DTMF Receiver Core Module Design Description

Design Description

TDSP MANUAL

December 2014
Preliminary

Other Trademarks

DTMF Receiver Overview 1
Block Descriptions 5
Serial Port Interface (SPI) 6
DMA Controller (DMA) 7
Memory Access Bus Arbiter (ARB) 7
u-Law PCM to Linear PCM (ULAW_LIN_CONV) 8
Tiny DSP (TDSP) 8
TDSP Data Strobe and Chip Select (TDSP_DS_CS) 12
Results Character Conversion (RESULTS_CONV) 12
ASCII Digit Register 13
Memory MAP 13
TDSP Instruction Set 15
TDSP Assembler 29
Source Statement Syntax 29
Define Assembly Time Constant Attribute 29
Constants 29
Initialize Word Attribute 29
Absolute Origin Attribute 29
Predefined Symbols and Abbreviations 29

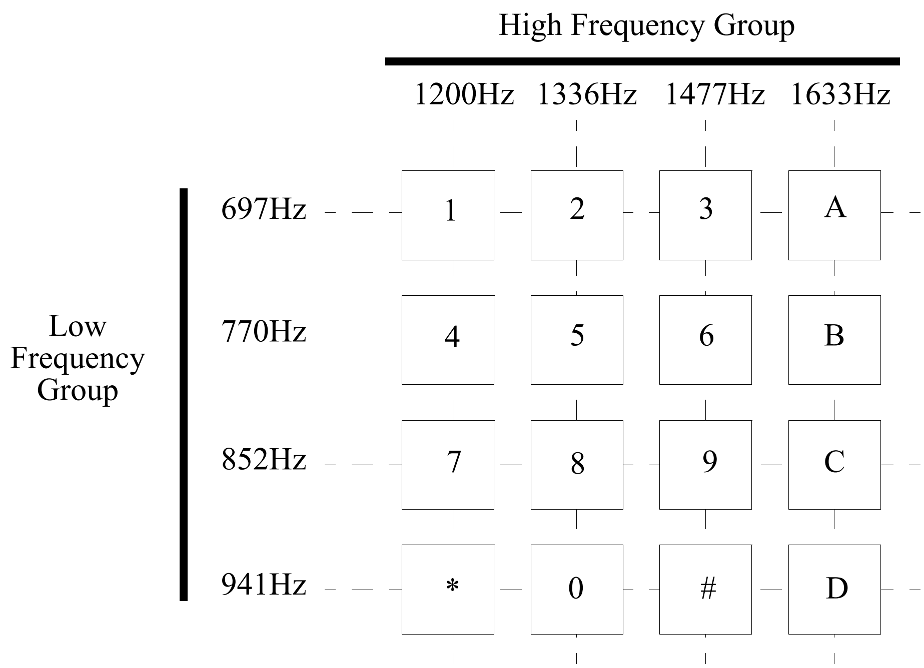
1

DTMF Receiver Design Description

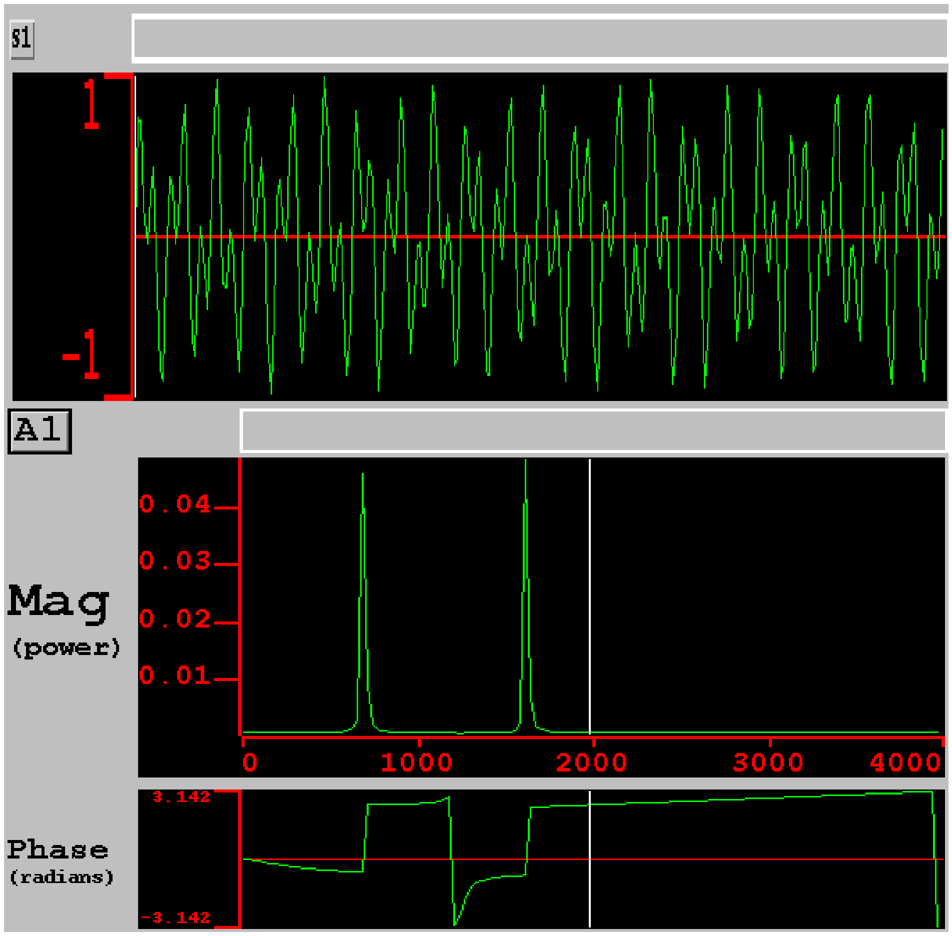
DTMF Receiver Overview

In a telephone network, two basic techniques are used for transmitting information between network entities: in-band and common channel signalling. In-band signalling shares the transmission facility for signalling and voice data. Common channel signalling uses one transmission facility for all signalling functions for a group of voice channels.

One common form of in-band signalling is dual tone multifrequency, or DTMF. DTMF signals are commonly generated by “Touch-Tone” telephones; most of us probably have this type of telephone in our homes today. Here is a layout of a standard DTMF keypad:



Notice that keys “A”, “B”, “C”, and “D” are not usually on telephones for home use. They are mainly used in commercial applications with special instruments. Pressing a key will cause the telephone to generate the indicated pair of tones, one from the high frequency group, and one from the low frequency group.



