PT13

Compact 8-bit Microprocessor IP Core

User Manual

Revision 0.6
13th April 2017

Revision History

Date	Revisions	Version
21-08-2011	First Draft.	0.5
13-04-2017	Re-written for Verilog code.	0.6

Contents

Tables

Figures

1. Introduction

PT13 is a microprocessor IP core intended for simple control applications. It has a compact but efficient instruction set and is designed to use internal FPGA memory for both program and data storage making it a very cost effective solution for embedded control applications.

An editor and assembler are provided for efficient generation of machine code.

The intellectual property block is written in TDF file format, a PALASM style test based design language for Altera (Intel) FPGAs.

Typical resource usage for an Altera FPGA is shown in Table 1 (as compiled for an EP4CE15 FPGA used on the SM03 evaluation board).

Logic Elements	Memory Bits	M9K blocks	9x9 Multipliers	18x18 multipliers
409	0 (internal)	0 (internal)	0	0

Table 1 PT13 Altera FPGA resource requirements

2. Signal Interconnections

The PT13 signal interconnect diagram is shown in Figure 3.

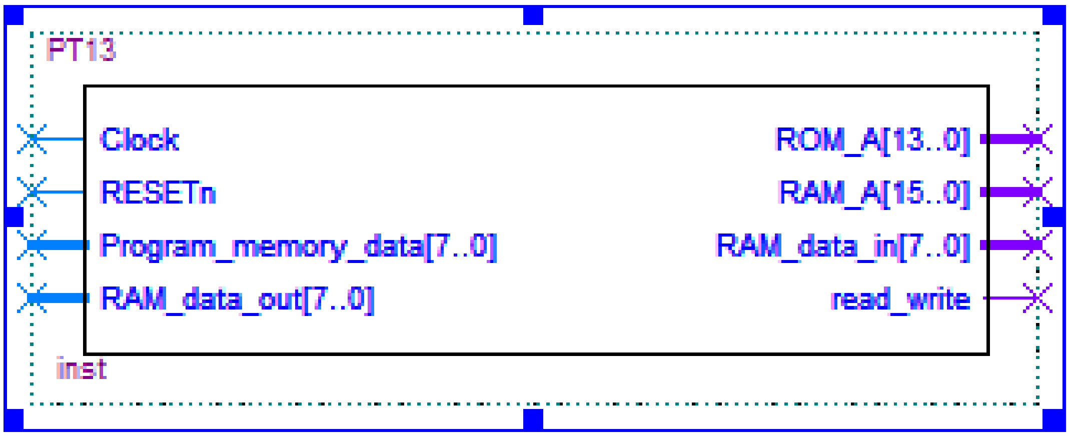

Figure 1 PT13 Input/Output connections

The signal descriptions are shown in Table 2, below.

Signal	Description
Clock	Clock input. The microprocessor cycle time is 1/16th of clock. The rising edge of the clock is used.
RESETn	When asserted (= ‘0’), resets the internal state machines, clears all registers and clears the program counter which starts executing instructions from location $0000.
Program_memory_data[7:0]	The data output from the program memory (ROM).
RAM_data_out[7:0]	The data output from the read/write memory (RAM) or input/output devices.
ROM_A[13:0]	The address output to read only program memory. Up to 16kbytes of ROM is supported.
RAM_A[15:0]	The address output to read/write memory or used to address input/output devices. Up to 64kbytes is supported.
RAM_data_in[7:0]	Data to be written into the RAM or input/output device.
read_write	Selects whether the RAM or input/output device is to read from (= ‘1’) or written to (= ‘0’).

Table 2 Input/Output signals

The Verilog instantiation for the PT13 is shown below:

```
PT13 PT13_inst
(
.Clock(Clock_sig) ,                                    // input  Clock_sig
.RESETn(RESETn_sig) ,                                  // input  RESETn_sig
.Program_memory_data(Program_memory_data_sig) ,        // input [7:0] Program_memory_data_sig
.RAM_data_out(RAM_data_out_sig) ,                      // input [7:0] RAM_data_out_sig

.ROM_A(ROM_A_sig) ,                                    // output [13:0] ROM_A_sig
.RAM_A(RAM_A_sig) ,                                    // output [15:0] RAM_A_sig
.RAM_data_in(RAM_data_in_sig) ,                        // output [7:0] RAM_data_in_sig
.read_write(read_write_sig)                            // output  read_write_sig
);
```

3. Technical Overview

A simplified block diagram of the PT13 compact microprocessor is shown in Figure 4.

