Y8000 CPU

Technical Manual



Every effort has been made to ensure the accuracy of the information contain herein. If you find errors or inconsistencies please bring them to our attention. In all cases, however, the Verilog HDL source code for the Y8000 design defines “proper operation”.

Copyright © 2024, 2026, Systemyde International Corporation. All rights reserved.

Notice:

“Z8000”, “Z8001”, “Z8002” and “Zilog” are registered trademarks of Zilog, Inc. All uses of these terms in this document are to be construed as adjectives, whether or not the noun “microprocessor”, “CPU” or “device” are actually present.

Index

Introduction 5

Programming Model 9

Addressing/Address Modes 15

Instruction Format 19

Instruction Set 21

External Interface/Timing 179

Interrupts and Traps 207

Refresh 213

Reset 215

Enhancements 217

Verilog HDL Source 219

Test Bench 223

This page is intentionally left blank

Introduction

This manual documents the operation of the Y8000 CPU. The Y8000 design is supplied in Verilog HDL format and can be implemented in any technology supported by a logic synthesis tool that accepts Verilog HDL. The design requires roughly 20K logic gate equivalents. Included in the design package is a test bench that exercises all implemented instructions, flag settings, and representative data patterns. The test patterns should achieve at least 95% fault coverage.

There were a number of variants of the 16-bit Z8000 architecture supplied by Zilog. The Z8002 and Z8004 support a 16-bit linear address space and are identical except that the Z8004 added support for virtual memory. The Z8001 and Z8003 support a 23-bit segmented address (7-bit segment and 16-bit offset) and are identical except that the Z8003 added support for virtual memory. All of these devices were implemented in NMOS technology and the Z8001/2 were available for -55C to +125C operation. Manufacturing of these devices ceased around 1990.

Zilog also supplied the Z16C01 and Z16C02, which were CMOS versions of the Z8001 and Z8002. Manufacturing of these devices ceased in 2012.

A number of mainstream semiconductor suppliers, as well as several captive aerospace semiconductor manufacturers, were licensed second-sources for Z8000 devices. By 2024 all of these suppliers had ceased production of Z8000 devices.

The Y8000 CPU was designed in a clean-room environment and is a clone of the Zilog Z8001 and Z8002 microprocessors, but also contains several backwards-compatible enhancements. The Y8000 design includes a mode input, that is assumed to be hardwired for any given implementation, which selects between Z8001 operation (`z8002 = 0`) and Z8002 (`z8002 = 1`) operation. This is identical to how Zilog differentiated the Z8001/2, the Z8003/4 and the Z16C01/2.

Only publicly available documentation has been used to create this design so there may be minor differences where the public documentation is misleading or lacking. With only a few exceptions, the instruction execution times are identical between the Y8000 and the Z8001/2. All known differences for individual instructions are listed in the instruction description chapter as notes.

This document should always be used as the final word on the operation of the Y8000 CPU, but it is useful to refer to the Zilog documentation if the description given here is too cryptic. The Z8000 architecture is fifty years old, so it is assumed that it is already at least somewhat familiar to the reader, but an overview is presented here.

The Z8000 architecture includes sixteen 16-bit general-purpose registers, and uses eight distinct data types ranging from single bits to 64-bit quadruple words and arbitrary-length byte strings. The architecture supports eight different types of addressing modes. A Z8000 architecture CPU (like the Y8000 CPU) has over 111 instruction types in its instruction set. The multiple addressing modes and data types, when coupled with the instruction types, produces more than 447 different instructions for the Y8000 CPU instruction set.

The architecture includes status signals that can be used to determine the nature of each bus transaction. The status signals can be decoded and used to implement systems having multiple memory address spaces, with each space being dedicated to a specific purpose.

The Z8000 architecture includes two operating modes: System mode, and Normal mode. This feature allows the operating system functions to be easily separated from application program functions to enhance operating system and application data security.

The architecture provides dedicated instructions for input/output operations, allowing the separation I/O operations from memory-related operations (the I/O space is separate from the memory space). The dedicated I/O instructions can only be accessed while operating in System mode. Applications running in Normal mode cannot directly affect the operation of the I/O ports. This architecture does not prevent the user from designing a system with memory-mapped I/O.

The Z8000 architecture includes an interrupt mechanism that processes interrupts and exceptions from various sources. These sources include a single non-maskable interrupt, a single non-vectored interrupt, up to 256 vectored interrupt sources, and traps caused by the following events: execution of a privileged instruction while in Normal mode, execution of an Extended instruction, execution of the System Call instruction, and a Segment trap. The interrupt mechanism stores the program status, transfers program control to an interrupt service routine, and restores program status at the end of the interrupt process. The interrupt mechanism of the Z8000 architecture includes a means of assigning priorities to the interrupts, but this is external to the CPU itself.

The full Z8000 architecture implements a segmented memory map that allows the processor to directly address six memory spaces of up to 8 Megabytes (8,388,608 bytes) each. Each memory segment consists of a 64K (65,536) byte block. The sixteen address lines present on the Z8000 processors are used to access the individual locations within the memory segment while a seven-bit segment register, and segment number outputs, are used to provide access to each of the segments. The Y8000 contains an extra segment address bit that doubles the number of memory segments available, but because of the way segmented addresses are encoded in the instruction set there are some restrictions on accessing these additional memory segments.

The six Z8000 memory spaces are not restricted to the use envisioned by the architecture. Simple external logic allows for the creation of Global/Foreground/Background memory spaces or other custom memory organizations.

The architecture includes a refresh-control block that is designed to generate memory refresh cycles for dynamic random access memory (DRAM).

The Z8000 architecture includes provisions to allow the processor to yield control of the system address/data, control, and status signals to another processor in the system in response to a bus request signal. When a bus request is granted, the processor enters the bus-disconnect state. Program execution is suspended and the CPU disconnects itself from the bus (signals are placed into a high-impedance state). When the bus request is removed from the processor, the CPU regains control of the bus, and continues with program execution.

The Z8000 architecture also includes a feature known as the Extended Processing Architecture (EPA). This feature allows the instruction set of the processor to be augmented by external devices (Extended Processing Units, or EPUs) on the bus. The Extended instructions are used to exploit this feature. If an EPU is available, the processor will handle only the data transfer portion of the instruction, and leave execution of the instruction to the EPU. If no EPU is present, the processor can handle the instruction itself using an Extended Instruction trap handling software routine. The EPA bit located in the Flag and Control Word (FCW) determines how the instruction will be handled, and the state of this bit is available as an external signal from the Y8000.

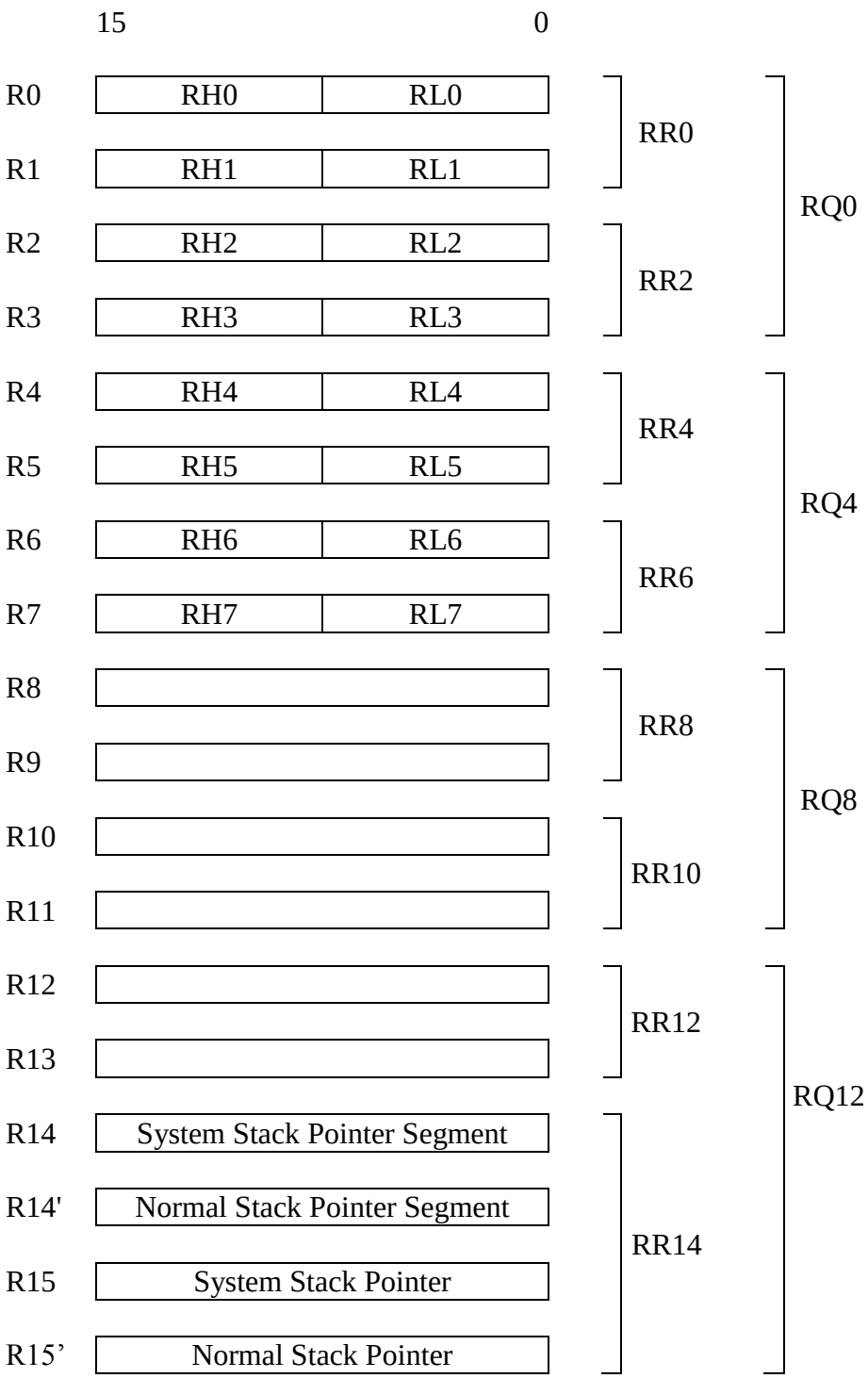
This page is intentionally left blank

Programming Model

The 16-bit Z8000 architecture contains sixteen 16-bit general-purpose registers. With only a few exceptions, all general-purpose registers can be used for any instruction operand. These registers allow data ranging from bytes to quadruple words. None of the general purpose registers are affected by reset.

Word registers are specified in assembly language programs as R0 through R15. Sixteen byte registers (RH0-RH7 and RL0-RL7) can be used as accumulators, and overlap the first eight word registers. Registers may be paired into eight double word registers RR0 through RR14 for 32-bit operands. Registers may also be grouped in groups of four quadruple word registers RQ0 through RQ12 for 64-bit operands. The double word and quadruple word registers are used by operations such as Multiply, Divide, and Extend Sign.

There are two hardware stack pointers, one for each operating mode (Normal and System). The System Stack Pointer is used in System mode, during interrupt or trap handling, and for System Calls. The Normal Stack Pointer is used in Normal mode and in this mode only the Normal Stack Pointer is accessible. When operating in System mode the System Stack Pointer is accessed as a general-purpose register and the Normal Stack Pointer is accessed as a special control register. Register R15 is the Stack Pointer in Non-segmented mode. Register pair RR14 is the stack pointer in Segmented mode.



General Purpose Registers

The Program Status reflects the current operating state of the processor. Included in the Program Status is the Flag and Control Word (FCW) and Program Counter (PC). The Program Status is automatically pushed onto the System stack in response to an interrupt or trap.

After reset, the Y8000 CPU fetches the Program Status from dedicated locations in the Program address space. The actual memory addresses used depend on the state of the `z8002` input to the design.

If the `z8002` input is tied Low the Y8000 CPU operates as an enhanced clone of the Z8001 CPU. In this case after reset the PC Segment number is first set to 0x00, the FCW is fetched from memory address 0x0002, the starting PC Segment is fetched from memory address 0x0004 and the starting PC offset from memory address 0x0006.

If the `z8002` input is tied High the Y8000 CPU operates as an enhanced clone of the Z8002 CPU. In this case after reset the FCW is fetched from memory address 0x0002 and the starting PC is fetched from memory address 0x0004.

The Flag and Control Word register contains both processor status bits and processor control bits. The low-order byte contains system status flags that are used by instructions to control program branching and looping. The high order byte contains control bits that are used to enable and disable the interrupt system and control certain operating modes.

The six processor status bits are:

- Carry (C) -- This bit indicates that a carry out of the high order bit position of a register being used as an accumulator has occurred.

- Zero (Z) -- This bit indicates that the result of a processor operation is zero.

- Sign (S) -- This bit indicates that the result of an processor operation has produced a negative number.

- Overflow (V)-- This bit indicates that an overflow has occurred (on processor arithmetic operations) or even parity (after processor logical operations).

- Decimal-Adjust (D) -- This bit is used in BCD arithmetic to indicate the type of instruction that was executed (addition or subtraction).