Above is a DTMF signal along with its frequency response.

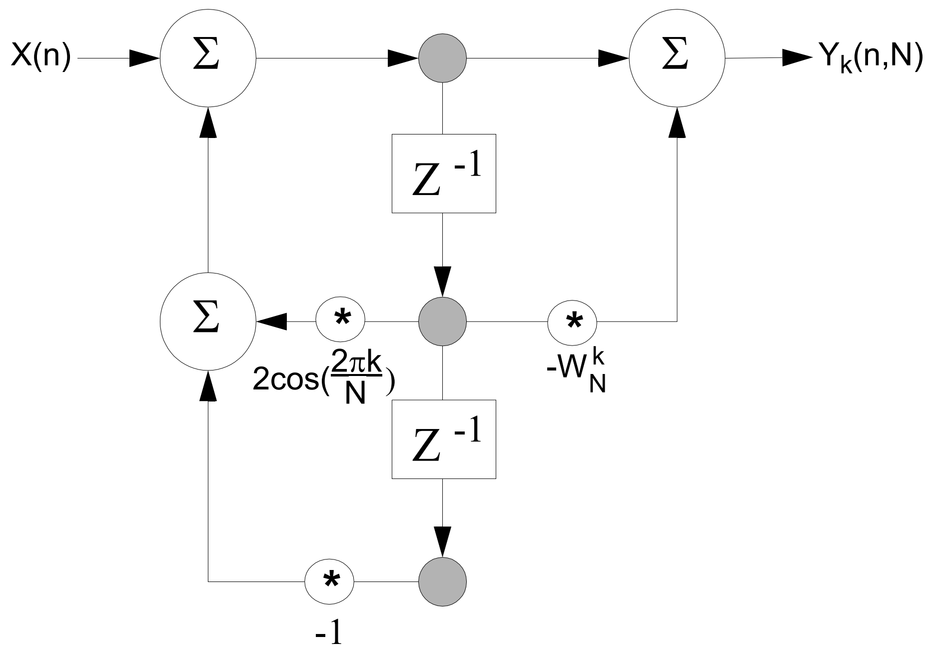
Telephone specifications, such as *Touch-Tone Calling - Requirements for Central Office (*AT&T Compatibility Bulletin No. 105, August 8, 1975) define a DTMF digit as follows:

- A pair of tones, one from the low frequency group, one from the high frequency group
- A nominal level, per frequency, of -6 dBm0
- A maximum rate for DTMF signalling of10 digits per second (or typically 100mS per digit)
- A DTMF digit must be present for at least 45 mS
- An inter-digit “quiet period” must exist between digits for at least 45 mS
- Upon reception, the signal difference between the low frequency tone and high frequency tone does not exceed 8 dB
- Upon reception, the signal difference between the high frequency tone and low frequency tone does not exceed 4 dB

A simple DTMF detector could be built using a group of band-pass filters, each followed by a peak detector, which has a natural time constant of about 35 mS. The output of the peak detector would drive a threshold comparator, which in turn would drive a decision logic circuit. Get the picture? A bunch of analog circuits that would probably require “tweaking” on your assembly line.

To detect the tones, our DTMF receiver will utilize a modified discrete Fourier transform (DFT) algorithm known as the Goertzel algorithm. The Goertzel algorithm is a very efficient way of calculating a partial frequency spectrum using a second-order recursive computation (calculation of the DFT is known as the “direct form”). Indeed, we are only really interested in calculating the frequency response at the DTMF center frequencies. Because of the recursive nature of the algorithm, the Goertzel algorithm will run entirely in firmware on the Tiny Digital Signal Processor (TDSP).

Here is a flow graph of the Goertzel algorithm:

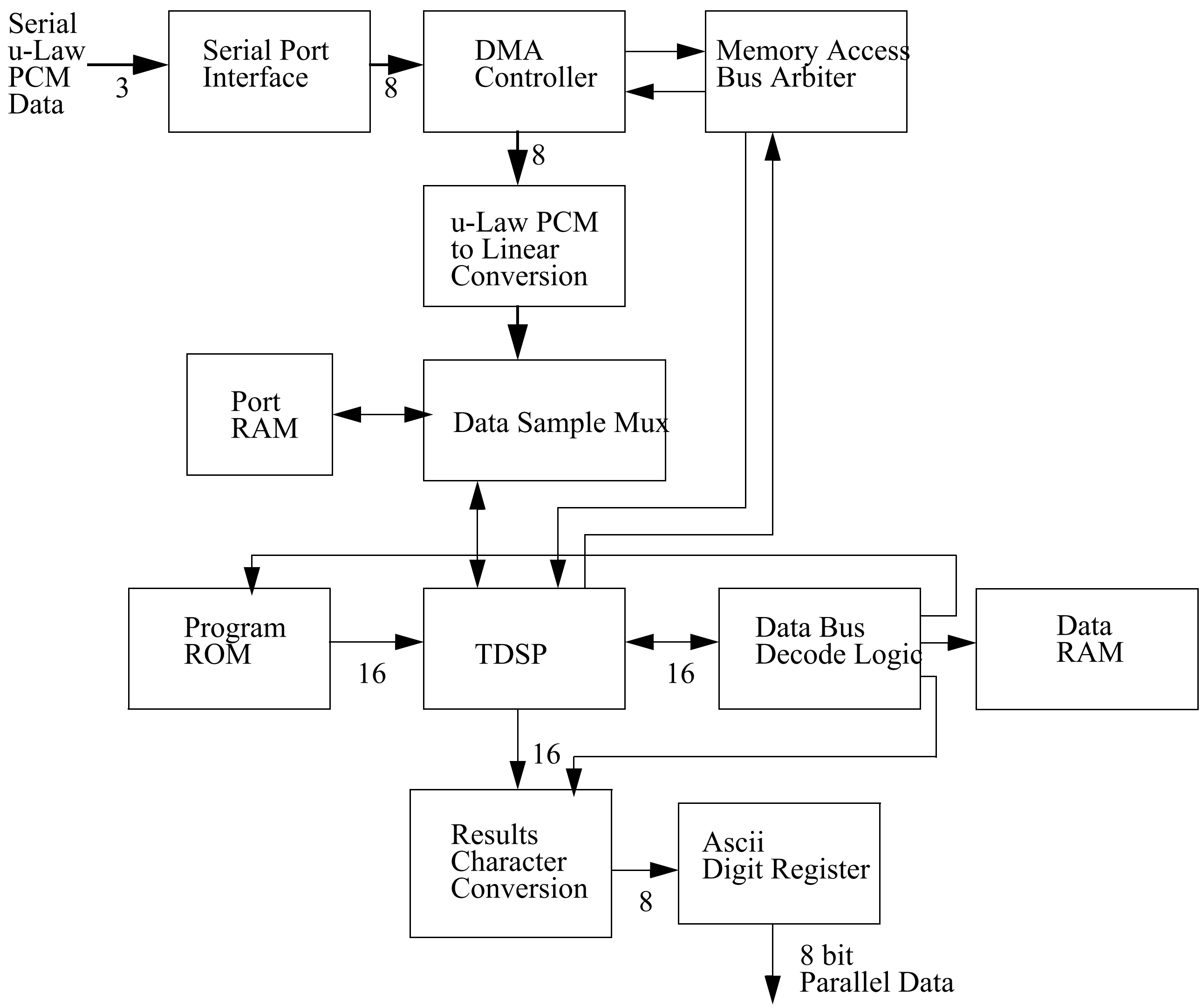


As suggested, the Goertzel algorithm takes the form of a second order infinite-impulse-response (IIR) filter. For spectral analysis, the only interesting calculation is the last iteration (N-1) of the algorithm. At this point $Y_K(N) = X(K)$, the DFT response. What may not be readily apparent is that only the left half of the graph is calculated for most of the input samples ($0 <= n <= N-2$). When $n = N-1$, then both half's of the graph are calculated. Since $W^k{}_N$ is a complex number, complex multiplication is only required once per algorithm iteration.

Once calculated, the sample window frequency response must be analyzed to determine what, if any, DTMF digit was found. This processing will take place using a Finite State Machine (FSM) module. All DTMF parameters are checked as defined except the twist check, which we've relaxed to +/- 12 dB for simplicity to a simple shift function. Obviously the astute student could easily modify the supplied block to perform the forward and reverse twist checks as specified.

Block Descriptions

Here is a block diagram of the major laboratory design database, the DTMF Receiver.



The following sections give a quick overview of each of the major blocks.

Serial Port Interface (SPI)

The serial port interface accepts u-law compressed PCM data, serialized LSB first, and reformats the data to byte orientation. The interface uses a clock signal to strobe the data on the signal's rising edge. A frame strobe is also used to indicate the start of a new data sample.

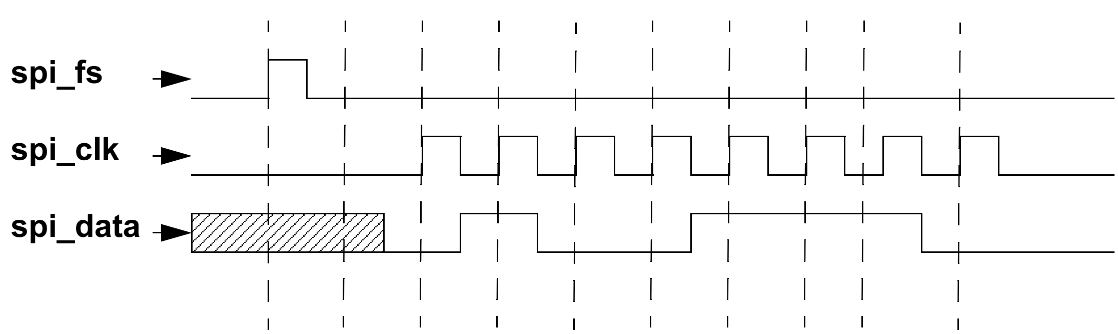
Once a character is received, the SPI signals the DMA controller that a new byte is ready to be moved to the Data Sample memory.

SPI will be coded as an explicit state machine.

Signal interface:

spi_clk - serial data clock input
spi_fs - serial data frame strobe input
spi_data - serial data input
clk - system clock input
reset - system reset input
dout[7:0] - parallel data output
read - parallel data output enable input
dflag - new data flag output
scan_input_1 - scan data input (chain 1)
scan_input_2 - scan data input (chain 2)
scan_enable - scan enable input
scan_output_1 - scan data output (chain 1)
scan_output_2 - scan data output (chain 2)

The serial interlace timing is represented in the following diagram:



DMA Controller (DMA)

The direct memory access (DMA) controller coordinates byte data movement between the SPI and Data Sample memory. Data transfer is initiated via the SPI. The DMA controller then attempts a data transfer by requesting access to the Data Sample memory via the Bus Arbiter. Once access is granted, the data sample byte is written to the data sample RAM.

The DMA Controller will maintain two contiguous buffers (in the same RAM) and provide TDSP with an indication of which buffer is currently being filled.

DMA will be coded as an implicit state machine.

Signal interface:

clk - system clock input
reset - system reset input
read_spi - SPI parallel data output enable output
dflag - SPI new data flag input
breq - memory bus request output
bgrant - memory bus grant input
a[7:0] - address bus output
as - data address strobe output
write - data write strobe output
top_buf_flag - top buffer flag
scan_input - scan data input
scan_enable - scan enable input
scan_output - scan data output

Memory Access Bus Arbiter (ARB)

The memory access bus arbiter (ARB) coordinates DMA and TDSP access to the Data Sample memory. The protocol is a simple REQUEST, GRANT scheme. Note that the arbiter is biased to allow TDSP priority access if both devices request at the same time.

ARB is coded as an explicit state machine.

Signal interface:

clk - system clock input
reset - system reset input
dma_breq - DMA bus request input
dma_bgrant - DMA bus grant output
tdsp_breq - TDSP bus request input

tdsp_bgrant - TDSP bus grant output
scan_input - scan data input
scan_enable - scan enable input
scan_output - scan data output

u-Law PCM to Linear PCM (ULAW_LIN_CONV)

This block expands the u-Law compress PCM samples to linear PCM samples. The u-Law compression/ expansion mechanism is specified in CCITT standard G.711.

