (7:0)	Input	High byte of register6
r5h (7:0)	Input	Low byte of register6
r5l (7:0)	Input	High byte of register7
r6h (7:0)	Input	Low byte of register7
r6l (7:0)	Input	Low byte of register1
r7h (7:0)	Input	High byte of register2
r7l (7:0)	Input	Low byte of register2
cnct (15:0)	Input	Word data from CNCT
rd_sel (4:0)	Input	Mux select control
mux_B_out (7:0)	Output	Byte output
mux_W_out (15:0)	Output	Word output

mux_rn.vhd		
Signal	Type	Description
rn_sel (4:0)	Input	Mux selection control
r0h (7:0)	Input	Low byte of register1
r0l (7:0)	Input	High byte of register2
r1h (7:0)	Input	Low byte of register2
r1l (7:0)	Input	High byte of register3
r2h (7:0)	Input	Low byte of register3
r2l (7:0)	Input	High byte of register4
r3h (7:0)	Input	Low byte of register4
r3l (7:0)	Input	High byte of register5
r4h (7:0)	Input	Low byte of register5
r4l (7:0)	Input	High byte of register6
r5h (7:0)	Input	Low byte of register6
r5l (7:0)	Input	High byte of register7
r6h (7:0)	Input	Low byte of register7
r6l (7:0)	Input	Low byte of register1
r7h (7:0)	Input	High byte of register2
r7l (7:0)	Input	Low byte of register2
ma (15:0)	Input	Address from MA
ir (15:0)	Input	Word data from IR
md_out (15:0)	Input	Memory data
pc (15:0)	Input	PC value from TMP
tmp (15:0)	Input	Word data from TMP
ccr (7:0)	Input	CCR data from register7
sr (15:0)	Input	TBD
mux_B_out (7:0)	Output	Byte output
mux_W_out (15:0)	Output	Word output

mux_rs.vhd		
Signal	Type	Description
ma (15:0)	Input	Address from MA
ir (15:0)	Input	Word data from IR
tmp (15:0)	Input	Word data from TMP
r0h (7:0)	Input	High byte of register0
r0l (7:0)	Input	Low byte of register0
r1h (7:0)	Input	High byte of register1
r1l (7:0)	Input	Low byte of register1
r2h (7:0)	Input	High byte of register2
r2l (7:0)	Input	Low byte of register2
r3h (7:0)	Input	High byte of register3
r3l (7:0)	Input	Low byte of register3
r4h (7:0)	Input	High byte of register4
r4l (7:0)	Input	Low byte of register4
r5h (7:0)	Input	High byte of register5
r5l (7:0)	Input	Low byte of register5
r6h (7:0)	Input	High byte of register6
r6l (7:0)	Input	Low byte of register6
r7h (7:0)	Input	High byte of register7
r7l (7:0)	Input	Low byte of register7
IMM (7:0)	Input	Immediate data from IR
rs_sel (4:0)	Input	MUX select control
mux_B_out (7:0)	Output	Byte output
mux_W_out (15:0)	Output	Word output

DivClk_UAT.vhd		
Signal	Type	Description
clk	Input	System clock (50 MHz)
reset	Input	System reset
DCI_UAT	Output	Divided clock (9.6 kHz)

register7L.vhd		
Signal	Type	Description
clk	Input	Divided clock (25MHz)
reset	Input	System reset
alu_W (15:0)	Input	Word output from ALU
alu_B (7:0)	Input	Byte output from ALU
acc_in (7:0)	Input	Data from ACC
load_sel	Input	Register load selection
reg_out (7:0)	Output	Register output

register_high.vhd		
Signal	Type	Description
clk	Input	Divided clock (25MHz)
reset	Input	System reset
load_sel (1:0)	Input	Register load selection
alu_W (15:0)	Input	Word output from ALU
alu_B (7:0)	Input	Byte output from ALU
acc_in (7:0)	Input	Data from ACC
reg_out (7:0)	Output	Register output

register_low.vhd		
Signal	Type	Description
clk	Input	Divided clock (25MHz)
reset	Input	System reset
load_sel (1:0)	Input	Register load selection
alu_W (15:0)	Input	Word output from ALU
alu_B (7:0)	Input	Byte output from ALU
acc_in (7:0)	Input	Data from ACC
reg_out (7:0)	Output	Register output

timer.vhd		
Signal	Type	Description
clk	Input	Divided clock (25MHz)
reset	Input	System reset
time_in (7:0)	Input	Initial value of timer
load	Input	Load initial value
done	Output	Done when timer = 0

UAT.vhd		
Signal	Type	Description
clk	Input	Divided clock (25MHz)
reset	Input	System reset
Load_UAT	Input	Trigger from controller
clk_UAT	Input	Divided clock (9.6kHz)
MA_In (15:0)	Input	Memory address
Data_In (15:0)	Input	Memory data
To_UAT	Output	Data to serial port

7.2 ASM chart for the controller