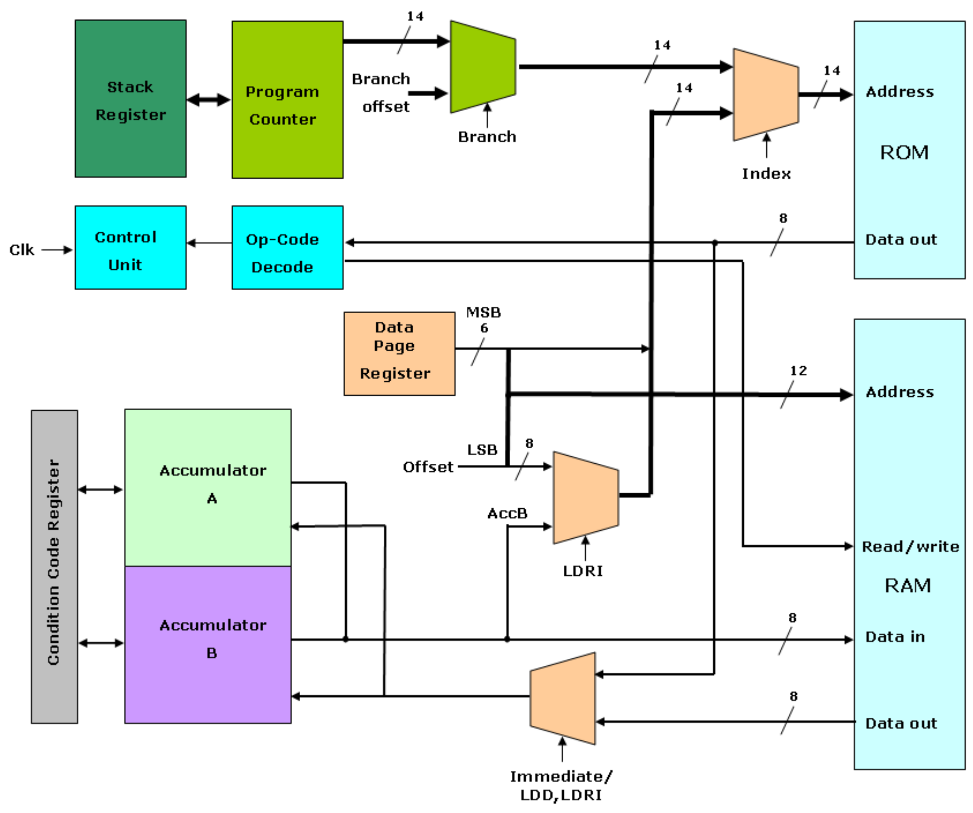

Figure 2 PT13 Block Diagram

To reduce the logic cell count for the IP core each instruction is designed to take the same number of clock cycles, 16, regardless of the addressing mode used. Therefore the execution time of each instruction with, for example, a 27MHz clock is 1.6875MHz, or 0.59µs/instruction.

After a reset the internal state machines are reset and the Program counter (PC) is cleared. The program counter is then used to address the program memory, starting at address $0000. The data at that first location is then decoded by the Op-code decode block which, in turn, controls the sequencing of the control unit. Each machine cycle of the processor takes 16 input clock cycles and is sub-divided into Fetch, Decode, Execute and Write-Back cycles.

If the Op-code is a branch instruction the branch address is read from the subsequent bytes of the ROM and replaces the PC contents to form the new address. Three stack pointer registers (SP) allows up to 3 program counter contents to be stored permitting a maximum of 3 nested sub routines to be called, (note: no RAM is needed for subroutine calls but only the program counter contents are stored).

Data is read from the RAM using an address formed by the lower 4 bits of a data page register and by an 8-bit offset which is part of the instruction. Data is written to the RAM using the same addressing method. The RAM address space is also used to connect peripheral devices.

This direct method of addressing may also be used to read data from the ROM, but with an extended address range using 6 bits of the data page register. The ROM contents may also be read using indexed addressing instructions where the lower byte is the content of Accumulator B. This is useful for reading data from tables for example.

PT13 has two accumulators, either of which may be operated on for all instructions save for the indexed addressing mode. Both accumulators have two status flags which are used for conditional branch instructions; zero which indicates the contents of the corresponding accumulator are $00 and which is valid for all instructions, and carry, which is used in arithmetic instructions and also for arithmetic shift and rotate instructions.

4. Connecting external memory and peripherals

Figure 3 shows the connections for a typical minimum ROM and RAM implementation. The RAM area in this case is implemented in the Altera FPGA's dedicated RAM but smaller RAM instances may be synthesised using logic blocks (registers) allowing the use of the PT13 in CPLDs such as the MAX series of devices.