Signal interface:

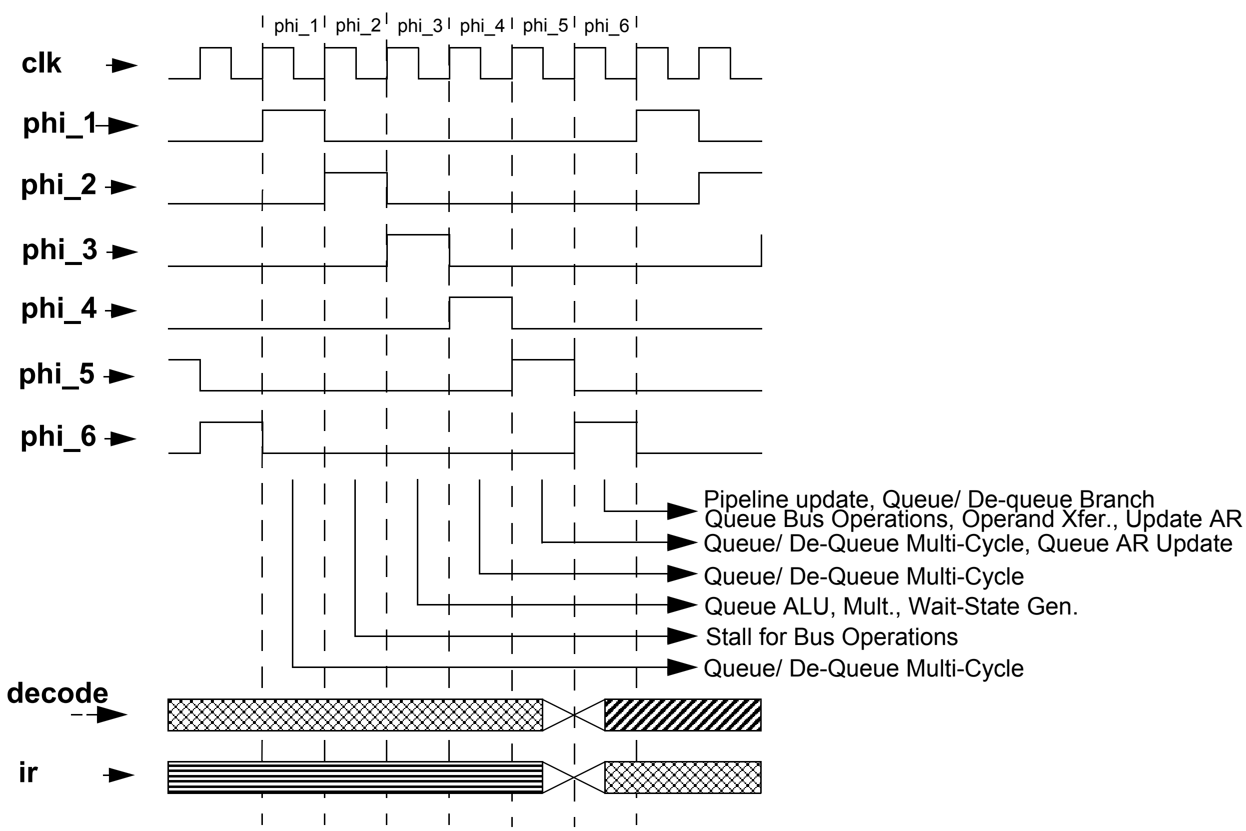
upcm[7:0] - u-Law compressed PCM input

lpcm[15:0]- linear PCM output

Tiny DSP (TDSP)

The Tiny Digital Signal Processor (DSP) mimics the instruction set of the TMS320 family of DSP's (actually its very close in functionality to the TMS32010, with a MAC instruction and bus arbiter interface).

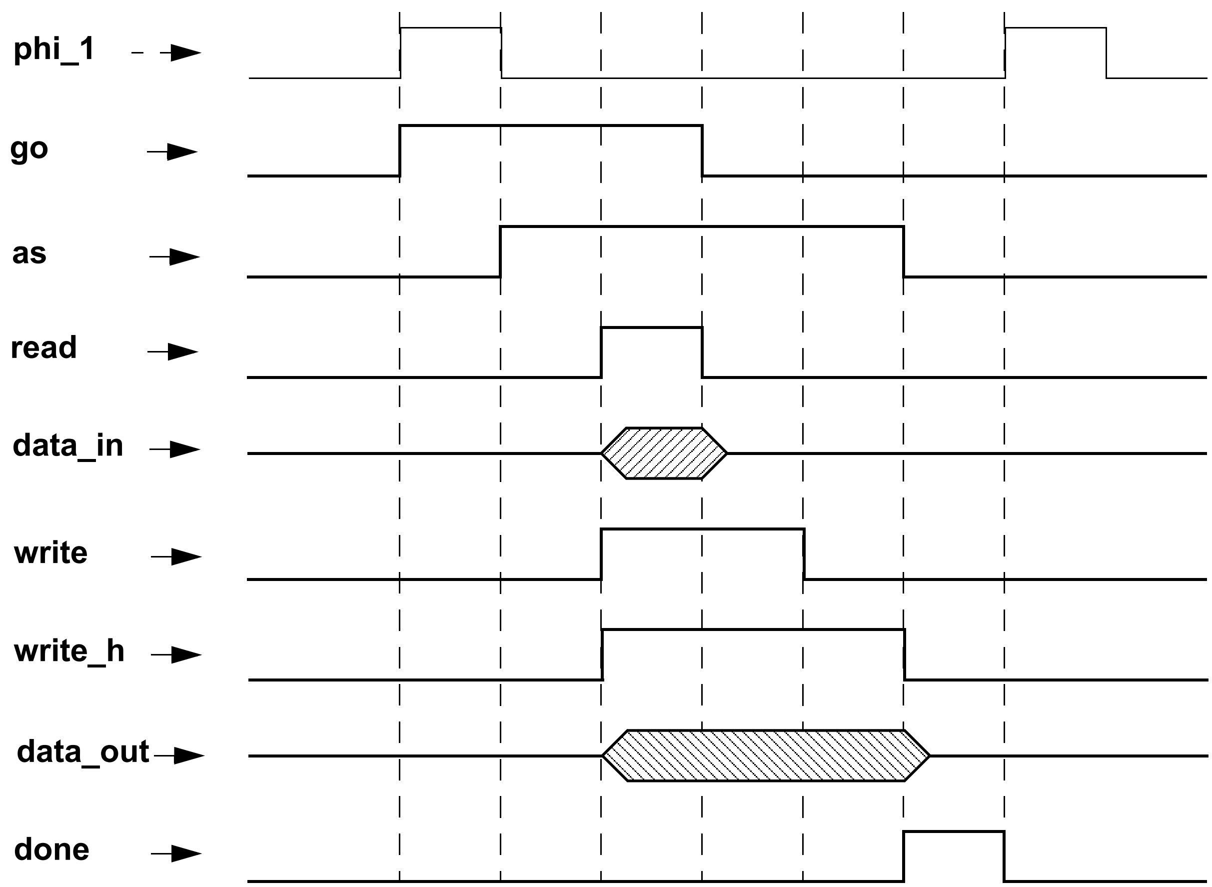
The instruction pipeline can be represented in the following timing diagram:



Signal interface:

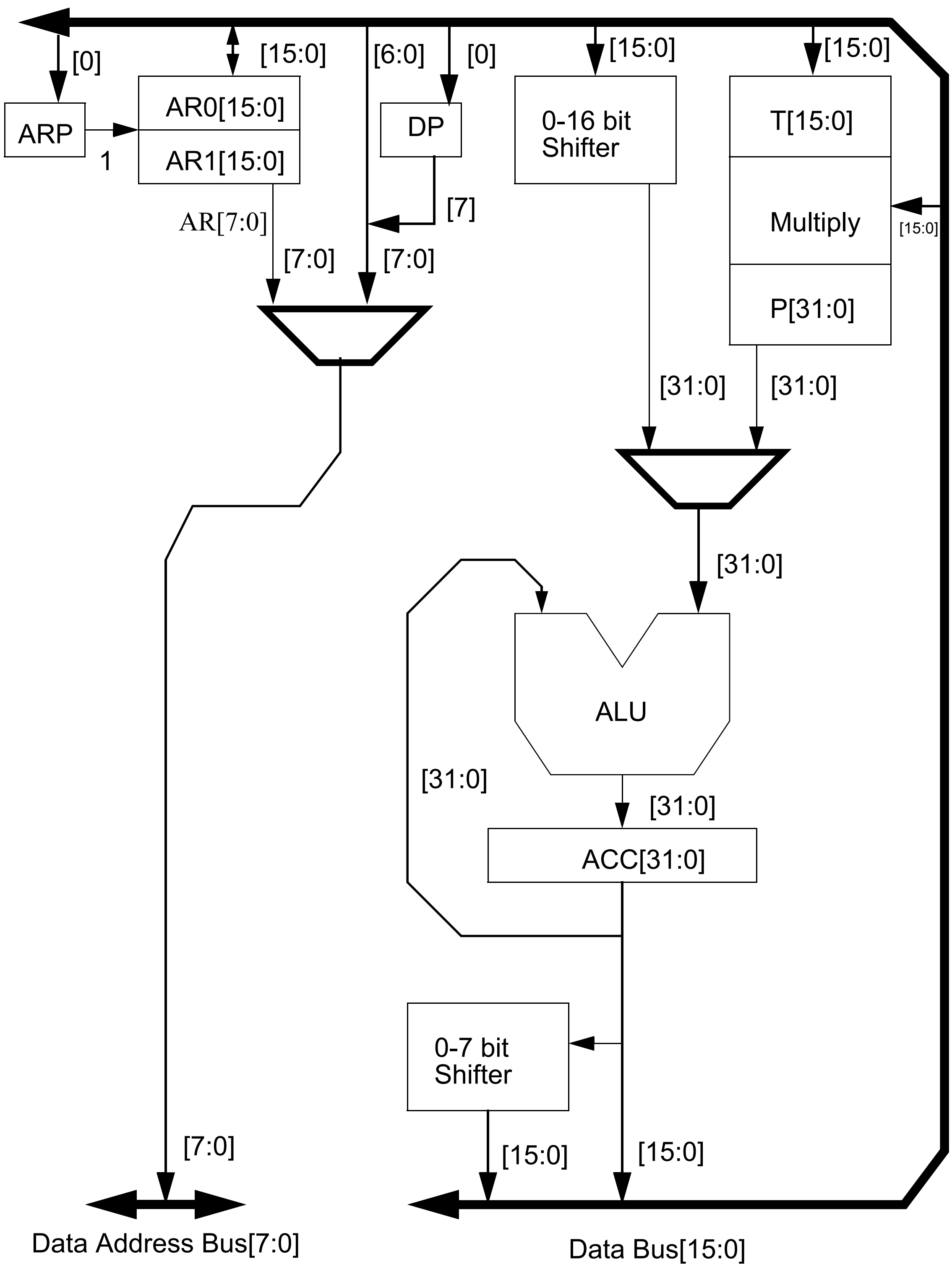
clk - system clock input
reset - system reset input
read - data read output
write - data write output
address[7:0] - data address bus output
data[15:0] - data bus
p_read - program read output
p_write - program write output
p_address[7:0] - program address bus
p_data[15:0] - program data bus
scan_input - scan data input
scan_enable - scan enable input
scan_output - scan data output

Here's a quick diagram representing the TDSP read and write cycle timing:



The tdsp data flow is shown in the following block diagram

TDSP Data Flow Diagram



TDSP Data Strobe and Chip Select (TDSP_DS_CS)

This block generates all the bus interface control signals for the DTMF core.

Signal interface :

address - data bus address input
write - data bus write input
read - data bus read input
reset - system reset input
as - data address strobe input
port_as - port bus address strobe input
port_address - port bus address inpout
port_write - port bus write input
port_read - port bus read input
top_buf_flag - top buffer flag input
t_write_ds - write data sample memory output
t_read_ds - read data sample memory output
t_write_d - write data memory output
t_read_d - read data memory output
t_write_rcc - write results character converter output
t_address_ds - address data sample memory output
bus_request_in - bus request input
bus_grant_in - bus grant input
bus_request_out - bus request output
bus_grant_out - bus grant output

Results Character Conversion (RESULTS_CONV)

Once the TDSP has completed the calculation of the signal spectrum, the results are written in block format to the “Results Character Conversion” (RCC) block. Once a block is written, the resulting spectrum is analyzed for DTMF digit content. If a digit is found, the resolved ASCII character representation is written to the Results circular buffer. Once a valid digit sequence is processed, the ASCII character is moved to the ASCII digit register for collection by the host.

RCC is coded as an implicit state machine.

Signal interface:

clk - system clock input
reset - system reset input

address[3:0]- address input
din[7:0]- data input
din_write - data input write input
dout[7:0]- data output
dout_write - data output write output
scan_input_1 - scan data input (chain 1)
scan_input_2 - scan data input (chain 2)
scan_enable - scan enable input
scan_output_1 - scan data output (chain 1)
scan_output_2 - scan data output (chain 2)

ASCII Digit Register

The ASCII digit register is simply a nine (9) bit register for holding the current 8 bit signal character, plus a one bit toggle flag.

