RISC8 Core

Version 1.0

Written by Tom Coonan
tcoonan@mindspring.com

1 Introduction

The Free-RISC8 is a Verilog implementation of a simple 8-bit processor. The RISC8 is binary compatible with code targetted for the Microchip 16C57 processor. Code may be developed and debugged using tools available from a number of 3rd Party tool developers. Programs existing for the 16C57 may be ported to the RISC8 for use in an FPGA, etc. The RISC8 package includes a simple tool for converting Intel HEX format binary files into a format suitable for Verilog simulation using the supplied testbench.

The design is synthesizable and has been used by various people in the past within ASICs as well as FPGAs. The package consists of the following Verilog and C files:

File	***Description***
test.v	Top-level testbench, including the behavioral Verilog program memory
cpu.v	Top-level synthesizable module.
idec.v	The Instruction Decoder. This module is instanced underneath the cpu module.
alu.v	The ALU. This module is instanced underneath the cpu module.
regs.v	The Register File. This module is instanced underneath the cpu module.
exp.v	Optional Expansion Module. This is an example module that shows how an expansion circuit is added onto the design. The module supplied with this release implements a very simple DDS (Direct Digital Synthesis) circuit that is used for the DDS demo.
dram.v	Memory model for Register File 'D'ata memory (it's a Synchronous RAM)
pram.v	Memory model for Program Memory 'P'ata memory (it's a Synchronous RAM)
hex2v.c, hex2v.exe	A C program that translates Intel HEX format data into the Verilog $readmemh compatible .ROM file.
basic.asm, basic.hex, basic.rom	The "Basic Confidence" test program which exercises all the instructions.
dds.asm, dds.hex, dds.rom	A demo that uses the DDS circuit. The demo outputs an FSK "burst".
runit	A script containing the Verilog command line required.
risc8.pdf	This file.

2 Quick Start

Extract all the files from the supplied ZIP file to any desired directory (make a new one...). Accompanying files including test.rom are expected to all be in the same directory. Once all the files have been extracted from the archive, the "Basic Confidence" simulation is ready to run. This test verifies that the core is able to reset and run all the RISC8 instructions. The file 'runit' invokes the Verilog simulator along with all the necessary Verilog files. The following output is an example of what you should see:

```
>runit
Host command: /tools/cadence99/tools/verilog/bin/verilog.exe
Command arguments:
    test.v
    cpu.v
    alu.v
    regs.v
    idec.v
    exp.v
    dram.v
    pram.v
```

```
VERILOG-XL 2.8.p001 log file created Dec 13, 1999  16:09:07
VERILOG-XL 2.8.p001   Dec 13, 1999  16:09:07
[... SNIP all the Verilog informative output ...]
Compiling source file "test.v"
Compiling source file "cpu.v"
Compiling source file "alu.v"
Compiling source file "regs.v"
Compiling source file "idec.v"
Compiling source file "exp.v"
Compiling source file "dram.v"
Compiling source file "pram.v"
Highest level modules:
test

Reading in SIN data for example DDS in EXP.V from sindata.hex
Free-RISC8.  Version 1.0
Free-RISC8 1.0.  This is the BASIC CONFIDENCE TEST.
Loading program memory with basic.rom
MONITOR_OUTPUT_SIGNATURE: Expected output observed on PORTB: 00
MONITOR_PORTC: Port C changes to: 00
MONITOR_PORTB: Port B changes to: 00
End RESET.
MONITOR_OUTPUT_SIGNATURE: Expected output observed on PORTB: 01
MONITOR_PORTB: Port B changes to: 01
MONITOR_OUTPUT_SIGNATURE: Expected output observed on PORTB: 02
MONITOR_PORTB: Port B changes to: 02
MONITOR_OUTPUT_SIGNATURE: Expected output observed on PORTB: 03
MONITOR_PORTB: Port B changes to: 03
MONITOR_OUTPUT_SIGNATURE: Expected output observed on PORTB: 04
MONITOR_PORTB: Port B changes to: 04
MONITOR_OUTPUT_SIGNATURE: Expected output observed on PORTB: 05
MONITOR_PORTB: Port B changes to: 05
MONITOR_OUTPUT_SIGNATURE: Expected output observed on PORTB: 06
MONITOR_PORTB: Port B changes to: 06
MONITOR_OUTPUT_SIGNATURE: Expected output observed on PORTB: 07
MONITOR_PORTB: Port B changes to: 07
MONITOR_OUTPUT_SIGNATURE: Expected output observed on PORTB: 08
MONITOR_PORTB: Port B changes to: 08
Done monitoring for output signature.  9 Matches, 0 Mismatches.
SUCCESS.
End of simulation signalled.  Killing simulation in a moment.
L232 "test.v": $finish at simulation time 2641100
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.4 secs to compile + 0.1 secs to link + 1.7 secs in
simulation
End of VERILOG-XL 2.8.p001   Dec 13, 1999  16:09:10
```