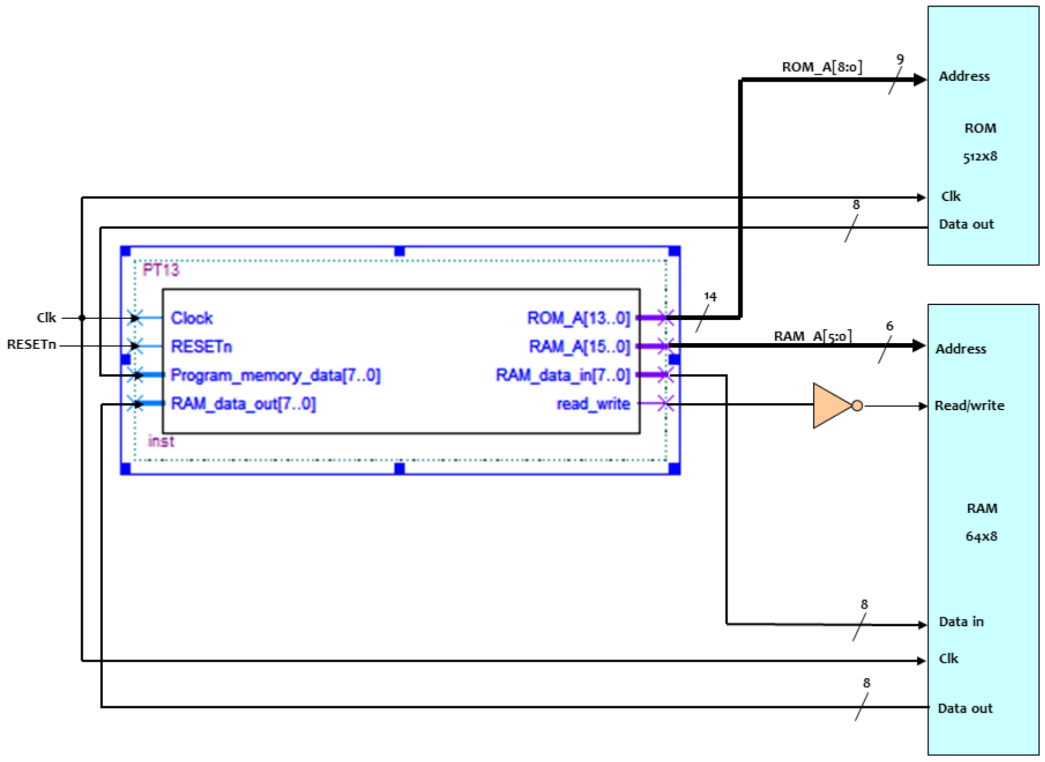

Figure 3 PT13 Memory connections.

An example of program read only memory (ROM) Verilog instantiation is shown below. The ROM is synchronous with a two clock pipeline delay using the same clock as the PT13. An example Verilog ROM module is supplied together with the PT13 code.

```
// Program memory
PT13_ROM PT13_ROM(.aclr(!RESETn), .address(rom_a[8:0]), .clock(Clock),
                                        .q(ROM_data[7:0]));
```

If using Altera FPGAs, the ROM can use Altera 1 port ROM Megafunction. The assembler, by default, automatically creates a .mif file and when instantiating the ROM you can point to that file so the latest code will always be compiled automatically.

Smaller ROMs may of course be used and the mif file size in the assembler may be limited accordingly.

The PT13 stack pointer does not rely on external memory (using internal registers) so it is possible to use the PT13 with RAM or to use registers as a small RAM area for data storage.

A small area of RAM will likely be necessary for storing variables. An example Verilog RAM module is supplied together with the PT13 code. An example of the Verilog instantiation is shown below.

This write enable has selected a single 64x8 bit block of RAM at address $000-$03F.

```
assign RAM_WR = (!ram_a[11] & !ram_a[10] & !ram_a[9] & !ram_a[8] & !ram_a[7] & !ram_a[6] & !read_write);

ram_infer_generic #(.data_width(8), .addr_width(6))
PT13_RAM(.clk(Clk27), .addr(ram_a[7:0]), .wrdata(ram_data_in[7:0]), .we(RAM_WR), .en(1'b1),
            .rddata(RAM_memory[7:0]));
```

I/O peripherals use the same address space as the user memory and it is necessary to add some simple address decoding and multiplexing to allow these to function.

For example we can use the address line RAM_A[6] to select between our 64x8 user and a write control register as shown in Figure 4. In this example our user memory lies between address $0000-$003F and the latch at any address from $003F-$007F.

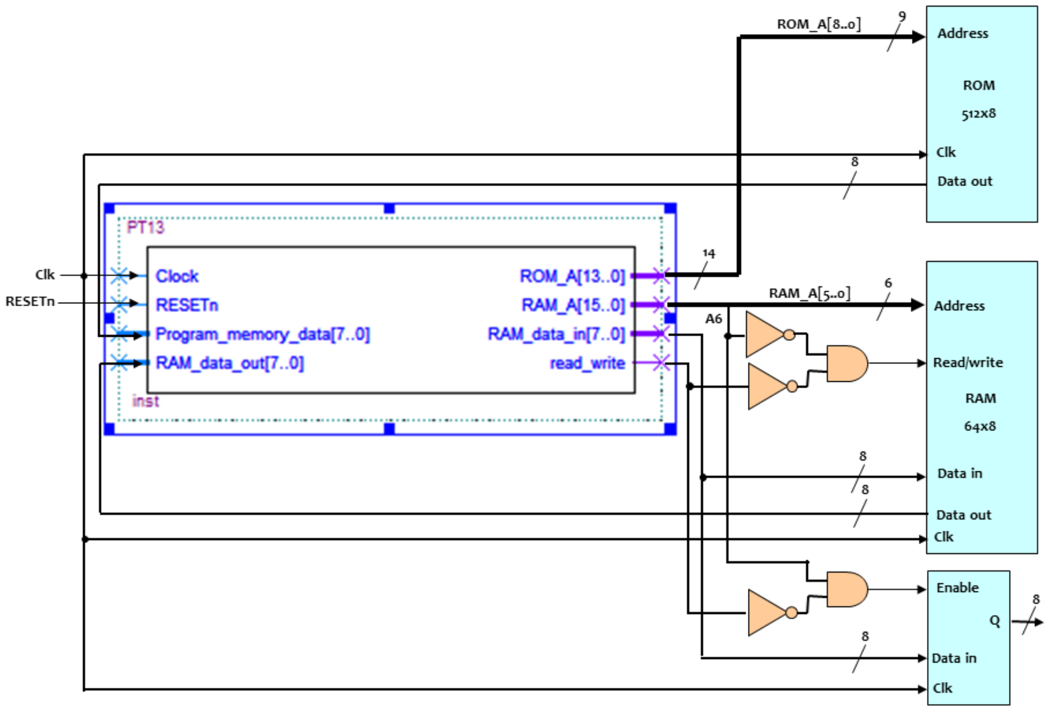

Figure 4 Connecting peripherals.