Upon reset, the digit holding register is set to 0xff, and the flag is set to 1.

Signal interface:

reset - system reset input
clk - system clock input
digit_in[7:0]- digit input
digit_out[7:0]- digit output
flag_in - digit flag input
flag_out - digit flag output
scan_input - scan data input
scan_enable - scan enable input
scan_output - scan data output

Memory MAP

(data space)

0x00 - 0xff-> tdsp program memory (256 bytes)

0x00 - 0x7f-> data sample memory (128 words)

0x80 - 0xdf-> data scratch memory (96 words)

0xe0 - 0xef-> results character conversion (16 words)

(port space)

0x00 - 0x07-> misc. control (8 words)

0x00-> select dma to generate address bit 7

0x01-> select tdsp to generate address bit 7

0x02-> tdsp select lower data sample buffer

0x03-> tdsp select upper data sample buffer

TDSP Instruction Set

Addressing mode notes:

Direct Addressing Mode - Direct addressing forms the data memory address by concatenating seven bits of the instruction word with the data page pointer. This implements a paging scheme in which each page contains 128 words.The physical address is built by appending the immediate address with the current data page pointer, for example:

```
{DP, OPCODE[6:0]}
```

Indirect Addressing Mode - Indirect addressing forms the data memory address from the least significant eight bits of one of the two auxiliary registers, AR0 or AR1. The auxiliary register pointer (ARP) selects the current auxiliary register for indirect address generation. The auxiliary registers can automatically post increment or post decrement in parallel with the execution of any indirect instruction to permit single-instruction-cycle manipulation of data structures in memory. Specific support for indirect addressing is included in the assemble as:

```
*     address AR(ARP)
*+    address AR(ARP), post increment AR(ARP)
*-    address AR(ARP), post decrement AR(ARP)
```

Immediate Addressing Mode - Immediate instructions derive data from part of the instruction word rather from the data RAM. This can be thought of as a shorthand for loading constants to certain registers. Note that the typical immediate data size is an 8 bit constant, although certain instructions can handle lager constants. For reference, most immediate data instruction opcodes end in “k”.

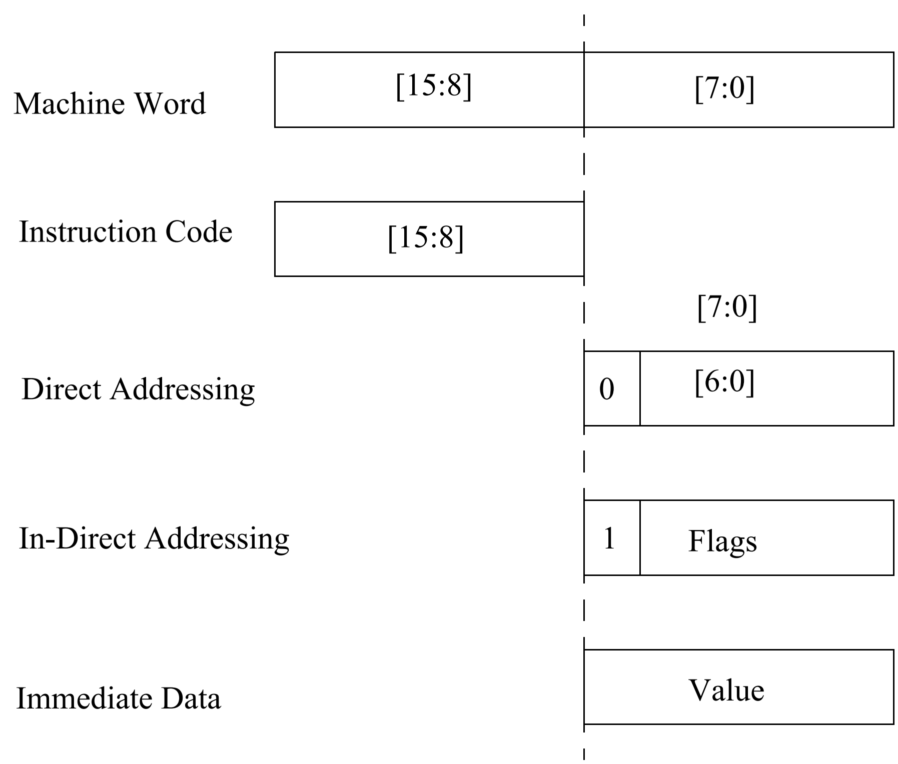
For all instructions, except where noted, (PC) +1 -> PC.

Symbols:

ACC	Accumulator
AR	auxiliary register 0 or 1
ARP	auxiliary register pointer
dma	data memory address
DP	data page pointer
P	multiply product register

PA	port address
PC	program counter
pma	program memory address
T	multiply Temporary register
->	assigned to
||	absolute value
()	contents of

Machine words are built as follows:



ABS- Absolute value of accumulator

Direct Addressing:	ABS
Indirect Addressing:	N/A
Operands:	N/A
Operation:	|ACC|

ADD- Add to low accumulator

Direct Addressing:	ADD dma, shift
Indirect Addressing:	ADD {*|*+|*-}, shift, next ARP
Operands:	0 <= shift < = 15, 0 <= dma <= 127, ARP = 0, 1
Operation:	(ACC) + (dma)*2^{shift} -> ACC
	Modify AR(ARP), and ARP as specified

ADDH- Add to high accumulator

Direct Addressing:	ADDH dma
Indirect Addressing:	ADDH {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(ACC) + (dma)*2^{16} -> ACC
	Modify AR(ARP), and ARP as specified

ADDS- Add to low accumulator with sign-extension suppressed

Direct Addressing:	ADDS dma
Indirect Addressing:	ADDS {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(ACC) + (dma) -> ACC
	Modify AR(ARP), and ARP as specified

AND- And with low accumulator

Direct Addressing:	AND dma
Indirect Addressing:	AND {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	((ACC) & (dma)) & 0x0000ffff -> ACC
	Modify AR(ARP), and ARP as specified

APAC- Add Product to accumulator

Direct Addressing:	APAC
Indirect Addressing:	N/A
Operands:	N/A
Operation:	(ACC) + (P) -> ACC

B- Branch unconditionally

Direct Addressing:	B pma
Indirect Addressing:	N/A
Operands:	0 <= pma <= 0x1ff
Operation:	pma -> PC