3 System Architecture

A system diagram for the Verilog core is shown in Figure 2.0. Modules boundaries are bolded with the Verilog filename indicated.

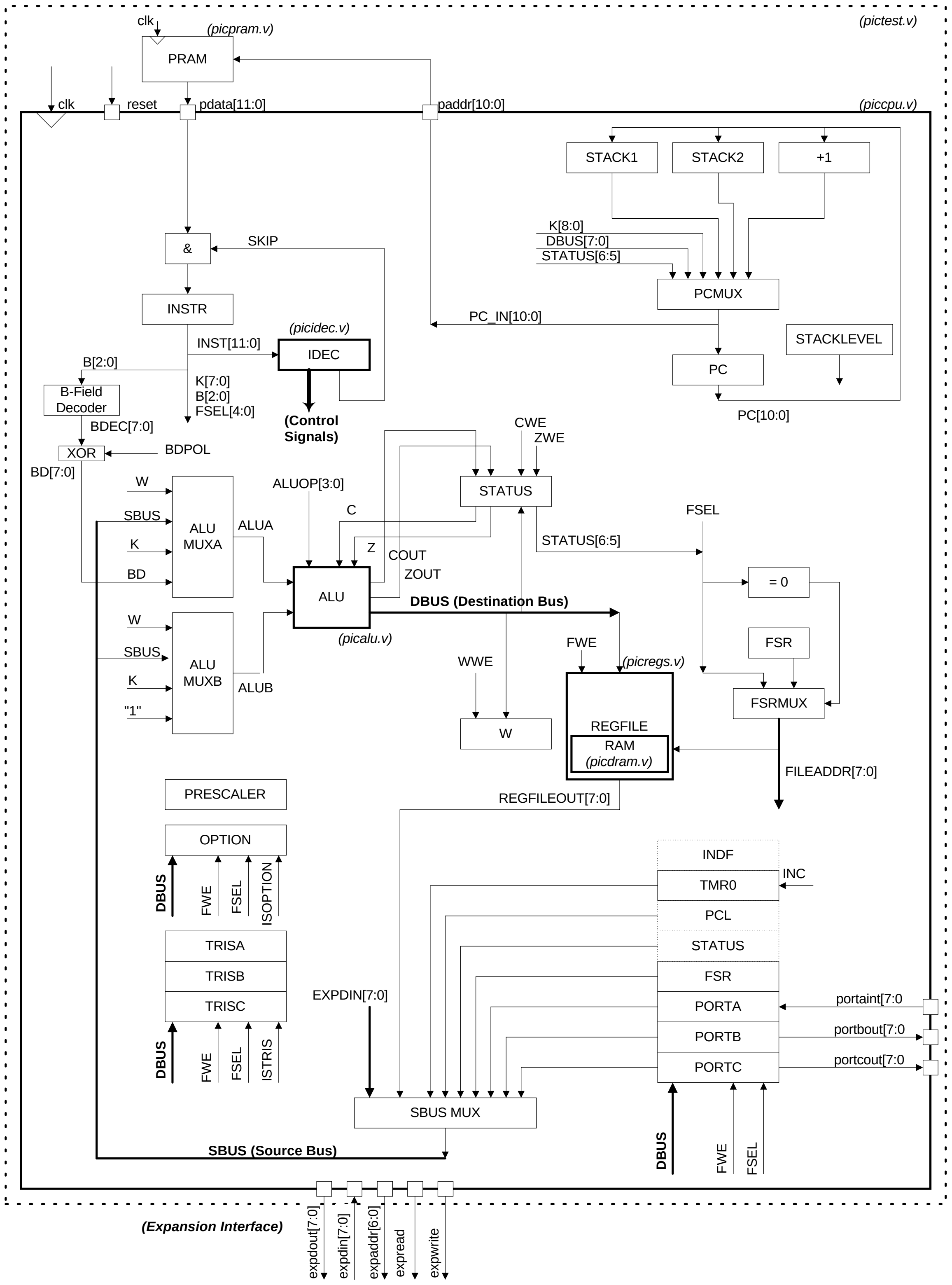
The RISC8 is a Harvard Architecture and is binary code compatible with the Microchip 16C57. Instructions are 12-bits wide and the data path is 8-bits wide. There are up to 72 data words and up to 2048 program words. It has an accumulator-based instruction set (33 instructions). The W register is the accumulator. The Program Counter (PC) and two Stack registers allow 2 levels of subroutines (this could be easily expanded). The RISC8 pipelines its Fetch and Execute. The Register File uses a banking scheme and an Indirect Addressing mode. The core's Register File is implemented as a flip-flop based Register File. The Program memory (PRAM) is a separate memory from the Register File and is outside the core. The PRAM is currently a simple Verilog memory array residing in test.v. The core is synchronous with one clock and has one synchronous reset. It is scan-insertion friendly.