- Half-Carry (H) --This bit is used to convert the binary result of a previous addition or subtraction of BCD numbers into the correct decimal result.

The control bits in the high order byte of the FCW are:

- Non-Vectored Interrupt Enable (NVI) -- This bit is used to enable or disable the processor's response to interrupts on its non-vectored interrupt input.

- Vectored Interrupt Enable (VI) -- This bit is used to enable or disable the processor's response to interrupts on its vectored interrupt input.

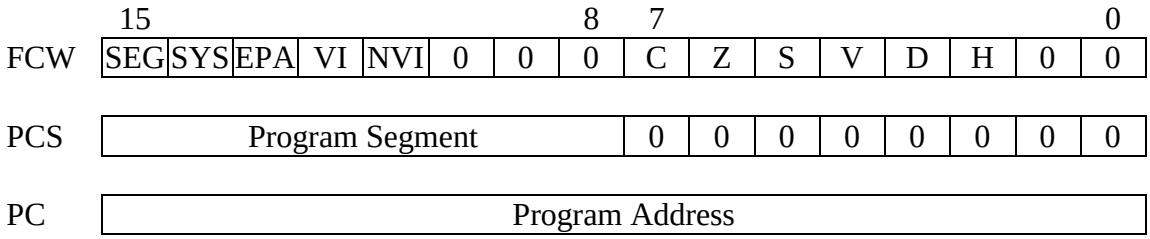
- System Mode (SYS) -- This bit determines if the processor is to operate in the System mode (High) or Normal mode (Low). The Normal/System hardware output signal of is normally the complement of this bit.

- Extended Processor Architecture (EPA) -- This bit indicates the presence or absence of Extended Processing Units (EPU) in the system. If EPUs are present, this bit should be set to one, and the processor will execute Extended instructions as they are encountered. If EPUs are not present in the system and an Extended instruction is fetched for execution, the processor will generate an Extended Instruction trap. The state of this bit is available as an output from the Y8000.

- Segmentation Mode (SEG) -- This bit is present only in a segmented Z8000 CPU and is always zero in a non-segmented Z8000 CPU. When set to one, the processor is executing in Segmented mode. When set to zero, the processor is executing in Non-segmented mode. This bit if forced to zero when the `z8002` mode input is High.

The Program Counter (PC) is a sixteen bit register. All instruction fetches are 16 bits wide, so the least significant bit of the PC should always be zero. However, the hardware neither forces nor checks that this is true.

When operating in Z8001 mode the PC also contains a 8-bit Program Counter Segment (PCS) register that selects the memory segment where executing code resides. Bit 7 of the PCS is not used by the Z8001, but is a backwards-compatible Y8000 upgrade. In Z8002 mode the PCS will never be written and will always contain 0x00. The lower byte of the PCS does not exist in the Y8000 and will always be read as 0x00.



Program Status Registers

The Program Status Area Pointer (PSAP) contains the address of the Program Status Area, (PSA) which is a table that contains FCW and PC values used by the interrupt and exception handling hardware of the processor. When an interrupt or trap occurs in the processor execution cycle, the Program Status Area is where the processor obtains new values for the FCW and PC in order to process the exception. The PSAP points to an area in the Program memory address space and is cleared to all zeros by reset. The lower byte of the PSAP is always zero, which means that the Program Status Area is aligned on a 256-byte boundary.

When operating in Z8001 mode the PSAP also contains a 8-bit Program Status Area Segment (PSAS) register that selects the memory segment where the PSA resides. Bit 7 of the PSAS is not used by the Z8001, but is a backwards-compatible Y8000 upgrade. In Z8002 mode the PSAPS can still be written using a LDCTL instruction. The lower byte of the PSAS does not exist in the Y8000 and will always be read as 0x00.

	15 … 8	7							0
PSAS	PSAP Segment	0	0	0	0	0	0	0	0
PSAP	PSAP Upper Pointer	0	0	0	0	0	0	0	0

Program Status Area Pointer

The Z8000 architecture contains hardware that can be used to automatically refresh dynamic memory in the system. The Refresh Control register (RFSH) contains an 8-bit row counter, and a six-bit rate counter, as well as an enable/disable control bit. Bit 15 of this register (the Enable bit) is cleared by reset. The Y8000 adds an Extended Refresh register (RFSX) that adds nine bits to the row counter and six bits to the rate counter, along with a separate enable bit for the expanded features. The entire RFSX register is cleared by reset and can only be accessed by a Y8000-specific LDCTLL instruction.

	15	14 … 9	8 … 0
RFSX	EN	Rate[11:6]	Row[16:8]

RFSH	EN	Rate[5:0]	Row[7:0]	0

Refresh Control

This page is intentionally left blank

Addressing/Address Modes

The Z8000 architecture supports separate memory and I/O address spaces, and each of these address spaces can be further subdivided if necessary. The exact address space being accessed is encoded on the four status signals that are output buy the processor with each bus transaction. The available address spaces are:

- Program address space, with two separate status encodings to distinguish the first word of an instruction fetch from some other program memory fetch.
- Data address space, used to access data.
- Stack address space, used to access data via the Stack Pointer R15 (RR14 in Segmented mode).
- Standard I/O address space, used for all regular I/O operations.
- Special I/O address space, which is usually reserved for I/O operations between the processor and a Memory Management Unit (MMU) or a Direct Memory Access (DMA) controller.

Each of the memory spaces can be further separated externally according to the System or Normal mode. This provides the ability to design and implement operating systems that protect the System operation and information from being corrupted or accessed by user applications.

In this architecture, each of the memory address spaces can be a maximum of 64K bytes, allowing the Y8000 CPU, operating as a Z8002 CPU, to address 384 KB of memory (without an external MMU or Global bus support logic). When operating as a Z8001 CPU the Y8000 CPU can address 8M bytes of memory in each address space (doubled when the new SN7 is used). Each I/O address space is a maximum of 64K port addresses.

I/O address space is accessible only from the System mode of operation. This prevents user programs from gaining access to System resources directly, giving the System software complete control over peripheral devices.

The Z8000 architecture is big-endian, meaning that the most-significant data element is addressed at the lowest memory address. Bytes transferred to or from odd memory address locations (address bit 0 = 1) are always transmitted on lines AD7-AD0 (data bit 0 on AD0). Bytes transferred to or from even memory address locations (address bit 0 = 0) are always transmitted on lines AD15-AD8 (data bit 0 on AD8).

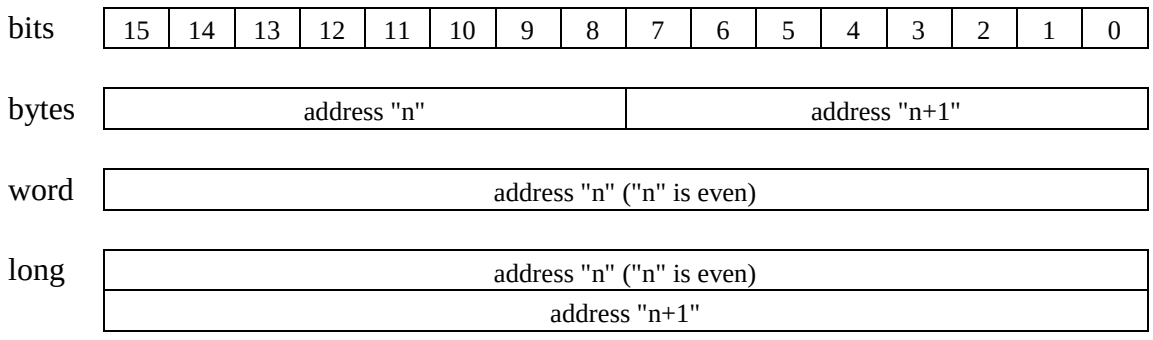
During byte writes, the CPU places the same byte on both halves of the bus. The system hardware must use AD0 to determine which half of the bus contains the actual data to be written. For byte reads, the CPU will read all 16-bits of data on the AD15-AD0 lines and automatically select the proper half of the bus that contains the active data.

I/O devices can use either 8-bit or 16-bit data busses for either I/O address space. The address of a peripheral with a 16-bit wide data bus may be odd or even. Peripherals having 8-bit wide data busses connected on lines AD7-AD0 must either be addressed using odd addresses or ignore the least significant bit of the address and use two I/O addresses per I/O port. Normally special I/O devices connect to the upper half of the bus, and thus use even addresses.

Each memory address space consists of a block of 64KB of memory, with addresses being consecutively numbered in ascending order. The 8-bit byte is the basic addressable element in memory address space. The Z8000 architecture supports three additional addressable data elements:

- Bits, either in bytes or words
- 16-bit words
- 32-bit double-words

The type of data element being accessed depends upon the instruction being executed. The assembler mnemonics allow for addressing bit, byte, word, or double-word data. Not all instructions can access all types of data. Addressable data elements are shown below:



A bit can be addressed by specifying an address (byte or word) and the location of the bit within the byte (7-0) or word (15-0). Bits are numbered right to left, from the least significant bit to the most significant bit.

The address of a data element longer than one byte (word, double-word) is the same as that of the byte with the lowest memory address within the word or double-word. This byte is the leftmost, highest-order (most significant) byte of the word or double-word.

Word and double-word addresses are always even-numbered. Low bytes of words are stored in odd-numbered addresses and high bytes of words are stored in even numbered addresses. Byte data can be stored in odd or even addresses.

Memory locations 0x0000-0x0007 in the Program memory space are reserved for the FCW, PC Segment (operating as a Z8001) and PC that are fetched after a reset. Except for this reserved memory space, there are no restrictions placed on any locations within the processor memory space, although the 256-byte block addressed by the PSAP is used for the Program Status information for interrupts, traps and system calls rather than program information.

The architecture supports eight data types directly, although Extended Processing Units may create and access new data types, such as floating point numbers. Five of the eight data types are fixed length data and the remaining three data types have variable lengths. Each data type is supported by numerous instructions that operate upon it directly. The data types are as follows:

- Bit
- Signed and unsigned byte, word, double word or quadruple word binary data
- Byte or word logical data
- Word or double-word address
- Byte of packed BCD (binary coded decimal) integer
- Dynamic-length string of byte data
- Dynamic-length string of word data
- Dynamic-length stack of word or double-word data

Bit data can be manipulated either in the general-purpose registers or in memory. Binary and BCD integers, and logical values may be manipulated in registers, although operands can be fetched directly from memory. Addresses can only be manipulated in registers, while strings and stack data can only be manipulated in memory.

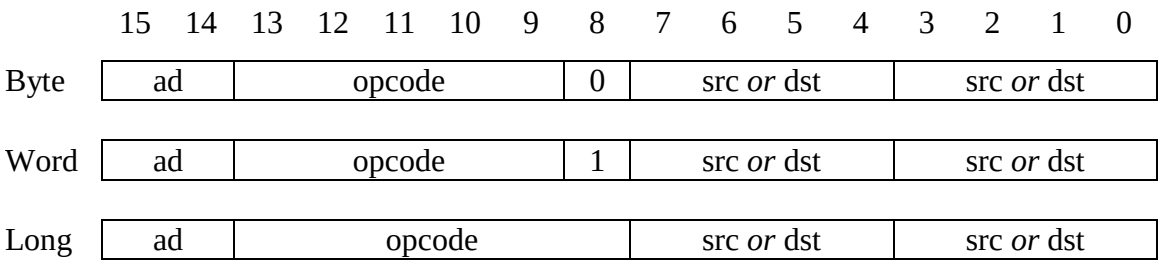
The operands for instructions can be specified using one of eight addressing modes. For some instructions no addressing mode is used, because the operand is implied. The majority of instructions can use any of the five common addressing modes: register, immediate, indirect register, direct address and indexed. A few instructions can use the relative address mode and only load and store instructions can use base address and base index addressing modes.

Because of the way that the addressing modes are encoded into the opcode, register R0 (and register pair RR0) cannot be used in the indirect address, index, base address or base index modes.

- Register: the contents of the register.
- Immediate: in the instruction.
- Indirect Register: the contents of the location whose address is in the register (or register pair in Segmented mode).
- Direct Address: the contents of the location whose address is in the instruction.
- Index: the contents of the location whose address is the sum of the address in the instruction plus the contents of the register.
- Relative Address: the contents of the location whose address is the sum of the contents of the Program Counter plus the displacement in the instruction.
- Base Address: the contents of the location whose address is the sum of the contents of a register (or register pair in Segmented mode) plus the displacement in the instruction.
- Base Index: the contents of the location addressed by the sum of the contents of one register (or register pair in Segmented mode) plus the contents of another register

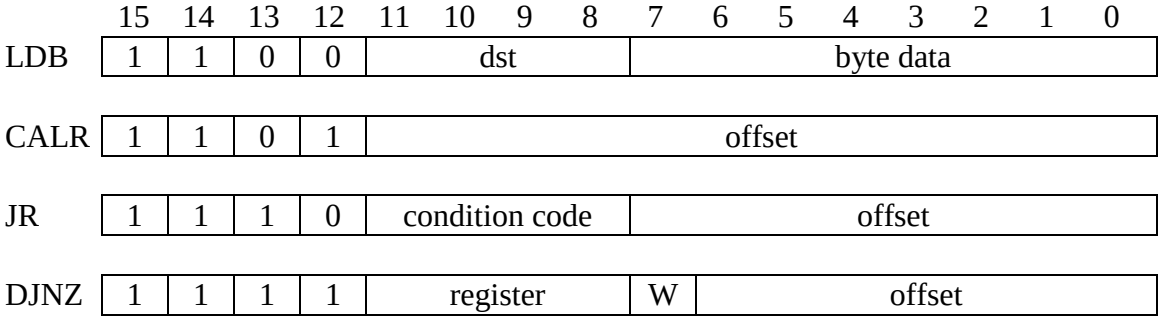
Instruction Formats

The format of instructions in the Z8000 architecture is very regular. The general instruction format uses a two bit field to select the addressing mode (**ad**), a five or six bit field for the operation code (**opcode**), a four bit field to select the source (**src**) operand and a four bit field to select the destination (**dst**) operand.