BANZ- Branch if auxiliary register != 0

Direct Addressing:	BANZ pma
Indirect Addressing:	BANZ pma, {*|*+|*-}, next ARP
Operands:	0 <= pma <= 0x1ff, ARP = 0, 1
Operation:	IF AR(ARP) != 0, THEN pma -> PC ELSE (PC) + 2 -> PC Modify AR(ARP), and ARP as specified

BGEZ- Branch if accumulator >= 0

Direct Addressing:	BGEZ pma
Indirect Addressing:	N/A
Operands:	0 <= pma <= 0x1ff
Operation:	IF (ACC) >= 0, THEN pma -> PC ELSE (PC) + 2 -> PC

BGZ- Branch if accumulator > 0

Direct Addressing:	BGZ pma
Indirect Addressing:	N/A
Operands:	0 <= pma <= 0x1ff
Operation:	IF (ACC) > 0, THEN pma -> PC ELSE (PC) + 2 -> PC

BIOZ- Branch if bio == 0

Direct Addressing:	BIOZ pma
Indirect Addressing:	N/A
Operands:	0 <= pma <= 0x1ff

Operation:	IF (BIO) == 0,
	THEN pma -> PC
	ELSE (PC) + 2 -> PC

BLEZ- Branch if accumulator <= 0

Direct Addressing:	BLEZ pma
Indirect Addressing:	N/A
Operands:	0 <= pma <= 0x1ff
Operation:	IF (ACC) <= 0,
	THEN pma -> PC
	ELSE (PC) + 2 -> PC

BLZ- Branch if accumulator < 0

Direct Addressing:	BLZ pma
Indirect Addressing:	N/A
Operands:	0 <= pma <= 0x1ff
Operation:	IF (ACC) < 0,
	THEN pma -> PC
	ELSE (PC) + 2 -> PC

BNZ- Branch if accumulator != 0

Direct Addressing:	BNZ pma
Indirect Addressing:	N/A
Operands:	0 <= pma <= 0x1ff
Operation:	IF (ACC) != 0,
	THEN pma -> PC
	ELSE (PC) + 2 -> PC

BV- Branch on overflow

Direct Addressing:	BV pma
Indirect Addressing:	N/A
Operands:	0 <= pma <= 0x1ff
Operation:	IF overflow flag == 1,
	THEN pma -> PC && overflow flag -> 0

ELSE (PC) + 2 -> PC

BZ- Branch if accumulator == 0

Direct Addressing:	BZ pma
Indirect Addressing:	N/A
Operands:	0 <= pma <= 0x1ff
Operation:	IF (ACC) == 0,
	THEN pma -> PC
	ELSE (PC) + 2 -> PC

CALA- Call subroutine indirect *(Not implemented)*

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

CALL- Call subroutine direct *(Not implemented)*

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

DINT- Disable interrupts *(Not implemented)*

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

DMOV- Data move in memory

Direct Addressing:	DMOV dma
Indirect Addressing:	DMOV {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(dma) -> dma + 1
	Modify AR(ARP), and ARP as specified

EINT- Enable interrupts *(Not implemented)*

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

IN- Input data from port

Direct Addressing:	IN dma, port address
Indirect Addressing:	IN {*|*+|*-}, port address, next ARP
Operands:	0 <= dma <= 127, 0 <= port address <= 7, ARP = 0, 1
Operation:	(port address) -> (dma) Modify AR(ARP), and ARP as specified

LAC- Load accumulator

Direct Addressing:	LAC dma, shift
Indirect Addressing:	LAC {*|*+|*-}, shift, next ARP
Operands:	0 <= shift <= 15, 0 <= dma <= 127, ARP = 0, 1
Operation:	$(dma)*2^{shift}$ -> ACC Modify AR(ARP), and ARP as specified

LACK- Load accumulator with immediate constant

Direct Addressing:	LACK eight-bit positive constant
Indirect Addressing:	N/A
Operands:	0 <= constant <= 255
Operation:	(eight-bit positive constant) -> (ACC)

LAR- Load Auxiliary register

Direct Addressing:	LAR AR, dma
Indirect Addressing:	LAR AR, {*|*+|*-}, shift, next ARP
Operands:	AR = 0, 1, 0 <= dma <= 127, ARP = 0, 1
Operation:	(dma) -> auxiliary register Modify AR(ARP), and ARP as specified

LARK- Load Auxiliary register with immediate constant

Direct Addressing:	LARK AR, eight-bit positive constant
Indirect Addressing:	N/A
Operands:	AR = 0, 1, 0 <= constant <= 255
Operation:	(eight-bit positive constant) -> (auxiliary register)

LARP- Load Auxiliary register pointer

Direct Addressing:	LARP one-bit constant
Indirect Addressing:	N/A
Operands:	0, 1
Operation:	(constant) -> (ARP)

LDP- Load data page pointer

Direct Addressing:	LDP dma
Indirect Addressing:	LDP {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(dma) & 0x01 -> data page pointer
	Modify AR(ARP), and ARP as specified

LDPK- Load data page pointer with immediate constant

Direct Addressing:	LDPK one-bit constant
Indirect Addressing:	N/A
Operands:	0 <= constant <= 1
Operation:	constant -> data page pointer

LST - Load status from data memory *(Not implemented)*

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

LT- Load multiply temporary operand

Direct Addressing:	LT dma
Indirect Addressing:	LT {*|*+|*-}, next ARP

Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(dma) -> T register
	Modify AR(ARP), and ARP as specified

LTA - Load multiply temporary operand and accumulate previous result

Direct Addressing:	LTA dma
Indirect Addressing:	LTA {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(dma) -> T register,
	(ACC) + (P register) -> ACC
	Modify AR(ARP), and ARP as specified

LTD- Load multiply temporary operand, accumulate previous result, shift data memory

Direct Addressing:	LTD dma
Indirect Addressing:	LTD {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(dma) -> T register,
	(ACC) + (P register) -> ACC,
	(dma) -> dma + 1
	Modify AR(ARP), and ARP as specified

LTP - Load multiply temporary operand, move product to accumulator

Direct Addressing:	LTP dma
Indirect Addressing:	LTP {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(dma) -> T register, (P register) -> ACC
	Modify AR(ARP), and ARP as specified

LTS - Load multiply temporary operand and subtract previous result

Direct Addressing:	LTS dma
Indirect Addressing:	LTS {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(dma) -> T register,