The ALU is very simple and includes the minimal set of 8-bit operations (ADD, SUB, OR, AND, XOR, ROTATE, etc.). The Instruction Decoder is a purely combinatorial look-up table that supplies key control signals. The basic 16C57 I/O Ports exist, but full bi-directional control is not automatically available (this could be implemented if truly desired in a core).

No interrupts are supported in the 16C5X family and are not offered in the RISC8. Instructions execute in one cycle with the exception of branching instructions requiring 2 cycles (when branches are actually taken). An argument often cited for the lack of interrupts is that the fast one-cycle execution and bit test instructions allows for very fast polling, and therefore reduces the need for interrupts.

Little debug is built into the core itself. Off-the-shelf development environments offer very good debugging capabilities including integrated Assemblers, simulator and debuggers with breakpoints, etc. Once a rough cut at the firmware is done in such a tool, then the Verilog simulator and waveform viewers allow further debugging with the core. The test.v module provides some limited debugging such as printing out changes to I/O ports, displaying updates to Register File locations, etc.

Expansion is done through an expansion bus on the main cpu.v module interface. The bus provides a basic address, read, write data in and out set of signals. The module exp.v shows one simple expansion circuit. If several expansion modules must coexist using this bus, then they must work out their own muxing scheme to drive expdin into the core. See the section on 'Expansion' for more details.



RISC8 System Diagram.

4 Compatibility with Microchip 16C57 Devices

The RISC8 can execute binary code compatible with the 16C57. Several flavors of 16C5X exist with different amounts of addressable memory, and different numbers of I/O pins. The Verilog core can be changed to correspond with any number of these I/O combinations or memory combinations.

The following features or characteristic differ:

Feature	Microchip 16C57	RISC8
Oscillator	Has several oscillator options.	Only has simple, direct clock input.
Clocking	Internally uses a 4-phase clock	One phase clock.
Reset	The 16C57 uses an active low MRST and a power-up circuit. Some 16C57s have brownout.	Simple active HIGH reset.
Sleep	Has sleep instruction and circuitry.	None. Sleep instruction will be ignored.
Tristatable ports	Has bi-directional ports with the TRIS instruction to program direction.	No actual tristates on buffers. Must wire as needed. Currently set up with PORTA as input and PORTB and PORTC as outputs. The TRIS instruction DOES actually load a TRISA, TRISB and TRISC registers, but these registers don't connect to anything at this time. May be used for debug purposes.
Watchdog Timer	WDT circuit	None.
Timer0	Free-running or external source	Only clocked by internal clock. Uses the 3 prescaler bits in OPTION register.
OPTION register and instruction	16C57 uses it	Only the bits associated with TIMER0 do anything.

5 Module Hierarchy

The hierarchy is as follows:

test.v			Testbench. Includes program ROM.
	cpu.v		Top-level cpu module
		idec.v	Instruction Decoder (combinational)
		alu.v	ALU (combinational)
		regs.v	Register File interface
		dram.v	Memory model, Synchronous 72x8
	pram.v		Memory model, Synchronous 2048x12
	exp.v		Example expasion module (a DDS for DDS demo).

Each major module is described in the following sections.