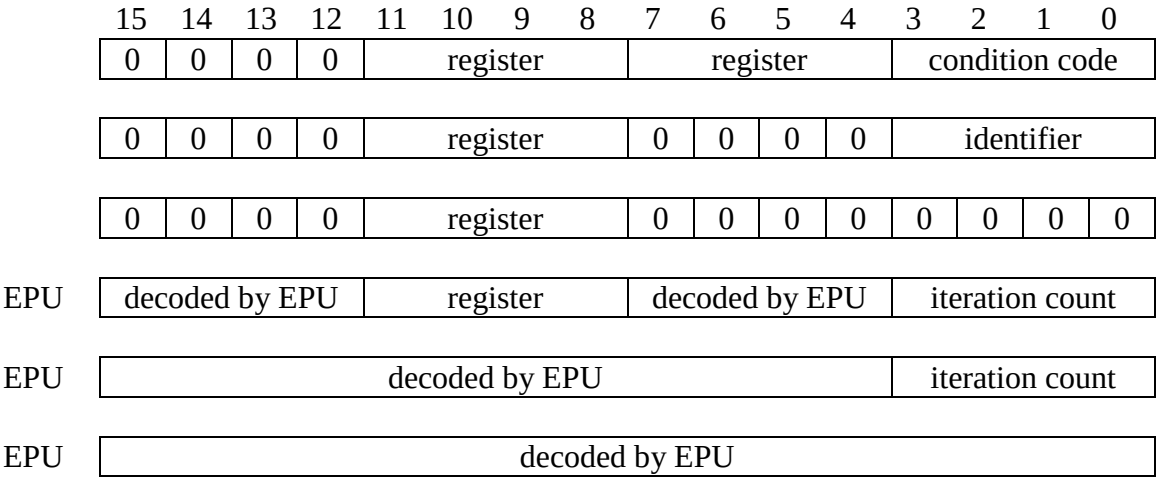
General Instruction Format (first word of instruction)

In the addressing mode field the bit combination 00 usually selects either immediate data or Indirect Register addressing, the bit combination 01 selects either Direct addressing or Index addressing and the bit combination 10 selects Register addressing. The choice between Immediate and Indirect Register or between Direct and Index is made using one of the bit combinations in bits 7-4. This is why R0 cannot be used with Indirect Register or Index addressing. The bit combination 11 in the address mode field is used to specify the compact format for the instruction. Four of the most commonly used instructions have their own compact format.



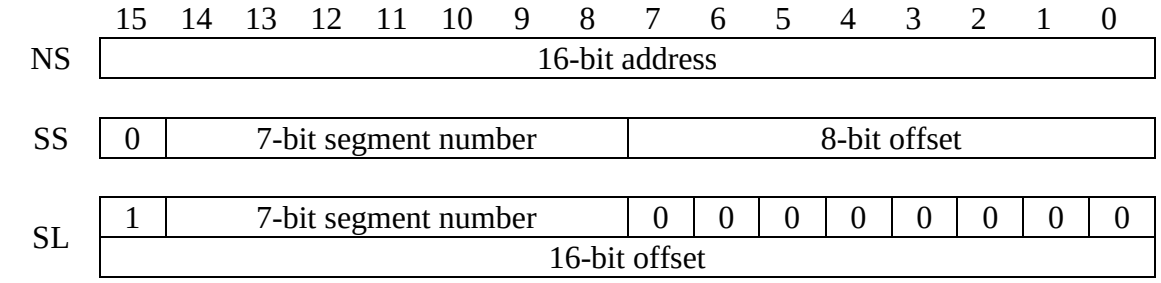
Compact Instruction Formats

Some infrequently used or complex instructions require two words to encode all of the information. All EPU instructions also require at least two words.



General Instruction Format (second word of instruction)

Opcodes that contain an address can include one or two address words, depending on the address type. In Non-segmented mode only one word of address is ever used. In Segmented mode a short address, consisting of a segment number and an offset byte, requires one word. In Segmented mode a long address consists of a first word containing the segment number and a second word containing the 16-bit offset, These three cases are shown below:



Address format (second/third word of instruction)

Using bit 15 to distinguish between Segmented Short and Segmented Long addressing is the reason that the Z8000 architecture only uses 7-bit segment numbers everywhere. But rather than enforce this restriction, the Y8000 allows 8-bit segment numbers for the Program Counter, the Program Status Area Pointer, as well as Indirect Register and Base Index addressing.

In the 32-bit version of the Z8000 architecture the two-word address format has additional options: a 7-bit segment number with a 24-bit offset, a 15-bit segment number with a 16-bit offset, or a 32-bit linear address. None of these options are implemented in the Y8000.

Instruction Set

This chapter presents the assembly language syntax, addressing modes, flag settings, binary encoding, and execution time for the Y8000 instruction set. The entire instruction set is presented in alphabetical order.

The assembly language syntax is identical to that used by the original Zilog assembler. Different assembler programs may or may not use identical syntax. The syntax is presented generically at the beginning of each instruction, with the details presented for each addressing mode later in each entry.

The operation of each instruction is specified in a format similar to Verilog HDL for minimum ambiguity, but no descriptive text or examples are included.

The effect of the instruction on each flag is listed, with a brief description. Normally the flags are updated by the main operation of the instruction, but for some complex instructions different flags may be affected by different parts of the instruction. This is specified in the description. Where the same flag is reused during a complex instruction a note is included at the end of the instruction description with the details.

Fields in the instruction are listed using shortcuts for common fields. These shortcuts should be self-explanatory in most cases, but will be detailed here for completeness.

The most common fields in the instruction specify a CPU register, as follows:

`Rbdd`	Byte register used as a destination operand.
`Rbss`	Byte register used as a source operand.
`Rddd`	Word register used as a destination operand.
`Rdnz`	Word register used in addressing the destination.
`Rrrr`	Word register (usually specifies a counter).
`Rsss`	Word register used as a source operand.
`Rsnz`	Word register used in addressing the source.
`R1nz`	Word register used in addressing the first source.
`R2nz`	Word register used in addressing the second source.
`Rxxx`	Word register used as an index.
`RQdd`	Register quad used as a destination operand.
`RRdd`	Register pair used as a destination operand.
`RRss`	Register pair used as a source operand.

The register shortcuts shown in the Assembly Syntax are for the non-segmented case. For registers used for Indirect Address, Base Address or Base Index Address addressing modes

in Segmented mode a register pair must be specified. The Zilog assembler will enforce this selection.

Registers are encoded according to the following table. Note that in the case of `Rdnz`, `Rsnz`, `R1nz` and `R2nz` the "0000" case is illegal and is usually used to select a different addressing mode. Illegal cases for `RRdd`, `RRss` and `RQdd` should never be used, because they have meaning in the 32-bit version of the Z8000 architecture.

Instructions will still execute with an illegal register encoding but the results may be scrambled in the registers because of the way the register addresses are treated internally in the design.

Encoding in opcode	`Rbss`, `Rbdd`	`Rddd`, `Rsss`, `Rrrr`, `Rxxx`	`Rsnz`, `Rdnz`, `R1nz`, `R2nz`	`RRdd`, `RRss`	`RQdd`
0000	RH0	R0	illegal	RR0	RQ0
0001	RH1	R1	R1	illegal	illegal
0010	RH2	R2	R2	RR2	illegal
0011	RH3	R3	R3	illegal	illegal
0100	RH4	R4	R4	RR4	RQ4
0101	RH5	R5	R5	illegal	illegal
0110	RH6	R6	R6	RR6	illegal
0111	RH7	R7	R7	illegal	illegal
1000	RL0	R8	R8	RR8	RQ8
1001	RL1	R9	R9	illegal	illegal
1010	RL2	R10	R10	RR10	illegal
1011	RL3	R11	R11	illegal	illegal
1100	RL4	R12	R12	RR12	RQ12
1101	RL5	R13	R13	illegal	illegal
1110	RL6	R14	R14	RR14	illegal
1111	RL7	R15	R15	illegal	illegal

Immediate data is encoded in the instruction in a number of different ways, depending on the instruction. The assembly language mnemonics will always use just "`#data`" or "`#n`", independent of the actual width or encoding in the instruction. The following shortcuts are employed:

Shortcut	Meaning
`b`	Positive (or zero) twos-complement number used for shift left count.
`-b`	Negative twos-complement number used for shift right count
`bbb`	Three bit unsigned value (range 0x0 - 0x7, corresponding to 0 - 7).
`bbbb`	Four bit unsigned value (range 0x0 - 0xF, corresponding to 0 - 15).
`#data`	Four, eight or sixteen bit immediate data.
`#data (high)`	Most significant word of thirty-two bit immediate data.
`#data (low)`	Least significant word of thirty-two bit immediate data.
`dddd`	Four bit unsigned value (range 0x0 - 0xF, corresponding to 0 - 15).
`ddd_dddd`	Seven bit unsigned value (range 0x00 - 0x7F, corresponding to 0 - 127).
`dddd_dddd`	Eight bit signed value (range 0x00 - 0xFF, or -127 to +128)
`dddd_dddd_dddd`	Twelve bit signed value (range 0x000 - 0xFFF, or -2048 to +2047).
`nnnn`	Four bit unsigned value (range 0x0 - 0xF, corresponding to 1 - 16).
`ssssrccc`	Eight bit System Call identifier (range 0x00 - 0xFF)

The "cccc" field encodes one of sixteen possible flags combinations to be tested as part of the instruction, as shown in the table below. Some encodings have more than one possible assembly language mnemonic, and the "always true" case is implied when no other case is specified.

cccc	Encoding in opcode	Flag combination	Meaning
F	0000	any	Always False
LT	0001	(S ^ V) = 1	Less Than
LE	0010	(Z || (S ^ V)) = 1	Less Than or Equal
ULE	0011	(C || Z) = 1	Unsigned Less Than or Equal
OV PE	0100	V = 1	Overflow Parity Even
MI	0101	S = 1	Minus
Z EQ	0110	Z = 1	Zero Equal
C ULT	0111	C = 1	Carry Unsigned Less Than
	1000	any	Always True
GE	1001	(S ^ V) = 0	Greater Than or Equal
GT	1010	(Z || (S ^ V)) = 0	Greater Than
UGT	1011	(!C && !Z) = 1	Unsigned Greater Than
NOV PO	1100	V = 0	No Overflow Parity Odd
PL	1101	S = 0	Plus
NZ NE	1110	Z = 0	Not Zero Not Equal
NC UGE	1111	C = 0	No Carry Unsigned Greater Than or Equal

The remaining shortcuts should be self-explanatory. The shortcut "CZSV" is a four bit field where each bit corresponds to the flag of the same name. The shortcut "VN" is a two bit field with each bit corresponding to one of the interrupt enable bits of the same name in the FCW. The "W" bit selects word (1) or byte (0) operation. The "S" bit selects Special I/O (1) or Normal I/O (0) operation.

The execution times are listed here only as a number of clock cycles. These numbers assume no extra wait states and no interrupts during execution of an iterative instruction, but do include any automatic wait states.

If a single clock number is given, it applies independent of address type or operating mode. If two numbers, separated by a "/" are given, the first number is for Non-segmented mode and the second number is for Segmented mode. If three numbers, each separated by "/" are given, the first number applies for Non-segmented mode, the second number applies in Segmented mode with a short address and the third number applies in Segmented mode with a long address.

In general, for Direct Address mode Segmented Short addressing adds one clock cycle to the normal execution time, while Segmented Long adds three clocks to the normal execution time. Indexed Address mode is slightly different, with Segmented Short requiring the same number of clocks as normal execution, and Segmented Long adding three clocks to the normal execution time. Of course there are some exceptions to this rule of thumb.

ADC

Add With Carry

ADC dst, src
ADCB

dst: R
src: R

Operation: dst <= dst + src + C

Flags:

C: Set if arithmetic carry from result MSB; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow; cleared otherwise.
D: Cleared (ADCB);
Unaffected (ADC).
H: Set if arithmetic carry from bit 3; cleared otherwise (ADCB);
Unaffected (ADC).