	(ACC) - (P register) -> ACC
	Modify AR(ARP), and ARP as specified

MAR - Modify auxiliary register

Direct Addressing:	MAR dma
Indirect Addressing:	MAR {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	Modifies AR(ARP), and ARP as specified

MPY - Multiply

Direct Addressing:	MPY dma
Indirect Addressing:	MPY {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(T register) * (dma) -> P register
	Modify AR(ARP), and ARP as specified

MPYK- Multiply with immediate constant

Direct Addressing:	MPYK constant
Indirect Addressing:	N/A
Operands:	-2^{12} <= constant <= 2^{12}
Operation:	(T register) * constant -> P register

MAC- Multiply and accumulate

Direct Addressing:	MAC dma
Indirect Addressing:	MAC {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(T register) * (dma) -> P register then
	(ACC) + (P register) -> ACC
	Modify AR(ARP), and ARP as specified

NOP- No operation

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

OR- Or with low accumulator

Direct Addressing:	OR dma
Indirect Addressing:	OR {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	((ACC) | (dma)) & 0x0000ffff -> ACC Modify AR(ARP), and ARP as specified

OUT - Output data from port

Direct Addressing:	OUT dma, port address
Indirect Addressing:	OUT {*|*+|*-}, port address, next ARP
Operands:	0 <= dma <= 127, 0 <= port address <= 7, ARP = 0, 1
Operation:	(dma) -> (port address) Modify AR(ARP), and ARP as specified

PAC- Move Product to accumulator

Direct Addressing:	PAC
Indirect Addressing:	N/A
Operands:	N/A
Operation:	(P register) -> ACC

POP- Pop top of stack to accumulator *(Not implemented)*

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

PUSH- Push accumulator onto stack *(Not implemented)*

Direct Addressing:	N/A
Indirect Addressing:	N/A
Operands:	N/A
Operation:	N/A

RET - Return from subroutine *(Not implemented)*

Direct Addressing:	N/A

Indirect Addressing: N/A

Operands: N/A

Operation: N/A

ROVM- Reset overflow mode register

Direct Addressing: ROVM

Indirect Addressing: N/A

Operands: N/A

Operation: 0 -> OVM status bit

SACH- Store high accumulator

Direct Addressing: SACH dma, shift

Indirect Addressing: SACH {*|*+|*-}, shift, next ARP

Operands: 0 <= shift <= 7, 0 <= dma <= 127,
ARP = 0, 1

Operation: (ACC[31:16])*2^{shift} -> dma
Modify AR(ARP), and ARP as specified

SACL- Store low accumulator

Direct Addressing: SACL dma

Indirect Addressing: SACL {*|*+|*-}, next ARP

Operands: 0 <= dma <= 127, ARP = 0, 1

Operation: (ACC[15:0]) -> dma
Modify AR(ARP), and ARP as specified

SAR- Store auxiliary register

Direct Addressing: SAR AR, dma

Indirect Addressing: SAR AR, {*|*+|*-}, next ARP

Operands: AR = 0, 1, 0 <= dma <= 127, ARP = 0, 1

Operation: (auxiliary register AR) -> dma

SOVM- Set overflow mode register

Direct Addressing: SOVM

Indirect Addressing: N/A

Operands: N/A

Operation: 1 -> overflow mode (OVM status bit)

SPAC- Subtract P register from accumulator

Direct Addressing: SPAC

Indirect Addressing: N/A

Operands: N/A

Operation: (ACC) - (P register) -> ACC

SST- Store status *(Not implemented)*

Direct Addressing: N/A

Indirect Addressing: N/A

Operands: N/A

Operation: N/A

SUB- Subtract from accumulator with shift

Direct Addressing: SUB dma, shift

Indirect Addressing: SUB {*|*+|*-}, shift, next ARP

Operands: 0 < shift < 15, 0 <= dma <= 127, ARP = 0, 1

Operation: (ACC) - (dma)*2^{shift} -> ACC

Modify AR(ARP), and ARP as specified

SUBC- Conditional subtract *(Not implemented)*

Direct Addressing: N/A

Indirect Addressing: N/A

Operands: N/A

Operation: N/A

SUBH- Subtract from high accumulator

Direct Addressing: SUBH dma

Indirect Addressing: SUB {*|*+|*-}, next ARP

Operands: 0 <= dma <= 127, ARP = 0, 1

Operation: (ACC) -(dma)*2^{16} -> ACC

Modify AR(ARP), and ARP as specified

SUBS- Subtract from accumulator with sign-extension suppressed

Direct Addressing:	SUBS dma
Indirect Addressing:	SUBS {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(ACC) -(dma) -> ACC
	Modify AR(ARP), and ARP as specified

TBLR- Table Read

Direct Addressing:	TBLR dma
Indirect Addressing:	TBLR {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(ACC[8:0]) -> pma
	(pma) -> dma
	Modify AR(ARP), and ARP as specified

TBLW- Table Write

Direct Addressing:	TBLW dma
Indirect Addressing:	TBLW {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	(ACC[8:0]) -> pma
	(dma) -> pma
	Modify AR(ARP), and ARP as specified

XOR- Xor with low accumulator

Direct Addressing:	XOR dma
Indirect Addressing:	XOR {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	((ACC) ^ (dma)) & 0x0000ffff -> ACC
	Modify AR(ARP), and ARP as specified

ZAC- Zero accumulator

Direct Addressing:	ZAC
Indirect Addressing:	N/A
Operands:	N/A
Operation:	0 -> ACC

ZALH- Zero accumulator and load high

Direct Addressing:	ZALH dma
Indirect Addressing:	ZALH {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	0 -> ACC[15:0]
	(dma) -> ACC[31:16]
	Modify AR(ARP), and ARP as specified

ZALS- Zero accumulator and load low with sign-extension suppressed

Direct Addressing:	ZALS dma
Indirect Addressing:	ZALS {*|*+|*-}, next ARP
Operands:	0 <= dma <= 127, ARP = 0, 1
Operation:	0 -> ACC[31:16]
	(dma) -> ACC[15:0]
	Modify AR(ARP), and ARP as specified

TDSP Assembler

The TDSP assembler, *tdspasm*, supports compilation of source files formatted using the following conventions. The assembler is case insensitive.

File names for assembly must with a ".asm" suffix. The assembly process will produce three (3) separate output files:

<file_name>.lst - composite machine, opcode listing

<file_name>.sym - cross reference table for symbols and their values

<file_name>.obj - machine object readable by your digital simulator