6 Synthesis

Four core modules (cpu, idec, alu and regs) are directly synthesizable. Special consideration is required for the two RAMs. The design should be fully testable using Full Scan, except for the memories. There are no intentional latches or tristates in the design. The main clock is the only clock in the design. The main reset line does not go through any additional gating or logic.

Memories require special consideration. Specific FPGA and ASIC technologies have specific RAM cells and techniques. The pram.v and dram.v modules may be thought of as “wrappers” inside of which the technology specific RAM details are implemented.

The Register File memory is represented in the Verilog lowest-level module, dram.v. This module is a memory model for a synchronous RAM. This module is intended as the default behavioral memory model and includes // synopsys translate_off directives. The module is synthesizable, however, should a flip-flop based Register File be desired. The Register File memory must implement a read/modify/write behavior. Writes should be registered (synchronous) but reads must be immediate (asynchronous). This behavior is required due to instructions that must must read/modify/write file registers within a single instruction, for example;

```
        incf        12, f       ; This instruction increments the file register at location 12
```

Many FPGA and ASIC technologies provide this type of memory.

The Program memory may be implemented as a ROM if desired, since it is not written to by the RISC8. Alternatively, an ASIC or FPGA implementation may want to implement this as a RAM for booting code. Small programs could actually be implemented as a logic-based CASE statement and synthesized. This is left up to the implementor. The testbench utilizes a simple register array and $readmemh calls load this “memory” from the “.rom” file.

7 CPU Module

The CPU module is the top-level synthesizable module. This is where all the special registers are implemented such as the INST, W, STACK1, STACK2 and the PC. Program Flow control is implemented here. All the internal busses and multiplexors are also implemented here. All I/O occurs here. Any special circuitry such as the Timer or custom circuitry is implemented in this module.

The RISC8 has 3 major ways it changes program flow; 1) a GOTO instruction, a 2) CALL subroutine instruction and 3) Conditional SKIP instructions.

GOTO instructions encode the destination address in literal field of the instruction. Subroutines are done in hardware using explicit STACK registers (versus a software stack and Stack Pointer registers). This is partly the result of the Harvard architecture and the strict separation of program and data spaces. Skip instructions are conditional and usually involve a bit test on a register.

Whenever a branch is taken, the Fetch/Execute pipeline must be “stalled”. Normally, the next instruction is always being fetched while the current instruction is executed. When a branch is taken, then the upcoming instruction is actually invalid. The RISC8 rectifies this situation by forcing a NOP instruction into the INST register on the instruction following a branch. This same trick is done in the core. The NOP instruction is, conveniently, 0x0000. Forcing a NOP instruction is done by simply anding the output of the INST register with zeros whenever a branch is detected. The core’s internal SKIP signal is asserted whenever a branch is detected and the NOP is to be forced.

Another artifact of the Fetch/Execute pipeline is the reset vector. The reset vector (the first address fetched and executed) is the last address in the code space. The PC is loaded, on reset, with the reset vector (e.g. 0x1FF) and a NOP is forced as the first instruction. In this way, the first address that is actually Fetched is 0x000 (e.g. 0x1FF + 1) where the program must begin. The core may be reset at any time by asserting the reset input for at least one clk edge.

8 Memory Interfaces

The interface to program memory is straight-forward in terms of the core itself. An 11-bit address is output and a 12-bit data input is expected. This read is synchronous. The program memory (PRAM) itself is modeled in pram.v which is a very simple synchronous ram model. The PRAM is outside the core (inside test.v but outside cpu.v).

The Register File interface is a synchronous interface with clk and reset inputs. Addressing inputs include a 2-bit **bank** and 5-bit **location** input. **Read** and **write** enable signals are inputs and there are two separate 8-bit data busses for input and output. The regs.v module performs the address logic where some words are mirrored into a common set of addresses. Beneath regs.v is the actual synchronous RAM model in dram.v. This module is similar to pram.v and is a simple synchronous RAM model.

9 ALU

The ALU is implemented in the alu.v file. The ALU is purely combinatorial. It has 2 8-bit data inputs, A and B as well as a single-bit CON Carry in input. A 4-bit operand input selects the ALU operation. It has an 8-bit data output and a single-bit carry output and also a single-bit zero output. The ALU does not select the appropriate source for its inputs nor does it decide when status flags are updated. This is done at the higher level by the Instruction Decoder and the CPU module.

The ALU supports the following operands.