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	ADC Rd, Rs ADCB Rbd, Rbs	`1011010W_Rsss_Rddd`	5

ADD

Add

ADD dst, src
ADDB
ADDL

dst: R
src: R, IM, IR, DA, X

Operation: dst <= dst + src

Flags:
C: Set if arithmetic carry from MSB; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow; cleared otherwise.
D: Cleared (ADDB);
Unaffected (ADD, ADDL).
H: Set if arithmetic carry from bit 3; cleared otherwise (ADDB);
Unaffected (ADD, ADDL).

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	ADD Rd, Rs ADDB Rbd, Rbs	`1000000W_Rsss_Rddd`	4
	ADDL RRd, RRs	`10010110_RRss_RRdd`	8
IM:	ADD Rd, #data	`00000001_0000_Rddd` `#data`	7
	ADDB Rbd, #data	`00000000_0000_Rbdd` `#data | #data`	7

	ADDL RRd, #data	`00010110_0000_RRdd`
		`#data (high)`
		`#data (low)`
			14

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	ADD Rd, @Rs ADDB Rbd, @Rs	`0000000W_Rsnz_Rddd`	7
	ADDL RRd, @Rs	`00010110_Rsnz_RRdd`	14
DA:	ADD Rd, address ADDB Rbd, address	`0100000W_0000_Rddd` `1 or 2 address words`	9/10/12
	ADDL RRd, address	`01010110_0000_RRdd` `1 or 2 address words`	15/16/18
X:	ADD Rd, addr(Rs) ADDB Rbd, addr(Rs)	`0100000W_Rsnz_Rddd` `1 or 2 address words`	10/10/13
	ADDL RRd, addr(Rs)	`01010110_Rsnz_RRdd` `1 or 2 address words`	16/16/19

AND

Logical AND

AND dst, src
ANDB

dst: R
src: R, IM, IR, DA, X

Operation: dst <= dst & src

Flags:
C: Unaffected.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if result parity is even; cleared otherwise (ANDB); Unaffected (AND).
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	AND Rd, Rs ANDB Rbd, Rbs	`1000011W_Rsss_Rddd`	4
IM:	AND Rd, #data	`00000111_0000_Rddd` `#data`	7
	ANDB Rbd, #data	`00000110_0000_Rbdd` `#data | #data`	7
IR:	AND Rd, @Rs ANDB Rbd, @Rs	`0000011W_Rsnz_Rddd`	7
DA:	AND Rd, address ANDB Rbd, address	`01000111_0000_Rddd` `1 or 2 address words`	9/10/12
X:	AND Rd, addr(Rs)	`0100011W_Rsnz_Rddd` `1 or 2 address words`	10/10/13

BIT

Bit Test Static

BIT dst, src
BITB

dst: R, IR, DA, X
src: IM

Operation: Z <= !dst[src]

Flags:
C: Unaffected.
Z: Set if selected bit is zero; cleared otherwise.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	BIT Rd, #b BITB Rbd, #b	`1010011W_Rddd_bbbb`	4
IR:	BIT @Rd, #b BITB @Rd, #b	`0010011W_Rdnz_bbbb`	8
DA:	BIT address, #b BITB address, #b	`0110011W_0000_bbbb` `1 or 2 address words`	10/11/13
X:	BIT addr(Rd), #b BITB addr(Rd), #b	`0110011W_Rdnz_bbbb` `1 or 2 address words`	11/11/14

Notes:

1. Only bits 2-0 of the opcode are used to select the bit in the case of BITB, and bit 3 of the opcode is ignored.

BIT

Bit Test Dynamic

BIT dst, src
BITB

dst: R
src: R

Operation: Z <= !dst[src]

Flags:
C: Unaffected.
Z: Set if selected bit is zero; cleared otherwise.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	BIT Rd, Rs BITB Rbd, Rs	0010011W_0000_Rsss 0000_Rddd_0000_0000	10

Notes:

1. Zilog Z8000 CPUs restrict the source register to be one of R0 - R7 for BITB. This restriction does not apply to the Y8000 design. Any register may be used as the source.

2. Only bits 3-0 of the source operand are used for the bit select for BIT; only bits 2-0 of the source operand are used for the bit select for BITB.

3. Only bits 11-8 of the second word of the opcode are used. All other bits in the second word of the opcode are ignored.

CALL

Call

CALL dst dst: IR, DA, X

Operation: SP <= SP - 2
@SP <= PC
PC <= dst

Flags: **C:** Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	CALL @Rd	`00011111_Rdnz_0000`	10/15
DA:	CALL address	`01011111_0000_0000` `1 or 2 address words`	12/17/19
X:	CALL address(Rd)	`01011111_Rdnz_0000` `1 or 2 address words`	13/18/20

Notes:

1. The address loaded into the PC is the ***address*** of the destination operand, not the data ***at*** the destination address.

2. Bits 3-0 of the opcode are ignored.

3. Zilog Z8000 documentation lists 12/18/20 (for DA) and 13/18/21 (for X).

CALR

Call Relative

CALR dst dst: RA

Operation: SP <= SP - 2
@SP <= PC
PC <= PC - (2 x displacement)

Flags: **C:** Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
RA:	CALR address	`1101dddd_dddd_dddd`	10/14

Notes:

1. Zilog documentation incorrectly states that the displacement is ***added*** to the Program Counter. Zilog Z8000 devices actually ***subtract*** the displacement from the Program Counter. The Y8000 design matches this behavior

2. The PC used for the address calculation is the PC of the ***next*** instruction.

3. The displacement is a 12-bit twos-complement number in the range -2048 to +2047. Thus the destination must be in the range -4092 to +4098 from the address of the CALR instruction.

3. Zilog Z8000 documentation lists 10/15 clocks.

CLR

Clear

CLR dst
CLRB

dst: R, IR, DA, X

Operation: dst <= 0

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	CLR Rd CLRB Rbd	`1000110W_Rddd_1000`	7
IR:	CLR @Rd CLRB @Rd	`0000110W_Rdnz_1000`	8
DA:	CLR address CLRB address	`0100110W_0000_1000` `1 or 2 address words`	11/12/14
X:	CLR addr(Rd) CLRB addr(Rd)	`0100110W_Rdnz_1000` `1 or 2 address words`	12/12/15

COM

Complement

COM dst
COMB

dst: R, IR, DA, X

Operation: dst <= ~dst

Flags:
C: Unaffected.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if result parity is even; cleared otherwise (COMB); Unaffected (COM).
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	COM Rd COMB Rbd	`1000110W_Rddd_0000`	7
IR:	COM @Rd COMB @Rd	`0000110W_Rdnz_0000`	12
DA:	COM address COMB address	`0100110W_0000_0000` `1 or 2 address words`	15/16/18
X:	COM addr(Rd) COMB addr(Rd)	`0100110W_Rdnz_0000` `1 or 2 address words`	16/16/19

COMFLG

Complement Flag

COMFLG flags flag: C, Z, S, P, V

Operation: FCW[7:4] <= FCW[7:4] ^ inst[7:4]

Flags:
C: Complemented if specified; unaffected otherwise.
Z: Complemented if specified; unaffected otherwise.
S: Complemented if specified; unaffected otherwise.
V: Complemented if specified; unaffected otherwise.
D: Unaffected.
H: Complemented.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	COMFLG flags	`10001101_CZSV_0101`	7

Notes:

1. Zilog Z8000 documentation lists the H flag as undefined, when in fact it is always complemented. The Y8000 design matches this behavior.

CP

Compare Register

CP dst, src
CPB
CPL

dst: R
src: R, IR, DA, X

Operation: dst - src

Flags:
C: Set if arithmetic borrow from MSB; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	CP Rd, Rs CPB Rbd, Rbs	`1000101W_Rsss_Rddd`	4
	CPL RRd, RRs	`10010000_RRss_RRdd`	8
IM:	CP Rd, #data	`00001011_0000_Rddd` `#data`	7
	CPB Rbd, #data	`00001010_0000_Rbdd` `#data | #data`	7

	CPL RRd, #data	`00010000_0000_RRdd`
		`#data (high)`
		`#data (low)`
			14

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	CP Rd, @Rs	`0000101W_Rsnz_Rddd`	7
	CPB Rbd, @Rs	`00001010_Rsnz_Rbdd`	7
	CPL RRd, @Rs	`00010000_Rsnz_RRdd`	14
DA:	CP Rd, address CPB Rbd, address	`0100101W_0000_Rddd` `1 or 2 address words`	9/10/12
	CPL RRd, address	`01010000_0000_RRdd` `1 or 2 address words`	15/16/18
X:	CP Rd, addr(Rs) CPB Rbd, addr(Rs)	`0100101W_Rsnz_Rddd` `1 or 2 address words`	10/10/13
	CPL RRd, addr(Rs)	`01010000_Rsnz_RRdd` `1 or 2 address words`	16/16/19

CP

Compare Immediate

CP dst, src
CPB

dst: IR, DA, X
src: IM

Operation: dst - src

Flags:
C: Set if arithmetic borrow from MSB; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	CP @Rd, #data	00001101_Rdnz_0001 #data	11
	CPB @Rd, #data	00001100_Rdnz_0001 #data | #data	11
DA:	CP address, #data	01001101_0000_0001 1 or 2 address words #data	14/15/17
	CPB address, #data	01001100_0000_0001 1 or 2 address words #data | #data	14/15/17

Addressing Modes	Assembly Syntax	Encoding	Clocks
X:	CP (addr)Rd, #data	01001101_Rdnz_0001 1 or 2 address words #data	15/15/18
	CPB (addr)Rd, #data	01001100_Rdnz_0001 1 or 2 address words #data | #data	15/15/18

CPD

Compare and Decrement

CPD dst, src, r, cc
CPDB

dst: R
src: IR

Operation: dst - src
Rs <= (Word) ? Rs - 2 : Rs - 1
r <= r - 1

Flags:
C: Set if arithmetic borrow from MSB for compare; cleared otherwise.
Z: Set if flags match cc after the last compare; cleared otherwise.
S: Set if result is negative for compare; cleared otherwise.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	CPD Rd, @Rs, r, cc CPDB Rbd, @Rs, r, cc	1011101W_Rsnz_1000 0000_Rrrr_Rddd_cccc	20

Notes:

1. The C, Z, S and V flags are set as usual by the compare. This flag combination is used by the cc check. The Z and V flags are subsequently set per the instruction description.

2. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

3. Bits 15-12 of the second word of the opcode are ignored.

CPDR

Compare, Decrement and Repeat

CPDR dst, src, r, cc
CPDRB

dst: R
src: IR

Operation:
dst - src
Rs <= (Word) ? Rs - 2 : Rs - 1
r <= r - 1
repeat until cc is true or r = 0

Flags:
C: Set if arithmetic borrow from MSB for the last compare; cleared otherwise.
Z: Set if flags match cc after the last compare; cleared otherwise.
S: Set if result is negative for the last compare; cleared otherwise.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	CPDR Rd, @Rs, r, cc CPDRB Rbd, @Rs, r, cc	1011101W_Rsnz_1100 0000_Rrrr_Rddd_cccc	11 + 9n

---

Notes:

1. The C, Z, S and V flags are set as usual by the compare. This flag combination is used by the cc check. The Z and V flags are subsequently set per the instruction description.

2. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

3. Bits 15-12 of the second word of the opcode are ignored.

Notes (continued):

4. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

CPI

Compare and Increment

CPI dst, src, r, cc
CPIB

dst: R
src: IR

Operation: dst - src
Rs <= (Word) ? Rs + 2 : Rs + 1
r <= r - 1

Flags:
C: Set if arithmetic borrow from MSB for the last compare; cleared otherwise.
Z: Set if flags match cc after the last compare; cleared otherwise.
S: Set if result is negative for the last compare; cleared otherwise.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	CPI Rd, @Rs, r, cc CPIB Rbd, @Rs, r, cc	1011101W_Rsnz_0000 0000_Rrrr_Rddd_cccc	20

Notes:

1. The C, Z, S and V flags are set as usual by the compare. This flag combination is used by the cc check. The Z and V flags are subsequently set per the instruction description.

2. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

3. Bits 15-12 of the second word of the opcode are ignored.

CPIR

Compare, Increment and Repeat

CPIR dst, src, r, cc
CPIRB

dst: R
src: IR

Operation: dst - src
Rs <= (Word) ? Rs + 2 : Rs + 1
r <= r - 1
repeat until cc is true or r = 0

Flags:
C: Set if arithmetic borrow from MSB for the last compare; cleared otherwise.
Z: Set if flags match cc after the last compare; cleared otherwise.
S: Set if result is negative for the last compare; cleared otherwise.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	CPIR Rd, @Rs, r, cc CPIRB Rbd, @Rs, r, cc	1011101W_Rsnz_0100 0000_Rrrr_Rddd_cccc	11 + 9n

---

Notes:

1. The C, Z, S and V flags are set as usual by the compare. This flag combination is used by the cc check. The Z and V flags are subsequently set per the instruction description.

2. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

3. Bits 15-12 of the second word of the opcode are ignored.

Notes (continued):

4. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

CPSD

Compare String and Decrement

CPSD dst, src, r, cc
CPSDB

dst: IR
src: IR

Operation:

dst - src
Rs <= (Word) ? Rs - 2 : Rs - 1
Rd <= (Word) ? Rd - 2 : Rd - 1
r <= r - 1

Flags:

C: Set if arithmetic borrow from MSB for the last compare; cleared otherwise.
Z: Set if flags match cc after the last compare; cleared otherwise.
S: Set if result is negative for the last compare; cleared otherwise.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	CPSD @Rd, @Rs, r, cc CPSDB @Rd, @Rs, r, cc	1011101W_Rsnz_1010 0000_Rrrr_Rdnz_cccc	25

Notes:

1. The C, Z, S and V flags are set as usual by the compare. This flag combination is used by the cc check. The Z and V flags are subsequently set per the instruction description.

2. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

3. Bits 15-12 of the second word of the opcode are ignored.

CPSDR

Compare String, Decrement and Repeat

CPSDR dst, src, r, cc
CPSDRB

dst: IR
src: IR

Operation:
dst - src
Rs <= (Word) ? Rs - 2 : Rs - 1
Rd <= (Word) ? Rd - 2 : Rd - 1
r <= r - 1
repeat until cc is true or r = 0

Flags:
C: Set if arithmetic borrow from MSB for the last compare; cleared otherwise.
Z: Set if flags match cc after the last compare; cleared otherwise.
S: Set if result is negative for the last compare; cleared otherwise.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	CPSDR @Rd, @Rs, r, cc CPSDRB @Rd, @Rs, r, cc	1011101W_Rsnz_1110 0000_Rrrr_Rdnz_cccc	11 + 14n

---

Notes:

1. The C, Z, S and V flags are set as usual by the compare. This flag combination is used by the cc check. The Z and V flags are subsequently set per the instruction description.

2. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

3. Bits 15-12 of the second word of the opcode are ignored.

Notes (continued):

4. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

CPSI

Compare String and Increment

CPSI dst, src, r, cc
CPSIB

dst: IR
src: IR

Operation:
dst - src
Rs <= (Word) ? Rs + 2 : Rs + 1
Rd <= (Word) ? Rd + 2 : Rd + 1
r <= r - 1

Flags:
C: Set if arithmetic borrow from MSB for the last compare; cleared otherwise.
Z: Set if flags match cc after the last compare; cleared otherwise.
S: Set if result is negative for the last compare; cleared otherwise.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	CPSI @Rd, @Rs, r, cc CPSIB @Rd, @Rs, r, cc	1011101W_Rsnz_0010 0000_Rrrr_Rdnz_cccc	25

Notes:

1. The C, Z, S and V flags are set as usual by the compare. This flag combination is used by the cc check. The Z and V flags are subsequently set per the instruction description.

2. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

3. Bits 15-12 of the second word of the opcode are ignored.

CPSIR

Compare String, Increment and Repeat

CPSIR dst, src, r, cc
CPSIRB

dst: IR
src: IR

Operation:

dst - src
Rs <= (Word) ? Rs + 2 : Rs + 1
Rd <= (Word) ? Rd + 2 : Rd + 1
r <= r - 1
repeat until cc is true or r = 0

Flags:

C: Set if arithmetic borrow from MSB for the last compare; cleared otherwise.
Z: Set if flags match cc after the last compare; cleared otherwise.
S: Set if result is negative for the last compare; cleared otherwise.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	CPSIR @Rd, @Rs, r, cc CPSIRB @Rd, @Rs, r, cc	1011101W_Rsnz_0110 0000_Rrrr_Rdnz_cccc	11 + 14n

---

Notes:

1. The C, Z, S and V flags are set as usual by the compare. This flag combination is used by the cc check. The Z and V flags are subsequently set per the instruction description.

2. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

3. Bits 15-12 of the second word of the opcode are ignored.

Notes (continued):

4. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

DAB

Decimal Adjust

DAB dst dst: R

Operation: dst <= dst + da_value

Flags:
C: Set or cleared according to the table in the notes.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	DAB Rbd	`10110000_Rbdd_0000`	5

Notes:

1. This instruction should only be executed on the result of an ADDB, ADCB, SUBB, or SBCB instruction, and serves to convert the binary result into a BCD number.

2. Bits 3-0 of the opcode are ignored.

3. Note that the sign flag is not really meaningful for a BCD number.

4. The byte and flags are modified according to the table below:

Notes (continued):

Instruction	C before DAB	dst[7:4] before DAB	H before DAB	dst[3:0] before DAB	Number added to dst	C after DAB
	0	0-9	0	0-9	00	0
	0	0-8	0	A-F	06	0
	0	0-9	1	0-3	06	0
ADDB or ADCB	0	A-F	0	0-9	60	1
	0	9-F	0	A-F	66	1
	0	A-F	1	0-3	66	1
	1	0-2	0	0-9	60	1
	1	0-2	0	A-F	66	1
	1	0-3	1	0-3	66	1
	0	0-9	0	0-9	00	0
SUBB or SBCB	0	0-8	1	6-F	FA	0
	1	7-F	0	0-9	A0	1
	1	6-F	1	6-F	9A	1

DEC

Decrement

DEC dst, src
DECB

dst: R, IR, DA, X
src: IM

Operation: dst <= dst - src (src = 1 to 16, encoded in opcode)

Flags:
C: Unaffected.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	DEC Rd, #n DECB Rbd, #n	`1010101W_Rddd_nnnn`	4
IR:	DEC @Rd, #n DECB @Rd, #n	`0010101W_Rdnz_nnnn`	11
DA:	DEC address, #n DECB address, #n	`0110101W_0000_nnnn` `1 or 2 address words`	13/14/16
X:	DEC addr(Rd), #n DECB addr(Rd), #n	`0110101W_Rdnz_nnnn` `1 or 2 address words`	14/14/17

DI

Disable Interrupt

DI int — int: VI, NVI — **PRIVILEGED INSTRUCTION**

Operation: FCW[12:11] <= FCW[12:11] & inst[1:0]

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	DI int	01111100_0000_00VN	7

Notes:

1. The interrupt enable bits are updated before the instruction completes, so an interrupt that is asserted while being disabled by this instruction will not be accepted.

DIV
Divide

DIV dst, src
DIVL

dst: R
src: R, IM, IR, DA, X

Operation: Word: dst[31:0] / src[15:0]
dst[31:16] <= remainder
dst[15:0] <= quotient
Long: dst[63:0] / src[31:0]
dst[63:32] <= remainder
dst[31:0] <= quotient

Flags:
C: See tables below.
Z: Set if the quotient or divisor is zero; cleared otherwise.
S: See tables below.
V: See tables below.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	DIV RRd, Rs	`10011011_Rsss_RRdd`	6 + div
	DIVL RQd, RRs	`10011010_RRss_RQdd`	6 + divl
IM:	DIV RRd, #data	`00011011_0000_RRdd` `#data`	8 + div

	DIVL RQd, #data	`00011010_0000_RQdd`
		`#data (high)`
		`#data (low)`
			11 + div

IR:	DIV RRd, @Rs	`00011011_Rsnz_RRdd`	8 + div
	DIVL RQd, @Rs	`00011010_Rsnz_RQdd`	11 + divl

Addressing Modes	Assembly Syntax	Encoding	Clocks
DA:	DIV RRd, address	01011011_0000_RRdd 1 or 2 address words	10/11/13 + div
	DIVL RQd, address	01011010_0000_RQdd 1 or 2 address words	13/14/16 + divl
X:	DIV RRd, addr(Rs)	01011011_Rsnz_RRdd 1 or 2 address words	11/11/14 + div
	DIVL RQd, addr(Rs)	01011010_Rsnz_RQdd 1 or 2 address words	14/14/17 + divl

Notes:

1. Zilog Z8000 documentation incorrectly describes the condition used to detect overflow for a negative quotient. The Zilog documentation specifies that for DIV a quotient less than -(2^16) is an overflow, but Z8000 devices actually signal overflow if the quotient is less than -(2^16 - 1). In the case of DIVL the documentation specifies that a quotient less than -(2^32) is an overflow, but Z8000 devices actually signal overflow if the quotient is less than -(2^32 - 1). The Y8000 design matches the behavior of the Zilog devices.

2. Zilog Z8000 documentation states that in the case of overflow, the destination register(s) and S flag are undefined. The Y8000 design does not attempt to match the Z8000 microprocessor's undefined value for this case, and the Y8000 design always clears the S flag in this case.

3. All four numbers (divisor, divider, quotient and remainder) are signed twos-complement numbers. The remainder will always have the same sign as the divisor.

4. The quotient can be a 17-bit number for DIV and a 33-bit number for DIVL. The S flag is the MSB of the quotient in this case, and can be used to extend the quotient to a full 32 or 64 bits in software. The other flags indicate whether the S flag is actually needed to represent the number correctly.

Notes (continued):

5. The execution time ("div" in the Clocks column above) and different flag cases for DIV are shown in the table below:

DIV case	C	Z	S	V	clocks
a. Quotient is a 16-bit twos-complement number.	0	zero	sign	0	86
b. Divisor is zero. The destination register is unchanged in this case.	0	1	0	1	5
c. Quotient is too large to represent in 17 bits. The destination register is undefined in this case.	0	0	0	1	16
d. Quotient is a 17-bit twos-complement number.	1	0	sign	1	86

6. The execution time ("divl" in the Clocks column above) and different flag cases for DIVL are shown in the table below:

DIVL case	C	Z	S	V	clocks
a. Quotient is a 32-bit twos-complement number.	0	zero	sign	0	494
b. Divisor is zero. The destination register is unchanged in this case.	0	1	0	1	8
c. Quotient is too large to represent in 33 bits. The destination register is undefined in this case.	0	0	0	1	30
d. Quotient is a 33-bit twos-complement number.	1	0	sign	1	494

7. Because the Rs (or RRs) and RRd (or RQd) are used to store intermediate results, they must be separate and non-overlapping.

Notes (continued):

8, The Divide instruction timing differences are shown in the table below.

Divide Instruction Instruction	Y8000 timing			Z8000 documentation		
	normal	div-by-0	overflow	normal	div-by-0	overflow
DIV RRd, Rs	92	11	22	107	13	25
DIV RRd, #data	94	13	24	107	13	25
DIV RRd, @Rs	94	13	24	107	13	25
DIV RRd, address	96	15	26	108	14	26
DIV RRd, addr(Rs)	97	16	27	109	15	27
DIVL RQd, RRs	500	14	36	744	30	51
DIVL RQd, #data	505	19	41	744	30	51
DIVL RQd, @Rs	505	19	41	744	30	51
DIVL RQd, address	507	21	43	745	31	52
DIVL RQd, addr(Rs)	508	22	44	746	32	53

DJNZ

Decrement and Jump if Not Zero

DJNZ R,dst dst: RA
DBJNZ

Operation: R <= R - 1
if R != 0 the PC <= PC - (2 x displacement)

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
RA:	DJNZ R, address DBJNZ Rb, address	`1111Rrrr_Wddd_dddd`	11

Notes:

1. The PC used for the address calculation is the PC of the ***next*** instruction.

2. The displacement is a 7-bit unsigned number in the range 0 to 127. Thus the destination must be in the range -252 to +2 from the address of the DJNZ (or DBJNZ) instruction.

EI

Enable Interrupt

EI int | int: VI, NVI | **PRIVILEGED INSTRUCTION**

Operation: FCW[12:11] <= FCW[12:11] | ~inst[1:0]

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	EI int	`01111100_0000_01VN`	7

Notes:

1. The interrupt enable bits are updated before the instruction completes, so an interrupt that is asserted while being enabled by this instruction will be accepted.

EX
Exchange

EX dst, src
EXB

dst: R
src: R, IR, DA, X

Operation:
tmp <= src
src <= dst
dst <= tmp

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	EX Rd, Rs EXB Rbd, Rbs	`1010110W_Rsss_Rddd`	6
IR:	EX Rd, @Rs EXB Rbd, @Rs	`0010110W_Rsnz_Rddd`	12
DA:	EX Rd, address EXB Rbd, address	`0110110W_0000_Rddd` `1 or 2 address words`	15/16/18
X:	Ex, Rd, addr(Rs) EXB Rbd, addr(Rs)	`0110110W_Rsnz_Rddd` `1 or 2 address words`	16/16/19

EXTS

Extend Sign

EXTSB dst
EXTS
EXTSL

dst: R

Operation: Byte: dst[15:8] <= (dst[7]) ? 0xFF : 0x00
Word: dst[31:16] <= (dst[15]) ? 0xFFFF : 0x0000
Long: dst[63:32] <= (dst[31]) ? 0xFFFFFFFF : 0x00000000

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	EXTSB Rd	10110001_Rddd_0000	11
	EXTS RRd	10110001_RRdd_1010	11
	EXTSL RQd	10110001_RQdd_0111	11

HALT

Halt

HALT **PRIVILEGED INSTRUCTION**

Operation: Halt and wait for interrupt

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	HALT	01111010_0000_0000	8 + 3n

Notes:

1. Interrupts are sampled once during the initial eight-clock sequence and once during every three-clock internal operation cycle.

2. Bits 3-0 of the opcode are ignored.

3. If an interrupt is active at the time of the first sample, the instruction executes in eight clock cycles. If an interrupt is sampled during an internal operation cycle, a subsequent internal operation cycle is performed before starting the interrupt acknowledge cycle.