The Q outputs of the latch can be used as DC control signals.

Should we wish to read back the data, or read any inputs, we need to add an external multiplexer for the data. In the example in Figure 5 we are reading back the Q outputs of the latch. The multiplexer uses RAM_A[6] to select between the data output from the RAM (if A6='0') or the latch (if A6='1').

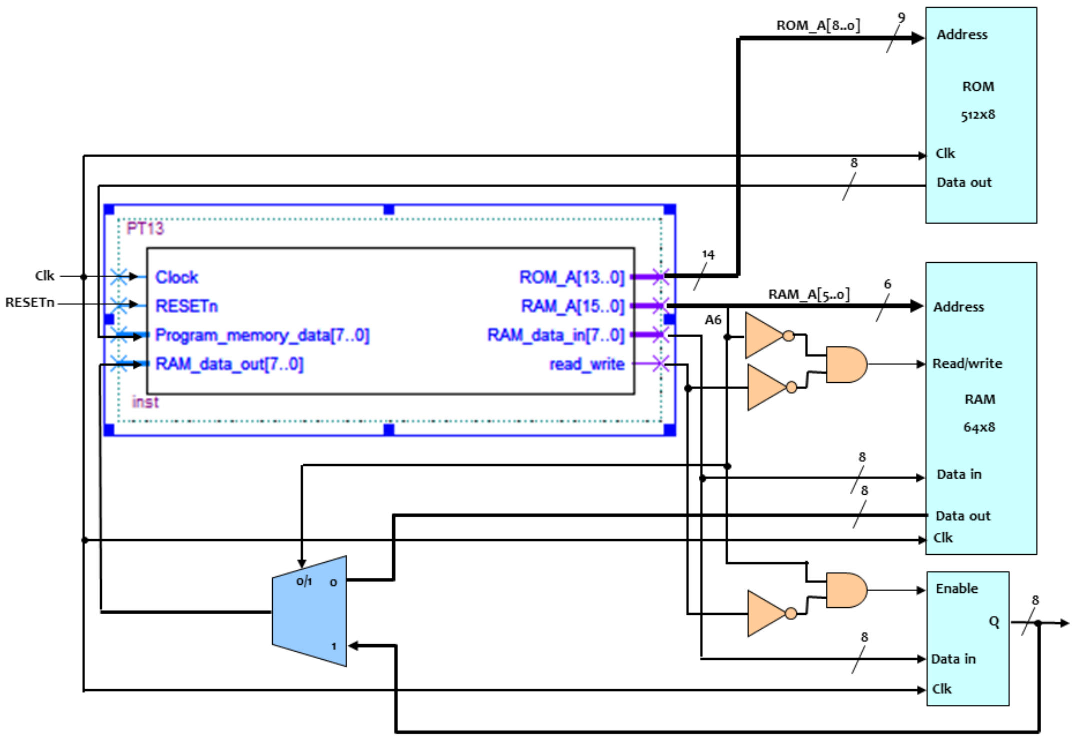

Figure 5 Read back of data.

Additional latches or a separate RAM or ROM memory area can be added by extending the address decoding.

5. Instruction Set

Arithmetic Instructions

ADD A, mm																$50, mm
PC								→								PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7+ mm7	a6+ mm6	a5+ mm5	a4+ mm4	a3+ mm3	a2+ mm2	a1+ mm1	a0+ mm0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Immediate Data $mm is added to Accumulator A contents and the result is written back into Accumulator A																

ADD B, mm																$51, mm
PC								→								PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7+ mm7	b6+ mm6	b5+ mm5	b4+ mm4	b3+ mm3	b2+ mm2	b1+ mm1	b0+ mm0
Immediate Data $mm is added to Accumulator B contents and the result is written back into Accumulator B																

ADD A,B																$54
PC								→								PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7+ b7	a6+ b6	a5+ b5	a4+ b4	a3+ b3	a2+ b2	a1+ b1	a0+ b0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Accumulator A contents are added to Accumulator B and the result is written back into Accumulator A																

SUB mm																$58 mm
PC								→								PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7-	a6-	a5-	a4-	a3-	a2-	a1-	a0-

									mm7	mm6	mm5	mm4	mm3	mm2	mm1	mm0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Immediate Data $mm is subtracted from Accumulator A and the result is written back into Accumulator A																

SUB B, mm																**$59, mm**
PC								→								PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7-mm7	b6-mm6	b5-mm5	b4-mm4	b3-mm3	b2-mm2	b1-mm1	b0-mm0
Immediate Data $mm is subtracted from Accumulator B contents and the result is written back into Accumulator B																