ALU Operand Select Code	Operation	Description
0000	ADD	A + B (The 16C5X does NOT add with carry input)
1000	SUB	A - B (The 16C5X does NOT subtract with borrow input)
0001	AND	A AND B
0010	OR	A OR B
0011	XOR	A XOR B
0100	COM	NOT A
0101	ROR	{A[0], A[7:1]}
0110	ROL	{A[6:0], A[7]}
0111	SWAP	{A[3:0], A[7:4]}

Figure 4.1 ALU Operations

Note that an Add with carry instruction is absent. All RISC8 instructions must use this basic set of supported operations.

10 Instruction Decoder

Instruction Decoding is implemented in the dec.v Verilog module. It is purely combinatorial. It is specifically implemented as a large Verilog **casex** statement; one or two case clauses per instruction (many instructions are broken into the d=0 and d=1 cases). Its outputs is a set of decodes used for various control purposes described below.

An instruction begins to be executed once it is registered into the INST register. This occurs every cycle, except when a branch is taken (more on this later). The RISC8 has 33 instructions. The Instruction in the INST register is 12-bits wide. Several fields are frequently defined in instructions, including the F, K and B fields. These subfields are created in the core from the original 12 INST register bits. The Instruction Set summary figure from the 16C57 data sheet follows for reference:

Each instruction implies a particular set of control signals for controlling, ALU source inputs, PC updating, Status register write enables, Register File addresses, etc. These control signals are encoded in one place in the module, idec.v. This module produces 15 control outputs.

The Instruction Decoder controls what goes into the ALU and what operation the ALU performs. The ALU has two input ports; A and B. The A and B inputs are in turn driven by multiplexors which select from either W, SBUS, K or the BD vector for ALUA, or from W, SBUS, K or the literal 00000001. Almost all data that will be written back to the register file goes through the ALU. Frequently, particular ALU operations all the transfer of data. Use these ALU "tricks" allows us to minimize the number of buses in the design. For example, to clear a register, the W register is XORed with itself in order to obtain 00000000. Likewise, another trick is to OR data with itself in order to simply "copy" the data through the ALU.

Status flags such as the Z and C bits (Zero and Carry out) are updated depending on the instruction. For each instruction, an enable signal must be generated. Likewise, enables for writing to the W and the Register File must be generated. Table 5.1 specifies all the Instruction Decoder control signals per instruction. This table is similarly implemented the Instruction Decoder module (idec.v).

Instruction	**ALU A Source**	**ALU B Source**	**ALU Operand**	**Output of ALU**	**WWE**	**FWE**	**ZWE**	**CWE**	**BDPOL**
Byte-Oriented File Register Operations									
NOP	X	X	X	X	0	0	0	0	0
MOVWF	W	W	OR	W	0	1	0	0	0
CLRW	W	W	XOR	0	1	0	1	0	0
CLRF	W	W	XOR	0	0	1	1	0	0
SUBWF (d=0)	F	W	SUB	F - W	1	0	1	1	0
SUBWF (d=1)	F	W	SUB	F - W	0	1	1	1	0
DECF (d=0)	F	1	SUB	F - 1	1	0	1	0	0
DECF(d=1)	F	1	SUB	F - 1	0	1	1	0	0
IORWF (d=0)	W	F	OR	W | F	1	0	1	0	0
IORWF(d=1)	W	F	OR	W | F	0	1	1	0	0
ANDWF (d=0)	W	F	AND	W & F	1	0	1	0	0
ANDWF(d=1)	W	F	AND	W & F	0	1	1	0	0
XORWF (d=0)	W	F	XOR	W ^ F	1	0	1	0	0
XORWF(d=1)	W	F	XOR	W ^ F	0	1	1	0	0
ADDWF (d=0)	W	F	ADD	W + F	1	0	1	1	0
ADDWF(d=1)	W	F	ADD	W + F	0	1	1	1	0
MOVF (d=0)	F	F	OR	F	1	0	1	0	0
MOVF(d=1)	F	F	OR	F	0	1	1	0	0
COMF (d=0)	F	X	NOT	~F	1	0	1	0	0
COMF(d=1)	F	X	NOT	~F	0	1	1	0	0
INCF (d=0)	F	1	ADD	F + 1	1	0	1	0	0
INCF(d=1)	F	1	ADD	F + 1	0	1	1	0	0
DECFSZ (d=0)	F	1	SUB	F - 1	1	0	0	0	0
DECFSZ(d=1)	F	1	SUB	F - 1	0	1	0	0	0
RRF (d=0)	F	X	ROR	{C, F[7:1]}	1	0	0	1	0
RRF(d=1)	F	X	ROR	{C, F[7:1]}	0	1	0	1	0
RLF (d=0)	F	X	ROL	{F[6:0], C}	1	0	0	1	0
RLF(d=1)	F	X	ROL	{F[6:0], C}	0	1	0	1	0
SWAPF (d=0)	F	X	SWAP	{F[3:0], F[7:4]}	1	0	0	0	0
SWAPF(d=1)	F	X	SWAP	{F[3:0], F[7:4]}	0	1	0	0	0
INCFSZ (d=0)	F	1	ADD	F + 1	1	0	0	0	0
INCFSZ (d=1)	F	1	ADD	F + 1	0	1	0	0	0
Bit-Oriented File Register Operations									
BCF	F	K	BCLR	F & (~(1 << K))	0	1	0	0	1
BSF	F	K	BSET	F | ~(1 << K)	0	1	0	0	0
BTFSC	F	K	BTST	F & (1 << K)	0	0	0	0	0
BTFSS	F	K	BTST	F & (1 << K)	0	0	0	0	0
Literal and Control Operations									