(S)IN

(Special) Input

IN dst, src **INB**	dst: R src: IR, DA	**PRIVILEGED INSTRUCTION**
SIN dst, src **SINB** dst, src	dst: R src: DA	

Operation: dst <= src

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	IN Rd, @Rs INB Rbd, @Rs	`0011110W_Rsnz_Rddd`	10
DA:	IN Rd, port INB Rbd, port SIN Rd, port SINB Rd, port	`0011101W_Rddd_010S` `port`	12

Notes:

1. Only the Transaction Status is different between IN and SIN.

2. The Clock numbers above include the one automatic WAIT state.

INC
Increment

INC dst, src
INCB

dst: R, IR, DA, X
src: IM

Operation: dst <= dst + src (src = 1 to 16, encoded in opcode)

Flags:
C: Unaffected.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	INC Rd, #n INCB Rbd, #n	`1010100W_Rddd_nnnn`	4
IR:	INC @Rd, #n INCB @Rd, #n	`0010100W_Rdnz_nnnn`	11
DA:	INC address, #n INCB address, #n	`0110100W_0000_nnnn` `1 or 2 address words`	13/14/16
X:	INC addr(Rd), #n INCB addr(Rd), #n	`0110100W_Rdnz_nnnn` `1 or 2 address words`	14/14/17

(S)IND

(Special) Input and Decrement

IND dst, src, r, cc
INDB
SIND
SINDB

dst: IR
src: IR

PRIVILEGED INSTRUCTION

Operation:
dst <= src
Rd <= (Word) ? Rd - 2 : Rd - 1
r <= r - 1

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	IND @Rd, @Rs, r INDB @Rd, @Rs, r SIND @Rd, @Rs, r SINDB @Rd, @Rs, r	0011101W_Rsnz_100S 0000_Rrrr_Rdnz_1000	21

Notes:

1. Zilog Z8000 documentation lists the Z flag as undefined. In the Y8000 design this flag is unaffected.

2. Only the Transaction Status is different between IND and SIND.

3. The Clock number above includes the one automatic WAIT state.

Notes (continued):

4. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

5. Bits 15-12 of the second word of the opcode are ignored. Bits 3-0 of the second word of the opcode are tested for "0000" to distinguish between repeating and non-repeating versions of the instruction.

(S)INDR

(Special) Input, Decrement and Repeat

INDR dst, src, r, cc
INDRB
SINDR
SINDRB

dst: IR
src: IR

PRIVILEGED INSTRUCTION

Operation:
dst <= src
Rd <= (Word) ? Rd - 2 : Rd - 1
r <= r - 1
repeat until r = 0

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	INDR @Rd, @Rs, r INDRB @Rd, @Rs, r SINDR @Rd, @Rs, r SINDR @Rd, @Rs, r	0011101W_Rsnz_100S 0000_Rrrr_Rdnz_0000	11 + 10n

---

Notes:

1. Zilog Z8000 documentation lists the Z flag as undefined. In the Y8000 design this flag is unaffected.

2. Only the Transaction Status is different between INDR and SINDR.

3. The Clock number above includes the one automatic WAIT state.

Notes (continued):

4. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

5. Bits 15-12 of the second word of the opcode are ignored. Bits 3-0 of the second word of the opcode are tested for "0000" to distinguish between repeating and non-repeating versions of the instruction.

6. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

(S)INI

(Special) Input and Increment

INI dst, src, r, cc
INIB
SINI
SINIB

dst: IR
src: IR

PRIVILEGED INSTRUCTION

Operation:
dst <= src
Rd <= (word) ? Rd + 2 : Rd + 1
r <= r - 1

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	INI @Rd, @Rs, r INIB @Rd, @Rs, r SINI @Rd, @Rs, r SINIB @Rd, @Rs, r	0011101W_Rsnz_000S 0000_Rrrr_Rdnz_1000	21

Notes:

1. Zilog Z8000 documentation lists the Z flag as undefined. In the Y8000 design this flag is unaffected.

2. Only the Transaction Status is different between INI and SINI.

3. The Clock number above includes the one automatic WAIT state.

Notes (continued):

4. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

5. Bits 15-12 of the second word of the opcode are ignored. Bits 3-0 of the second word of the opcode are tested for "0000" to distinguish between repeating and non-repeating versions of the instruction.

(S)INIR

(Special) Input, Increment and Repeat

INIR dst, src, r, cc	dst: IR	**PRIVILEGED**
INIRB	dst: IR	**INSTRUCTION**
SINIR		
SINIRB		

Operation:
dst <= src
Rd <= (Word) ? Rd + 2 : Rd + 1
r <= r - 1
repeat until r = 0

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	INIR @Rd, @Rs, r INIRB @Rd, @Rs, r SINIR @Rd, @Rs, r SINIR @Rd, @Rs, r	0011101W_Rsnz_000S 0000_Rrrr_Rdnz_0000	11 + 10n

---

Notes:

1. Zilog Z8000 documentation lists the Z flag as undefined. In the Y8000 design this flag is unaffected.

2. Only the Transaction Status is different between INIR and SINIR.

3. The Clock number above includes the one automatic WAIT state.

Notes (continued):

4. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

5. Bits 15-12 of the second word of the opcode are ignored. Bits 3-0 of the second word of the opcode are tested for "0000" to distinguish between repeating and non-repeating versions of the instruction.

6. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

IRET

Interrupt Return

IRET **PRIVILEGED INSTRUCTION**

Operation:

```
SP <= SP + 2
FCW <= @SP
SP <= SP + 2
if (!z8002) begin
   PCSEG <= @SP
   SP <= SP + 2
   end
PC <= @SP
SP <= SP + 2
```

Flags:
C: Loaded from stack.
Z: Loaded from stack.
S: Loaded from stack.
V: Loaded from stack.
D: Loaded from stack.
H: Loaded from stack.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	IRET	01111011_0000_0000	13/16

Notes:

1. The identifier that is pushed as part of the interrupt or trap acknowledge cycle is skipped (and thus is effectively discarded).

2. In the Zilog Z8001 the operation of IRET is undefined in Non-segmented mode. For the Y8000 operating in Z8001 mode this instruction is always executed in Segmented mode, irrespective of the state of the SEG bit in the FCW. This is consistent with the handling of interrupts in Z8001 mode.

JP
Jump

JP cc, dst dst: IR, DA, X

Operation: if cc is true: PC <= dst

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	JP cc, @Rd	`00011110_Rdnz_cccc`	7 or 10/11
DA:	JP cc, address	`01011110_0000_cccc` `1 or 2 address words`	7/8/10
X:	JP cc, address(Rd)	`01011110_Rdnz_cccc` `1 or 2 address words`	8/9/11

Notes:

1. The address loaded into the PC is the ***address*** of the destination operand, not the data ***at*** the destination address.

2. In the case of IR addressing the execution time is 7 clocks if cc is false (the jump is not taken) and 10/11 clocks if cc is true and the jump is taken. Zilog documentation lists execution times of 10/15 clocks if the jump is taken.

3. Zilog Z8000 documentation lists 8/8/11 for Indexed addressing.

JR

Jump Relative

JR cc, dst dst: IR, DA, X

Operation: if cc is true: PC <= PC + (2 x displacement)

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
RA:	JR cc, address	`1110cccc_dddd_dddd`	6

Notes:

1. The PC used for the address calculation is the PC of the ***next*** instruction.

2. The displacement is an 8-bit twos-complement number in the range -128 to +127. Thus the destination must be in the range -254 to +256 from the address of the JR instruction.

LD

Load Register

LD dst, src
LDB
LDL

dst: R
src: R, IR, DA, X, BA, BX

Operation: dst <= src

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	LD Rd, Rs LDB Rbd, Rbs	`1010000W_Rsss_Rddd`	3
	LDL RRd, RRs	`10010100_RRss_RRdd`	5
IR:	LD Rd, @Rs LDB Rbd, @Rs	`0010000W_Rsnz_Rddd`	7
	LDL RRd, @Rs	`00010100_Rsnz_RRdd`	11
DA:	LD Rd, address LDB Rbd, address	`0110000W_0000_Rddd` `1 or 2 address words`	9/10/12
	LDL RRd, address	`01010100_0000_RRdd` `1 or 2 address words`	12/13/15

Addressing Modes	Assembly Syntax	Encoding	Clocks
X:	LD Rd, addr(Rs) LDB Rbd, addr(Rs)	0110000W_Rsnz_Rddd 1 or 2 address words	10/10/13
	LDL RRd, addr(Rs)	01010100_Rsnz_RRdd 1 or 2 address words	13/13/16
BA:	LD Rd, Rs(#disp) LDB Rbd, Rs(#disp)	0011000W_Rsnz_Rddd displacement	14
	LDL RRd, Rs(#disp)	00110101_Rsnz_RRdd displacement	17
BX:	LD Rd, Rs(Rx) LDB Rbd, Rs(Rx)	0111000W_Rsnz_Rddd 0000_Rxxx_0000_0000	14
	LDL RRd, Rs(Rx)	01110101_Rsnz_RRdd 0000_Rxxx_0000_0000	17

Notes:

1. In the case of BX addressing only bits 11-8 of the second word of the opcode are used. All other bits in the second word of the opcode are ignored.

LD

Load Memory

LD dst, src
LDB
LDL

dst: IR, DA, X, BA, BX
src: R

Operation: dst <= src

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	LD @Rd, Rs LDB @Rd, Rbs	`0010111W_Rdnz_Rsss`	8
	LDL @Rd, RRs	`00011101_Rdnz_RRss`	11
DA:	LD address, Rs LDB address, Rbs	`0110111W_0000_Rsss` `1 or 2 address words`	11/12/14
	LDL address, RRs	`01011101_0000_RRss` `1 or 2 address words`	14/15/17
X:	LD addr(Rd), Rs LDB addr(Rd), Rbs	`0110111W_Rdnz_Rsss` `1 or 2 address words`	12/12/15
	LDL addr(Rd), RRs	`01011101_Rdnz_RRss` `1 or 2 addresswords2`	15/15/18

Addressing Modes	Assembly Syntax	Encoding	Clocks
BA:	LD Rd(#disp), Rs LDB Rd(#disp), Rbs	0011001W_Rdnz_Rsss displacement	14
	LDL Rd(#disp), RRs	00110111_Rdnz_RRss displacement	17
BX:	LD Rd(Rx), Rs LDB Rd(Rx), Rbs	0111001W_Rdnz_Rsss 0000_Rxxx_0000_0000	14
	LDL Rd(Rx), RRs	01110111_Rdnz_RRss 0000_Rxxx_0000_0000	17

Notes:

1. In the case of BX addressing only bits 11-8 of the second word of the opcode are used. All other bits in the second word of the opcode are ignored.

LD

Load Immediate Value

LD dst, src
LDB
LDL

dst: R, IR, DA, X
src: IM

Operation: dst <= src

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	LD Rd, #data	00100001_0000_Rddd #data	7
	LDB Rbd, #data	00100000_0000_Rbdd #data | #data	7
	LDB Rbd, #data	1100Rbdd_data8data	5
	LDL RRd, #data	00010100_0000_RRdd #data (high) #data (low)	11

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	LD @Rd, #data	00001101_Rdnz_0101 #data	11
	LDB @Rd, #data	00001100_Rdnz_0101 #data | #data	11
	LDL @Rd, #data	00001101_Rdnz_0111 #data (high) #data (low)	17
DA:	LD address, #data	01001101_0000_0101 1 or 2 address words #data	14/15/17
	LDB address, #data	01001100_0000_0101 1 or 2 address words #data | #data	14/15/17
X:	LD addr(Rd), #data	01001101_Rdnz_0101 1 or 2 address words #data	15/15/18
	LDB addr(Rd), #data	01001100_Rdnz_0101 1 or 2 address words #data | #data	15/15/18

Notes:

1. Two formats exist for LDB register. The single-word format executes two clock cycles faster than the two-word format and will normally be chosen by an assembler.

2. The instruction LDL @Rd, #data is not present in the original Z8000, but the opcode is backwards-compatible with the 32-bit version of the Z8000 architecture and its third-party clones.

LDA

Load Address

LDA dst, src

dst: R
src: DA, X, BA, BX

Operation: dst <= address of src

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
DA:	LDA Rd, address	01110110_0000_Rddd 1 or 2 address words	12/13/15
X:	LDA Rd, addr(Rs)	01110110_Rsnz_Rddd 1 or 2 address words	13/13/16
BA:	LDA Rd, Rs(#disp)	00110100_Rsnz_Rddd displacement	15
BX:	LDA Rd, Rs(Rx)	01110100_Rsnz_Rddd 0000_Rxxx_0000_0000	15

Notes:

1. In the case of BX addressing only bits 11-8 of the second word of the opcode are used. All other bits in the second word of the opcode are ignored.

2. In Segmented mode with DA or X addressing mode the entire second word of the instruction is loaded into the register that holds the segment number.