SUB A,B																**$5C**
PC								→								PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7-b7	a6-b6	a5-b5	a4-b4	a3-b3	a2-b2	a1-b1	a0-b0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Accumulator B contents are subtracted from Accumulator A and the result is written back into Accumulator A																

Logic Instructions

AND A, mm																$60, mm
PC								→								PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			0	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7 & mm7	a6 & mm6	a5 & mm5	a4 & mm4	a3 & mm3	a2 & mm2	a1 & mm1	a0 & mm0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Immediate Data $mm is logically ANDed with Accumulator A contents and the result is written back into Accumulator A																

AND B, mm																$61, mm
PC								→								PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			0	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7 & mm7	b6 & mm6	b5 & mm5	b4 & mm4	b3 & mm3	b2 & mm2	b1 & mm1	b0 & mm0
Immediate Data $mm is logically ANDed with Accumulator B contents and the result is written back into Accumulator B																

AND A,B																$64
PC								→								PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			0	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7 & b7	a6 & b6	a5 & b5	a4 & b4	a3 & b3	a2 & b2	a1 & b1	a0 & b0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Accumulator A is logically ANDed with Accumulator B and the result is written back into Accumulator A																

OR A, mm																$68, mm
PC								→								PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			0	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7 | mm7	a6 | mm6	a5 | mm5	a4 | mm4	a3 | mm3	a2 | mm2	a1 | mm1	a0 | mm0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0

Immediate Data $mm is logically ORed with Accumulator A contents and the result is written back into Accumulator A

OR B, mm																**$69, mm**
PC								→								PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			0	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7 | mm7	b6 | mm6	b5 | mm5	b4 | mm4	b3 | mm3	b2 | mm2	b1 | mm1	b0 | mm0
Immediate Data $mm is logically ORed with Accumulator B contents and the result is written back into Accumulator B																

OR A,B																**$6C**
PC								→								PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			0	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7 | b7	a6 | b6	a5 | b5	a4 | b4	a3 | b3	a2 | b2	a1 | b1	a0 | b0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Accumulator A is logically ORed with Accumulator B and the result is written back into Accumulator A																

XOR A, mm																**$70, mm**
PC								→								PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			0	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7 ^ mm7	a6 ^ mm6	a5 ^ mm5	a4 ^ mm4	a3 ^ mm3	a2 ^ mm2	a1 ^ mm1	a0 ^ mm0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Immediate Data $mm is logically XORed with Accumulator A contents and the result is written back into Accumulator A																

XOR B, mm																**$71, mm**
PC								→								PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			0	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7 ^ mm7	b6 ^ mm6	b5 ^ mm5	b4 ^ mm4	b3 ^ mm3	b2 ^ mm2	b1 ^ mm1	b0 ^ mm0

Immediate Data $mm is logically XORed with Accumulator B contents and the result is written back into Accumulator B

XOR A,B																$74
PC								→								PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			0	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7≠ b7	a6≠ b6	a5≠ b5	a4≠ b4	a3≠ b3	a2≠ b2	a1≠ b1	a0≠ b0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Accumulator A is logically XORed with Accumulator B and the result is written back into Accumulator A																

LSL A																$78
PC								→								PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			a7	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a6	a5	a4	a3	a2	a1	a0	0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Accumulator A is logically shifted left. The LSB becomes logic zero and a7 (MSB) is moved top the carry bit.																

LSL B																$79
PC								→								PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			b7	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a6	a5	a4	a3	a2	a1	a0	0
b7	b6	b5	b4	b3	b2	b1	b0	→	b6	b5	b4	b3	b2	b1	b0	0
Accumulator B is logically shifted left. The LSB becomes logic zero and b7 (MSB) is moved to the carry bit.																

LSR A																$7C
PC								→								PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			a0	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	0	a7	a6	a5	a4	a3	a2	a1
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Accumulator A is logically shifted right. The MSB becomes logic zero and a0 (LSB) is moved to the carry bit.																

LSR B																$7D
PC								→								PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			b0	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a6	a5	a4	a3	a2	a1	a0	0
b7	b6	b5	b4	b3	b2	b1	b0	→	0	b7	b6	b5	b4	b3	b2	b1

Accumulator B is logically shifted right. The MSB becomes logic zero and b0 (LSB) is moved to the carry bit.

ASR A																$80
PC								→								PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			a0	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a7	a6	a5	a4	a3	a2	a1
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0

Accumulator A is arithmetically shifted right. The MSB remains a7 value and a0 (LSB) is moved to the carry bit.

ASR B																$81
PC								→								PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			b0	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b7	b6	b5	b4	b3	b2	b1

Accumulator B is logically shifted right. The MSB remains b7 value and b0 (LSB) is discarded.

ROL A																$84
PC								→								PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			a7	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a6	a5	a4	a3	a2	a1	a0	Ca
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0

Accumulator A is rotated left through carry. The LSB becomes the previous carry bit and a7 (MSB) is moved into the carry bit.

ROL B																$85
PC								→								PC + 1

dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			b7	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a6	a5	a4	a3	a2	a1	a0	0
b7	b6	b5	b4	b3	b2	b1	b0	→	b6	b5	b4	b3	b2	b1	b0	Cb

Accumulator B is rotated left through carry. The LSB becomes the previous carry bit and b7 (MSB) is moved to the carry bit.