OPTION	W	W	OR	W	0	1	0	0	0
SLEEP	X	X	X	X	0	0	0	0	0
CLRWDT	X	X	X	X	0	0	0	0	0
TRIS	W	W	OR	W	0	1	0	0	0
RETLW	K	K	OR	K	0	0	0	0	0
CALL	X	X	X	X	0	0	0	0	0
GOTO	X	X	X	X	0	0	0	0	0
MOVLW	K	K	OR	K	1	0	0	0	0
IORLW	W	K	OR	K | W	1	0	1	0	0
ANDLW	W	K	AND	K & W	1	0	1	0	0
XORLW	W	K	XOR	K ^ W	1	0	1	0	0

Instruction Decoder table look-up

11 Register File

The Register File is implemented in the Verilog file regs.v. The Register File is somewhat more complicated than the program code memory. The program memory is outside the core, and is implemented as a simple memory. The Register File requires an input write port and an output read port. It is also partitioned into several "banks". These banks are sometimes mapped into one common set of memory words. It is also desirable to "nullify" particular locations which are used for custom peripherals (so as to not waste silicon). The module dreg.v contains all the logic that maps register addresses (which includes banks and offsets) to physical RAM addresses. Beneath this module is the generic memory model (dram.v). Table 6. shows ...

File Register		**Final RAM Address**	**Description**
FSR[6:5]	**f[4:0]**		
00	0x00 – 0x07	N/A	Special Purpose Registers (8)
00	0x08 – 0x0F	0x00 – 0x07	Common Registers (8)
00	0x10 – 0x1F	0x08 – 0x17	Bank #0 Registers (16)
01	0x00 – 0x07	N/A	Special Purpose Registers (8 mirrored)
01	0x08 – 0x0F	0x00 – 0x07	Common Registers (8 mirrored)
01	0x10 – 0x1F	0x18 – 0x2F	Bank #1 Registers (16)
10	0x00 – 0x07	N/A	Special Purpose Registers (8 mirrored)
10	0x08 – 0x0F	0x00 – 0x07	Common Registers (8 mirrored)
10	0x10 – 0x1F	0x30 – 0x47	Bank #2 Registers (16 mirrored)
11	0x00 – 0x07	N/A	Special Purpose Registers (8 mirrored)
11	0x08 – 0x0F	0x00 – 0x07	Common Registers (8 mirrored)
11	0x10 – 0x1F	0x48 – 0x5F	Bank #3 Registers (16 mirrored)

At this time, the Register File contains 70 8-bit data words. The 16C57 has 72 registers. The core has 70 registers available because, at this time, there are 2 locations used for a custom peripheral. As peripherals are added in this way, locations must be taken from the memory space.

The 16C57 devices use a 4-phase clock derived from a external crystal. The RISC8 uses a single clock input and derives a 4 phase synchronous clock. When considering using memory hard cells, this clocking must be considered carefully. The original 16C57 utilized different clock phases to accomplish a Register File read followed by write operation. Likewise, the core uses these phases in order to perform a read and a write within a single instruction "cycle".