LDAR

Load Address Relative

LDAR dst, src

dst: R
src: RA

Operation: dst <= address of src

Flags:

C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
RA:	LDAR Rd, address	`00110100_0000_Rddd` `displacement`	15

LDCTL

Load Into Control Register

LDCTL dst, src

dst: control register
src: R

PRIVILEGED INSTRUCTION

Operation: dst <= src

Flags:
C: Loaded from source for FCW destination; unaffected otherwise.
Z: Loaded from source for FCW destination; unaffected otherwise.
S: Loaded from source for FCW destination; unaffected otherwise.
V: Loaded from source for FCW destination; unaffected otherwise.
D: Loaded from source for FCW destination; unaffected otherwise.
H: Loaded from source for FCW destination; unaffected otherwise.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	LDCTL FCW, Rs	01111101_Rsss_1010	7
	LDCTL REFRESH, Rs	01111101_Rsss_1011	7
	LDCTL NSP, Rs	01111101_Rsss_1111	7
	LDCTL NSPSEG, Rs	01111101_Rsss_1110	7
	LDCTL PSAP, Rs	01111101_Rsss_1101	7
	LDCTL PSAPSEG, Rs	01111101_Rsss_1100	7

Notes:

1. All unused Control Register bits should be written with zeros for compatibility.

2. In Z8002 mode both the NSPSEG and PSAPSEG registers can still be written.

LDCTL

Load From Control Register

LDCTL dst, src

dst: R
src: control register

PRIVILEGED INSTRUCTION

Operation: dst <= src

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	LDCTL Rd, FCW	`01111101_Rddd_0010`	7
	LDCTL Rd, REFRESH	`01111101_Rddd_0011`	7
	LDCTL Rd, NSP	`01111101_Rddd_0111`	7
	LDCTL Rd, NSPSEG	`01111101_Rddd_0110`	7
	LDCTL Rd, PSAP	`01111101_Rddd_0101`	7
	LDCTL Rd, PSAPSEG	`01111101_Rddd_0100`	7

Notes:

1. All unused Control Register bits return zeros.

2. In Z8002 mode both the NSPSEG and PSAPSEG registers can still be read.

LDCTLB

Load Into Flags Register

LDCTL dst, src

dst: flags
src: R

Operation: dst <= src

Flags:

C: Loaded from source.
Z: Loaded from source.
S: Loaded from source.
V: Loaded from source.
D: Loaded from source.
H: Loaded from source.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	LDCTLB FCW, Rbs	`10001100_Rbss_1001`	7

Notes:

1. All unused Flag Register bits should be written with zeros for compatibility.

LDCTLB

Load From Flags Register

LDCTLB dst, src

dst: R
src: flags

Operation: dst <= src

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	LDCTLB Rbd, FLAGS	`10001100_Rbdd_0001`	7

Notes:

1. All unused Flag Register bits return zeros.

LDCTLL

Load Into Long Control Register

LDCTLL dst, src

dst: control register
src: R

PRIVILEGED INSTRUCTION

Operation: dst <= src

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	LDCTLL NSP, RRs	10011101_RRss_0110	9
	LDCTLL PSAP, RRs	10011101_RRss_1100	9
	LDCTLL REFRESH, RRs	10011101_RRss_1011	9

Notes:

1. All unused Control Register bits should be written with zeros for compatibility.

2. These instructions, for NSP and PSAP, are not present in the original Z8000, but the opcodes are backwards-compatible with the 32-bit version of the Z8000 architecture and its third-party clones.

3. The ability to load the PSAP with a single instruction eliminates the problem that will occur if an interrupt is requested between the load of the two words of the PSAP.

4. The REFRESH version of this instruction is not present in the 32-bit version of the Z8000 architecture because no refresh mechanism is included there. The opcode shown is listed as unimplemented in that architecture.

LDCTLL

Load From Long Control Register

LDCTLL dst, src

dst: R
src: control register

PRIVILEGED INSTRUCTION

Operation: dst <= src

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	LDCTLL RRd, NSP	`10011111_RRdd_0110`	9
	LDCTLL RRd, PSAP	`10011111_RRdd_1100`	9
	LDCTLL RRd, REFRESH	`10011111_RRdd_1011`	9

Notes:

1. All unused Control Register bits return zeros.

2. These instructions, for NSP and PSAP, are not present in the original Z8000, but the opcodes are backwards-compatible with the 32-bit version of the Z8000 architecture and its third-party clones.

3. The REFRESH version of this instruction is not present in the 32-bit version of the Z8000 architecture because no refresh mechanism is included there. The opcode shown is listed as unimplemented in that architecture.

LDD

Load and Decrement

LDD dst, src, r, cc
LDDB

dst: IR
src: IR

Operation:
dst <= src
Rs <= (Word) ? Rs - 2 : Rs - 1
Rd <= (Word) ? Rd - 2 : Rd - 1
r <= r - 1

Flags:
C: Unaffected.
Z: Set if r is zero after the decrement; cleared otherwise.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	LDD @Rd, @Rs, r LDDB @Rd, @Rs, r	1011101W_Rsnz_1001 0000_Rrrr_Rdnz_1000	20

Notes:

1. The Z8000 documentation lists the Z flag as undefined, but actually it operates identically to the V flag. The Y8000 design matches this behavior.

2. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

3. Bits 15-12 of the second word of the opcode are ignored. Bits 3-0 of the second word of the opcode are tested for "0000" to distinguish between repeating and non-repeating versions of the instruction.

LDDR

Load, Increment and Repeat

LDDR dst, src, r, cc
LDDRB

dst: IR
src: IR

Operation:
dst <= src
Rs <= (Word) ? Rs -2 : Rs -1
Rd <= (Word) ? Rd - 2 : Rd - 1
r <= r - 1
repeat until r = 0

Flags:
C: Unaffected.
Z: Set if r is zero after the decrement; cleared otherwise.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	LDDR @Rd, @Rs, r LDDRB @Rd, @Rs, r	1011101W_Rsnz_1001 0000_Rrrr_Rdnz_0000	11 + 9n

---

Notes:

1. The Z8000 documentation lists the Z flag as undefined, but actually it operates identically to the V flag. The Y8000 design matches this behavior.

2. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

3. Bits 15-12 of the second word of the opcode are ignored. Bits 3-0 of the second word of the opcode are tested for "0000" to distinguish between repeating and non-repeating versions of the instruction.

Notes (continued):

4. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

LDI

Load and Increment

LDI dst, src, r, cc
LDIB

dst: IR
src: IR

Operation:
dst <= src
Rs <= (Word) ? Rs + 2 : Rs + 1
Rd <= (Word) ? Rd + 2 : Rd + 1
r <= r - 1

Flags:
C: Unaffected.
Z: Set if r is zero after the decrement; cleared otherwise.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	LDI @Rd, @Rs, r LDIB @Rd, @Rs, r	1011101W_Rsnz_0001 0000_Rrrr_Rdnz_1000	20

Notes:

1. The Z8000 documentation lists the Z flag as undefined, but actually it operates identically to the V flag. The Y8000 design matches this behavior.

2. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

3. Bits 15-12 of the second word of the opcode are ignored. Bits 3-0 of the second word of the opcode are tested for "0000" to distinguish between repeating and non-repeating versions of the instruction.

LDIR

Load, Increment and Repeat

LDIR dst, src, r, cc
LDIRB

dst: IR
src: IR

Operation:

```
dst <= src
Rs <= (Word) ? Rs + 2 : Rs + 1
Rd <= (Word) ? Rd + 2 : Rd + 1
r <= r - 1
repeat until r = 0
```

Flags:

C: Unaffected.
Z: Set if r is zero after the decrement; cleared otherwise.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	LDIR @Rd, @Rs, r LDIRB @Rd, @Rs, r	`1011101W_Rsnz_0001` `0000_Rrrr_Rdnz_0000`	11 + 9n

Notes:

1. The Z8000 documentation lists the Z flag as undefined, but actually it operates identically to the V flag. The Y8000 design matches this behavior.

2. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

3. Bits 15-12 of the second word of the opcode are ignored. Bits 3-0 of the second word of the opcode are tested for "0000" to distinguish between repeating and non-repeating versions of the instruction.

Notes (continued):

4. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

LDK

Load Constant

LDK dst, src

dst: R
src: IM

Operation: dst <= src (src = 0 to 15)

Flags:

C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	LDK Rd, #data	`10111101_Rddd_dddd`	5

LDM

Load Multiple - Registers From Memory

LDM dst, src, n

dst: R
src: IR, DA, X

Operation: dst <= src (n words)

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	LDM Rd, @Rs, #n	00011100_Rsnz_0001 0000_Rddd_0000_nnnn	11 + 3n
DA:	LDM Rd, address, #n	01011100_0000_0001 0000_Rddd_0000_nnnn 1 or 2 address words	14/15/17 + 3n
X:	LDM Rd, addr(Rs), #n	01011100_Rsnz_0001 0000_Rddd_0000_nnnn 1 or 2 addresswords	15/15/18 + 3n

---

Notes:

1. Registers are loaded starting with Rd and increasing. R0 follows R15 in the case of a wrap-around.

2. There are no restrictions on the Rd and Rs registers.

3. Bits 15-12 and 7-4 of the second word of the opcode are ignored.

LDM

Load Multiple - Memory From Registers

LDM dst, src, n

dst: IR, DA, X
src: R

Operation: dst <= src (n words)

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	LDM @Rd, Rs, #n	00011100_Rdnz_1001 0000_Rsss_0000_nnnn	11 + 3n
DA:	LDM address, Rs, #n	01011100_0000_1001 0000_Rsss_0000_nnnn 1 or 2 address words	14/15/17 + 3n
X:	LDM addr(Rd), Rs, #n	01011100_Rdnz_1001 0000_Rsss_0000_nnnn 1 or 2 address words	15/15/18 + 3n

---

Notes:

1. Registers are stored starting with Rd and increasing. R0 follows R15 in the case of a wrap-around.

2. There are no restrictions on the Rd and Rs registers.

3. Bits 15-12 and 7-4 of the second word of the opcode are ignored.

LDPS

Load Program Status

LDPS src src: IR, DA, X **PRIVILEGED INSTRUCTION**

Operation:
```
FCW <= @(src address)
if (! Z8002) begin
   PCSEG <= @(src address + 2)
   PC <= @(src address + 4)
   end
else PC <= @(src address +2)
```

Flags:
C: Loaded from src.
Z: Loaded from src.
S: Loaded from src.
V: Loaded from src.
D: Loaded from src.
H: Loaded from src.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	LDPS @Rs	00111001_Rsnz_0000	12/16
DA:	LDPS address	01111001_0000_0000 1 or 2 address words	16/20/22
X:	LDPS addr(Rs)	01111001_Rsnz_0000 1 or 2 address words	17/21/23

Notes:

1. The new contents of the FCW, in particular SEG and N/S, do not take effect until this instruction completes.

2. Bits 3-0 of the opcode are ignored.

3. Zilog Z8000 documentation lists 17/20/23 for Indexed addressing.

LDR

Load Relative Register

LDR dst, src
LDRB
LDRL

dst: R
src: RA

Operation: dst <= src

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
RA:	LDR Rd, address LDRB Rbd, address	0011000W_0000_Rddd displacement	14
	LDRL RRd, address	00110101_0000_RRdd displacement	17

Notes:

1. The PC used for the address calculation is the PC of the ***next*** instruction.

2. The displacement is a 16-bit twos-complement number in the range -32768 to +32767. Thus the source must be in the range -32766 to +32769 from the address of the LDR instruction. The 16-bit address calculation wraps around if necessary.

3. Data is read from the program address space.

LDR

Load Relative Memory

LDR dst, src
LDRB
LDRL

dst: RA
src: R

Operation: dst <= src

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
RA:	LDR address, Rs LDRB address, Rbs	0011001W_0000_Rsss displacement	14
	LDRL address, RRs	00110111_0000_RRss displacement	17

Notes:

1. The PC used for the address calculation is the PC of the ***next*** instruction.

2. The displacement is a 16-bit twos-complement number in the range -32768 to +32767. Thus the destination must be in the range -32766 to +32769 from the address of the LDR instruction. The 16-bit address calculation wraps around if necessary.