ROR A																$88
PC								→								PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			a0	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	Ca	a7	a6	a5	a4	a3	a2	a1
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0

Accumulator A is rotated right through carry. The MSB becomes the previous carry bit and a0 (LSB) is moved into the carry bit.

ROR B																$89
PC								→								PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			b0	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a6	a5	a4	a3	a2	a1	a0	0
b7	b6	b5	b4	b3	b2	b1	b0	→	Cb	b7	b6	b5	b4	b3	b2	b1

Accumulator B is rotated right through carry. The MSB becomes the previous carry bit and b7 (MSB) is moved to the carry bit.

Branch Instructions

BCC A,Addr																$E0 Addr
PC								→	If Ca = 0, PC = Addr else PC = PC + 3							
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Program counter is loaded with the value ‘Addr’ if Ca = 0.																

BCC B,Addr																$E1 Addr
PC								→	If Cb = 0, PC = Addr else PC = PC + 3							
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Program counter is loaded with the value ‘Addr’ if Cb = 0.																

BCS A,Addr																$E4 Addr
PC								→	If Ca = 1, PC = Addr else PC = PC + 3							
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Program counter is loaded with the value ‘Addr’ if Ca = 1.																

BCS B,Addr																$E5 Addr
PC								→	If Cb = 1, PC = Addr else PC = PC + 3							
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Program counter is loaded with the value ‘Addr’ if Cb = 1.																

BEQ A,Addr																**$E8 Addr**
PC								→	**If Za = 1, PC = Addr else PC = PC + 3**							
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Program counter is loaded with the value ‘Addr’ if Za = 1, (Accumulator A = $00).																

BEQ B,Addr																**$E9 Addr**
PC								→	**If Zb = 1, PC = Addr else PC = PC + 3**							
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Program counter is loaded with the value ‘Addr’ if Zb = 1, (Accumulator B = $00).																

BNE A,Addr																**$EC Addr**
PC								→	**If Za = 0, PC = Addr else PC = PC + 3**							
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Program counter is loaded with the value ‘Addr’ if Za = 0, (Accumulator A ≠ $00).																

BNE B,Addr																**$ED Addr**
PC								→	**If Zb = 0, PC = Addr else PC = PC + 3**							
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Program counter is loaded with the value ‘Addr’ if Zb = 0, (Accumulator B ≠ $00).																

BRA,Addr			**$FC Addr**
PC	→		**PC = Addr**

dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Program counter is loaded with the value ‘Addr’.																

BSR,Addr																$FD Addr
PC								→	PC = Addr Stack Pointer register loaded with previous PC contents							
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Stack pointer register is loaded with PC contents before Program counter is loaded with the value ‘Addr’.																

RTS																$FE
PC								→	PC = Stack Pointer register							
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Program counter is loaded with the Stack Pointer Register value.																

Load and Store Instructions

LDI A,mm																$10,mm
PC								→								PC = PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			0	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	mm7	mm6	mm5	mm4	mm3	mm2	mm1	mm0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Accumulator A is loaded with Immediate data $mm.																

LDI B,mm																$11,mm
PC								→								PC = PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			0	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	mm7	mm6	mm5	mm4	mm3	mm2	mm1	mm0
Accumulator B is loaded with Immediate data $mm.																

LDI DP,mm																$12,mm
PC								→								PC = PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	mm7	mm6	mm5	mm4	mm3	mm2	mm1	mm0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
The Data Page Register is loaded with Immediate data $mm.																

LDD A,of																$18,of
PC								→								PC = PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			0	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	(DP, of)7	(DP, of)6	(DP, of)5	(DP, of)4	(DP, of)3	(DP, of)2	(DP, of)1	(DP, of)0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Accumulator A is loaded with the contents of Read/Write memory at the 14 bit address location, (Data Page Register[5..0], of[7..0]).																

LDD B,of																$19,of
PC								→								PC = PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			0	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	(DP, of)7	(DP, of)6	(DP, of)5	(DP, of)4	(DP, of)3	(DP, of)2	(DP, of)1	(DP, of)0
Accumulator B is loaded with the contents of Read/Write memory at 14 bit address location, (Data Page Register[5..0], of[7..0]).																

LDRI																$1C
PC								→	PC = PC + 1							
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			0	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	(DP, B)7	(DP, B)6	(DP, B)5	(DP, B)4	(DP, B)3	(DP, B)2	(DP, B)1	(DP, B)0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
Accumulator A is loaded with the contents of Read/Write memory at 14 bit address location, (Data Page Register[5..0], Accumulator B)																

STR A,of																$24,of
PC								→								PC = PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
The contents of Accumulator A are written into Read/Write memory at the 12 bit address location, (Data Page Register[3..0], of[7..0]).																

STR B,of																$25,of
PC								→								PC = PC + 2
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
The contents of Accumulator B are written into Read/Write memory at the 12 bit address location, (Data Page Register[3..0], of[7..0]).																