The Register File may be read and written to during one instruction cycle. By using the Q1-Q4 phases, a simple synchronous RAM can still be used. Data is read from the Register File during Q2 (e.g. Q1 is used to enabled the read). Updates to the Register File occur at the end of Q4. The data retrieved from the Register File is presented to the SBUS mux. During the instruction cycle, an output from the ALU will drive the DBUS which goes to the Register File's data input. If the instruction decoder asserts the FWE write enable, then this data must be written back into the Register File at the end of Q4.

12 Firmware Development

An advantage to using the RISC8 over a purely home-brew processor is the wealth of existing development tools. Development is typically done both on the PC and on UNIX. Existing code development tools used to develop code for the 16C57 may be used for the RISC8. It is assumed that the development of working code should be done in one of the many high-quality assmebler/debugger tools that are available from a number of 3rd-party vendors. Once an Intel HEX format binary file is produced, it must be converted into a format acceptable to the Verilog $readmemh format. The included C program, hex2v.c, can do this conversion. The program is a simple command-line program that acccepts the Intel HEX filename as an input argument. The output is the $readmemh-compatible data and can be piped to a ".rom" file.

After the .ROM file is made, the Verilog simulation can be run;

```
verilog  test.v  cpu.v  regs.v  idec.v  alu.v exp.v  dram.v  pram.v
```

The testbench test.v provides some limited debugging capability. Several Verilog 'monitor' tasks are available that wil display changing register values, etc. It is expected that a waveform viewer such as CWAVES or UNDERTOW will be used for detailed debugging.

C compilers may also be used just as 16C57-compatible Assemblers may be used as long as they can generate the required Intel HEX format output.

13 Expansion

In this case, 'Expansion' refers to the integration of new custom modules to the system. This is done through a special set of signals in the cpu module interface. Any number of addresses in the top of the register address space may be reserved for an expansion circuit. The exp.v module provided reserves 2 such locations. The exp.v module implements a very simple DDS circuit used in the DDS demo.

Note that locations reserved for an expansion circuit must be decoded in the cpu.v module. Look for the block of code that drives the signal, expsel. The case statement should be modified as needed. The initial configuration is that the top 4 locations are reserved for expansion circuits. Note that these top 4 locations CAN NOT be used for normal register storage.

The expansion interface signals are:

Signal	Description
expdin[7:0]	Input back to the RISC8 core. This is 8-bit data from the expansion module(s) to the core. Should be valid when expread is asserted.
expdout[7:0]	Output from the RISC8 core. This is 8-bit data to the expansion module(s) from the core. Is valid when expwrite is asserted. The expansion modules are responsible for decoding expaddr in order to know which expansion address is being written to.
expaddr[6:0]	This is the final data space address for reads or writes. It includes any indirect addressing. NOTE: within the cpu, the signal expsel must be asserted when an expansion location is being addressed versus when an ordinary Register File location is being addressed. The cpu needs to know the difference so that is controls the MUX properly.
expread	Asserted (HIGH) when the RISC8 core is reading from an expansion address.
expwrite	Asserted (HIGH) when the RISC8 core is writing to an expansion address.

Expansion circuits should use clk and reset in the normal way. Accesses are done in one cycle. The test module exp.v illustrates how to interface to the Expansion Bus, and is used in the DDS demo.

14 Test Programs

Two Assembler programs and HEX files are included in the package. The ‘basic’ program is a simple program that exercises all the RISC8 instructions. The testbench test.v is initially configured to run this test. A second program, DDS, is included that demonstrates a somewhat realistic program that uses the expansion capability.

The BASIC program runs a series of 9 subtests. All tests are self-verifying and output to PORTB a byte code indicating SUCCESS or FAIL. A companion Verilog task in test.v monitors for these codes and, if the BASIC test passes, will report success. The initial configuration of test.v should do this when it is run. See basic.asm for more details.

The DDS program demonstrates a simple program that also uses the exp.v expansion circuit. It will control the DDS circuit and will cause a modulated sin wave on the dds_out pin from the cpu module. If this output is observed with a waveform viewer set to an “Analog” format, the waveform is clearly seen. See dds.asm for more details.

15 Bugs

Following are some known bugs and deficiencies.

Item #	**Description**
1	The DC bit in STATUS is unimplemented and won’t work.
2	TRIS only seems to update the TRIS register, but doesn’t affect ports.