Control Instructions

NOP																$00
PC								→								PC = PC + 1
dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0	→	dp7	dp6	dp5	dp4	dp3	dp2	dp1	dp0
		Cb	Zb			Ca	Za	→			Cb	Zb			Ca	Za
a7	a6	a5	a4	a3	a2	a1	a0	→	a7	a6	a5	a4	a3	a2	a1	a0
b7	b6	b5	b4	b3	b2	b1	b0	→	b7	b6	b5	b4	b3	b2	b1	b0
No operation. Program counter is incremented by one.																

6. Using the Editor and Assembler

To assist in the writing of code an editor and assembler are provided.

The ConTEXT editor can be downloaded from the PT13 webpage, http://www.singmai.com/IP_Cores/PT13.htm), or from the editor website (http://www.contexteditor.org/).

Just follow the instructions on the ConTEXT website to install the editor. Once installed the highlighter file, PT13_assembler.chl, for the PT13 assembler should also be downloaded from the SingMai webpage and copied into the ConTEXT/highlighters directory. When the editor is run, click on File/New to open a new text file and then select Tools…Set Highlighter where you should see PT13 Assembler. Once selected the instruction codes and keywords of the PT13 assembler will be highlighted in colour to assist in the writing of code. For further information on the editor features either visit the ConTEXT website or click on Help…Contents in the editor.

The assembler is called as13 and is run in a Command Prompt window. A command prompt can be opened by typing 'cmd' in the 'Start'...'Run' prompt or a short cut can be setup that can be made to automatically start in your directory. The as13 assembles the PT13 assembly code and outputs the binary PT13 machine code. The command line syntax is:

as13 [options] filename

where filename is the input filename containing the program to assemble. It is usually convenient to keep the assembler executable (as13.exe) and the file to be assembled in the same directory.

The default output file for program memory defaults to as13text.mif which can be used directly by the Quartus FPGA compiler for the ROM program memory contents. Alternatively the as13 can output a binary file.

The as13 will not stop assembling the input file when it finds an error. Instead it will list all errors found in the whole assembly file and it will produce some reasonable guess at the desired output machine code.

Command Line Options

The as13 has the following options:

- -EB Change to big endian. This is the default.
- -EL Change to little endian.
- -O type Define output type. "-O binary" selects binary output format. "-O mif" selects Memory Initialization File (MIF). MIF is the default.
- -o base Place output in "base"text.bin and "base"data.bin. Default is "as13"
- -s ram=size Define RAM size. Default is 8192 bytes.
- -s rom=size Define ROM size. Default is 16384 bytes.
- -v level Change verbosity level. Default is 0. if level > 0, debug information is output to file dbg.txt.
- -version Display version

The default endianness of output is big endian. This only affects opcode parameters that are larger than a byte (for example, the bra opcode uses int16).

WARNING: when using -O binary, the endianness of the binary file is not affected by the -EL/-EB feature. The endianness of the binary .bin file will match the endianness of the computer where as13 is run. Since most users will be using WinTel, the endianness of the .bin file will be little endian.

Synthetic Opcodes

"Synthetic opcodes" are opcodes supported by the PT13 assembler but these opcodes are assembled into a different PT13 opcode equivalent. Synthetic opcodes are listed here with their equivalent PT13 opcode.

"CLR A" clears accumulator A. It is assembled as "LDI A,0".
"CLR B" clears accumulator B. It is assembled as "LDI B,0".
"INC A" increments accumulator A. It is assembled as "ADD A,1".
"INC B" increments accumulator B. It is assembled as "ADD B,1".
"DEC A" decrements accumulator A. It is assembled as "SUB A,1".
"DEC B" decrements accumulator B. It is assembled as "SUB B,1".

Syntax Rules

1. Each opcode and all of its parameters MUST be on one line. A "line" is terminated by a newline.
2. Upper and lower case are equivalent in register names, opcodes, condition codes and assembler directives. Upper and lower case are significant only in labels.
3. White space is ignored (white space includes tabs, spaces, blank lines, etc). Remember rule #1.
4. Maximum line length is 1022 characters. Longer lines will give undefined results.
5. ; is the comment character. C++ comments (//) are also accepted. C-style comments /* */ are not valid.
6. The registers are referred to with the letters assigned to them by PT13 documentation. For example, A, B, and DP.
7. Maximum length of an opcode, register, number, or label (including ‘:’) is 254 characters. Longer names will give undefined results.

Integers

A binary integer is `0b' or `0B' followed by zero or more of the binary digits `01'.
An octal integer is `0' followed by zero or more of the octal digits (`01234567').
A decimal integer starts with a non-zero digit followed by zero or more digits (`0123456789').
A hexadecimal integer is `0x', `0X', or '$' followed by one or more hexadecimal digits chosen from `0123456789abcdefABCDEF'.

Labels

Labels can be used in place of all values that are larger than a byte. Currently only the branch opcodes allow such values. Therefore, they are the only op-codes that allow the use of labels.

Labels must be at least 1 character long. They must start with an alphabetic charcter a-z, A-Z, or underscore (_). The remaining characters must use an alphabetic character a-z, A-Z, underscore (_), or any digit 0 through 9.

When labelling a line, the label must be followed immediately by colon (:). Labelled lines CANNOT have an op-code or assembler command (.something) after the colon. Only comments and white space are allowed.

Labels label the next op-code line. When referring to the label, do not use the colon.

Labels cannot use the same string as any op-code or register name.

The maximum number of labels that can be used in a single program is 16383. Using more labels will give undefined results.

Assembler Directives

Symbols beginning with a dot `.' are assembler directives. Assembler directives CANNOT have an op-code or label on the same line. Only comments and whitespace are allowed after the assembler directive. The following are the current assembler directives:

.byte number
Tells as13 to emit one byte of value "number".

.data
Tells as13 to assemble the following statements onto the end of the text section.

.org new-lc, fill
Advance the location counter of the current section (text or data) to new-lc. new-lc must be a number. You can't use .org to cross sections: if new-lc is too large, the .org directive will give a warning and will be ignored. new-lc cannot be a label, it must be a known number.

.org may only increase the location counter, or leave it unchanged; you cannot use .org to move the location counter backwards. If moving backwards, the .org directive will give a warning and will be ignored.

When the location counter is advanced, the intervening bytes are filled with "fill" which must be a number. "fill" is NOT optional.

.skip size, fill
Tells as13 to emit "size" bytes of value "fill". "fill" is NOT optional. .skip and .space are identical features.

.space size, fill
Tells as13 to emit "size" bytes of value "fill". "fill" is NOT optional. .skip and .space are identical features.

.text
Tells as13 to assemble the following statements onto the end of the text section. .text is the default section.

.word number
Tells as13 to emit one 16-bit (2 bytes) of value "number". Byte order is determined by -EB or -EL option to as13.

C Preprocessor

The as13 **DOES NOT** support C preprocessing. If C preprocessing is desired, the user must perform this step outside of as13.

as13 **DOES NOT** support C arithmetic or logical constant expressions like "5 * 6", "0x1 | 0x4", or "0x1 << 5". Therefore, if C preprocessor does not resolve these into a number, as13 will give a syntax error.

7. Example Code

The following is a simple piece of code to generate a 3 second delay (at 27MHz clock). External memory is used for one delay constant because of the limitations of two 8 bit accumulators.

```
welcome_delay:
  ldi  dp,$00                    ; Point to memory
  ldi  a,$23                     ; Delay time (third loop)
  str  a,$01                     ; Store in memory
w_delay_3:
  ldi  a,$FF                     ; Second delay loop
w_delay_1:
  ldi  b,$FF                     ; First delay loop
w_delay_2:
  sub  b,$01
  bne  b,w_delay_2
  sub  a,$01
  bne  a,w_delay_1
  ldd  a,$01
  sub  a,$01
  str  a,$01
  bne  a,w_delay_3
```

This code example is used to write an instruction to a generic 8x2 line LCD character display.

```
; LCD display control functions
; LCD_control latch: Bit 2 = RS; Bit 1 = RW; Bit 0 = EN

Function_set:
  ldi  dp,$02                    ; Point to LCD display
  str  a,$01                     ; Store instruction in data register
  ldi  b,$00                     ; Set RS=0, RW=0, EN=0
  str  b,$00                     ; Store instruction
  ldi  b,$01                     ; Set RS=0, RW=0, EN=1
  str  b,$00
  ldi  b,$00                     ; Set RS=0, RW=0, EN=0
  str  b,$00
  ldi  a,$10                     ; wait >4.1ms
FS_delay2:
  ldi  b,$FF
FS_delay1:
  sub  b,$01                     ; delay = 1.185us * AccB
  bne  b,FS_delay1               ; loop until AccA/B=$00
  sub  a,$01
  bne  a,FS_delay2
  rts

LCD_instruction_write:
  ldi  dp,$02                    ; Point to LCD display
  str  a,$01                     ; Store instruction in data register
  ldi  b,$00                     ; Set RS=0, RW=0, EN=0
  str  b,$00                     ; Store instruction
  ldi  b,$01                     ; Set RS=0, RW=0, EN=1
  str  b,$00
  ldi  b,$00                     ; Set RS=0, RW=0, EN=0
  str  b,$00
  ldi  b,$02                     ; Set RS=0, RW=1, EN=0
  str  b,$00
```

```
    rts

  busy:                          ; Do not use AccB
    ldi  dp,$02                  ; Point to LCD display
    ldi  a,$03                   ; Set RS=0, RW=1, EN=1
    str  a,$00                   ; Write to control register
  read_busy:
    ldd  a,$01                   ; Read status
    and  a,$80                   ; Busy flag = bit 7
    bne  a,read_busy
    ldi  a,$02                   ; Set RS=0, RW=1, EN=0
    str  a,$00                   ; Write to control register
    rts

  display_character:             ; Do not use AccB
    ldi  dp,$02                  ; Point to LCD display
    str  a,$01                   ; Store character (AccA) in LCD data register
    ldi  a,$04                   ; Set RS=1, RW=0, EN=0
    str  a,$00                   ; Write to control register
    ldi  a,$05                   ; Set RS=1, RW=0, EN=1
    str  a,$00                   ; Write to control register
    ldi  a,$06                   ; Set RS=1, RW=1, EN=0
    str  a,$00                   ; Write to control register
    rts
```

Further coding examples can be downloaded from the SingMai website, http://www.singmai.com/IP_Cores/PT13.htm.