3. Data is written to the program address space.

MBIT

Multi-Micro Bit Test

MBIT **PRIVELEGED INSTRUCTION**

Operation: FCW[5] <= MIB

Flags:
C: Unaffected.
Z: Unaffected.
S: Set if MIB is High, cleared otherwise.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	MBIT	01111011_0000_1010	7

Notes:

1. The MIB input is sampled during the T1 state of this instruction.

MREQ

Multi-Micro Request

MREQ dst | dst: R | **PRIVILEGED INSTRUCTION**

Operation:

```
if (!MIB) begin
   FCW[6:5] <= 0x00    //request not signalled
   MOB <= 1
   end
else begin
   MOB <= 0
   dst <= dst - 1; repeat until dst = 0
   if (!MIB) FCW[6:5] <= 0x11    //request granted
   else begin
      FCW[6:5] <= 0x10    //request not granted
      MOB <= 1
      end
   end
```

Flags:

C: Unaffected.
Z: Set if request was signalled; cleared otherwise.
S: Set if request was signalled and granted; cleared otherwise.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	MREQ Rd	`01111011_Rddd_1101`	12 + 7n

Notes:

1. Rd is unchanged if no request was signalled, and zero if a request was signalled.

2. This instruction is not interruptible; use caution with the value in Rd.

MRES

Multi-Micro Reset

MRES **PRIVELEGED INSTRUCTION**

Operation: MOB <= High

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	MRES	01111011_0000_1001	5

Notes:

1. The MOB output changes state on the instruction boundary.

MSET

Multi-Micro Set

MSET **PRIVELEGED INSTRUCTION**

Operation: MOB <= Low

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	MSET	01111011_0000_1000	5

Notes:

1. The MOB output changes state on the instruction boundary.

MULT

Multiply

MULT dst, src
MULTL

dst: R
src: R, IM, IR, DA, X

Operation: Word: dst[31:0] <= dst[15:0] x src[15:0]
Long: dst[63:0] <= dst[31:0] x src[31:0]

Flags:
C: See tables below.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Cleared.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	MULT RRd, Rs	`10011001_Rsss_RRdd`	6 + mult
	MULTL RQd, RRs	`10011000_RRss_RQdd`	6 + multl
IM:	MULT RRd, #data	`00011001_0000_RRdd` `#data`	8 + mult

	MULTL RQd, #data	`00011000_0000_RQdd`
		`#data (high)`
		`#data (low)`
			11 + multl

IR:	MULT RRd, @Rs	`00011001_Rsnz_RRdd`	8 + mult
	MULTL RQd, @Rs	`00011000_Rsnz_RQdd`	11 + multl

Addressing Modes	Assembly Syntax	Encoding	Clocks
DA:	MULT RRd, address	01011001_0000_RRdd 1 or 2 address words	10 + mult
	MULTL RQd, address	01011000_0000_RQdd 1 or 2 address words	13 + mult
X:	MULT RRd, addr(Rs)	01011001_Rsnz_RRdd 1 or 2 address words	11 + mult
	MULTL RQd, addr(Rs)	01011000_Rsnz_RQdd 1 or 2 address words	14 + multl

Notes:

1. All three numbers (multiplicand, multiplier and product) are signed twos-complement numbers.

2. Because RRd (or RQd) and Rs (or RRs) are used to store intermediate results, they must be separate and non-overlapping.

3. The execution time and different C flag cases for MULT are shown in the table below:

MULT case	C	mult clocks
a. Product requires 32 bits.	1	63
b. Product is zero.	0	9
c. Product can be represented in 16 bits. Most significant word is merely sign- extension data.	0	63

Notes (continued):

4. The execution time and different C flag cases for MULTL are shown in the table below: The execution time for MULTL is data-dependent. The ***m*** in the table below is the number of ones in the ***absolute value*** of the multiplicand (dst[31:0]).

MULTL case	C	multl clocks
a. Product requires 64 bits.	1	252 + 4***m***
b. Product is zero.	0	15
c. Product can be represented in 32 bits. Most significant two words are merely sign-extension data.	0	252 + 4***m***

5. The Multiply instruction timing differences are shown in the table below.

Multiply instruction	**Y8000 timing**			**Z8000 documentation**		
	min	**max**	**zero result**	**min**	**max**	**zero result**
MULT RRd, Rs	69	69	15	70	70	18
MULT RRd, #data	71	71	17	70	70	18
MULT RRd, @Rs	71	71	17	70	70	18
MULT RRd, address	73	73	19	71	71	19
MULT RRd, addr(Rs)	74	74	20	72	72	20
MULTL RQd, RRs	262	386	21	289	506	30
MULTL RQd, #data	267	391	26	289	506	30
MULTL RQd, @Rs	267	391	26	289	506	30
MULTL RQd, address	269	393	28	290	507	31
MULTL RQd, addr(Rs)	270	394	29	291	508	32

NEG

Negate

NEG dst
NEGB

dst: R, IR, DA, X

Operation: dst <= 0 - dst

Flags:
C: Set if result is zero; cleared otherwise (indicating borrow).
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	NEG Rd NEGB Rbd	`1000110W_Rddd_0010`	7
IR:	NEG @Rd NEGB @Rd	`0000110W_Rdnz_0010`	12
DA:	NEG address NEGB address	`0100110W_0000_0010` `1 or 2 address words`	15/16/18
X:	NEG addr(Rd) NEGB addr(Rd)	`0100110W_Rdnz_0010` `1 or 2 address words`	16/16/19

NOP

No operation

NOP

Operation: none

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	NOP	10001101_0000_0111	7

OR

Logical OR

OR dst, src
ORB

dst: R
src: R, IM, IR, DA, X

Operation: dst <= dst | src

Flags:
C: Unaffected.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if result parity is even; cleared otherwise (ORB); Unaffected (OR).
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	OR Rd, Rs ORB Rbd, Rbs	`1000010W_Rsss_Rddd`	4
IM:	OR Rd, #data ORB Rbd, #data	`0000010W_0000_Rddd` `#data`	7
IR:	OR Rd, @Rs ORB Rbd, @Rs	`0000010W_Rsnz_Rddd`	7
DA:	OR Rd, address ORB Rbd, address	`0100010W_0000_Rddd` `1 or 2 address words`	9/10/12
X:	OR Rd, addr(Rs) ORB Rbd, addr(Rs)	`0100010W_Rsnz_Rddd` `address`	10/10/13

(S)OTDR

(Special) Output, Decrement and Repeat

OTDR dst, src, r	dst: IR	**PRIVILEGED**
OTDRB	dst: IR	**INSTRUCTION**
SOTDR		
SOTDRB		

Operation:
dst <= src
Rs <= (Word) ? Rs -2 : Rs -1
r <= r - 1
repeat until r = 0

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	OTDR @Rd, @Rs, r OTDRB @Rd, @Rs, r SOTDR @Rd, @Rs, r SOTDRB @Rd, @Rs, r	0011101W_Rsnz_101S 0000_Rrrr_Rdnz_0000	11 + 10n

---

Notes:

1. Zilog Z8000 documentation lists the Z flag as undefined. In the Y8000 design this flag is unaffected.

2. Only the Transaction Status is different between OTDR and SOTDR.

3. The Clock number above includes the one automatic WAIT state.

Notes (continued):

4. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

5. Bits 15-12 of the second word of the opcode are ignored. Bits 3-0 of the second word of the opcode are tested for "0000" to distinguish between repeating and non-repeating versions of the instruction.

6. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

(S)OTIR

(Special) Output, Increment and Repeat

OTIR dst, src, r	dst: IR	**PRIVILEGED**
OTIRB	dst: IR	**INSTRUCTION**
SOTIR		
SOTIRB		

Operation:
dst <= src
Rs <= (Word) ? Rs + 2 : Rs + 1
r <= r - 1
repeat until r = 0

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	OTIR @Rd, @Rs, r OTIRB @Rd, @Rs, r SOTIR @Rd, @Rs, r SOTIRB @Rd, @Rs, r	0011101W_Rsnz_001S 0000_Rrrr_Rdnz_0000	11 + 10n

---

Notes:

1. Zilog Z8000 documentation lists the Z flag as undefined. In the Y8000 design this flag is unaffected.

2. Only the Transaction Status is different between OTIR and SOTIR.

3. The Clock number above includes the one automatic WAIT state.

Notes (continued):

4. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

5. Bits 15-12 of the second word of the opcode are ignored. Bits 3-0 of the second word of the opcode are tested for "0000" to distinguish between repeating and non-repeating versions of the instruction.

6. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

(S)OUT

(Special) Output

OUT dst, src	dst: IR, DA	**PRIVILEGED**
OUTB	dst: R	**INSTRUCTION**
SOUT dst, src	dst: DA	
SOUTB	src: R	

Operation: dst <= src

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	OUT @Rd, Rs OUTB @Rd, Rbs	`0011111W_Rdnz_Rsss`	10
DA:	OUT port, Rst OUTB port, Rbs SOUT port, Rst SOUTB port, Rbs	`0011101W_Rsss_011S` `port`	12

Notes:

1. Zilog Z8000 documentation lists the Z flag as undefined. In the Y8000 design this flag is unaffected.

2. Only the Transaction Status is different between OUT and SOUT.

3. The Clock numbers above include the one automatic WAIT state.

(S)OUTD

(Special) Output and Decrement

OUTD dst, src, r	dst: IR	**PRIVILEGED**
OUTDB	dst: IR	**INSTRUCTION**
SOUTD		
SOUTDB		

Operation:
dst <= src
Rs <= (Word) ? Rs - 2 : Rs - 1
r <= r - 1

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	OUTD @Rd, @Rs, r OUTDB @Rd, @Rs, r SOUTD @Rd, @Rs, r SOUTDB @Rd, @Rs, r	0011101W_Rsnz_101S 0000_Rrrr_Rdnz_1000	21

Notes:

1. The Z8000 documentation lists the Z flag as undefined. In the Y8000 design this flag is unaffected.

2. Only the Transaction Status is different between OUTD and SOUTD.

3. The Clock number above includes the one automatic WAIT state.

Notes (continued):

4. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

5. Bits 15-12 of the second word of the opcode are ignored. Bits 3-0 of the second word of the opcode are tested for "0000" to distinguish between repeating and non-repeating versions of the instruction.

6. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

(S)OUTI

(Special) Output and Increment

OUTI dst, src, r	dst: IR	**PRIVILEGED**
OUTIB	dst: IR	**INSTRUCTION**
SOUTI		
SOUTIB		

Operation: dst <= src
Rs <= (Word) Rs + 2 : Rs + 1
r <= r - 1

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	OUTI @Rd, @Rs, r OUTIB @Rd, @Rs, r SOUTI @Rd, @Rs, r SOUTIB @Rd, @Rs, r	0011101W_Rsnz_001S 0000_Rrrr_Rdnz_1000	21

Notes:

1. The Z8000 documentation lists the Z flag as undefined. In the Y8000 design this flag is unaffected.

2. Only the Transaction Status is different between OUTI and SOUTI.

3. The Clock number above includes the one automatic WAIT state.

Notes (continued):

4. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

5. Bits 15-12 of the second word of the opcode are ignored. Bits 3-0 of the second word of the opcode are tested for "0000" to distinguish between repeating and non-repeating versions of the instruction.

6. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

POP

Pop

POP dst, src
POPL

dst: R, IR, DA, X
src: IR

Operation: dst <= src
Rs <= (Long) ? Rs + 4 : Rs + 2

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	POP Rd, @Rs	`10010111_Rsnz_Rddd`	8
	POPL RRd, @Rs	`10010101_Rsnz_RRdd`	12
IR:	POP @Rd, @Rs	`00010111_Rsnz_Rdnz`	12
	POPL @Rd, @Rs	`00010101_Rsnz_Rdnz`	19
DA:	POP address, @Rs	`01010111_Rsnz_0000` `1 or 2 address words`	16/16/18
	POPL address, @Rs	`01010101_Rsnz_0000` `1 or 2 address words`	23/23/25

Addressing Modes	Assembly Syntax	Encoding	Clocks
X:	POP addr(Rd), @Rs	01010111_Rsnz_Rdnz 1 or 2 address words	16/16/18
	POPL addr(Rd), @Rs	01010101_Rsnz_Rdnz 1 or 2 address words	23/23/25

Notes:

1. R0 cannot be used as a stack pointer. This limitation comes about because of the instruction encoding rules.

2. Because the Rd and Rs registers are changed by the instruction, they must be separate and non-overlapping.

3. Zilog documentation from 1983 lists execution times of 16/16/18 and 23/23/25 for X addressing mode. Zilog documentation from 1985 lists execution times of 16/16/19 and 23/23/26 for X addressing mode.

PUSH

Push

PUSH dst, src
PUSHL

dst: IR
src: R, IM, IR, DA, X

Operation:
tmp <= src
Rd <= (Long) ? Rd - 4 : Rd - 2
dst <= tmp

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	PUSH @Rd, Rs	`10010011_Rdnz_Rsss`	9
	PUSHL @Rd, RRs	`10010001_Rdnz_RRss`	12
IM:	PUSH @Rd, #data	`00001101_Rdnz_1001` `#data`	12

	PUSHL @Rd, #data	`00010001_Rdnz_0000`
		`#data (high)`
		`#data (low)`
			17

IR:	PUSH @Rd, @Rs	`00010011_Rdnz_Rsnz`	13
	PUSHL @Rd, @Rs	`00010001_Rdnz_Rsnz`	20

Addressing Modes	Assembly Syntax	Encoding	Clocks
DA:	PUSH @Rd, address	01010011_Rdnz_0000 1 or 2 address words	14/14/17
	PUSHL @Rd, address	01010001_Rdnz_0000 1 or 2 address words	21/21/23
X:	PUSH @Rd, addr(Rs)	01010011_Rdnz_Rsnz 1 or 2 address words	14/14/17
	PUSHL @Rd, addr(Rs)	01010001_Rdnz_Rsnz 1 or 2 address words	21/21/24

Notes:

1. R0 cannot be used as a stack pointer. This limitation comes about because of the instruction encoding rules.

2. Zilog Z8000 documentation states that the Rs and Rd registers must be separate and non-overlapping for PUSHL. This restriction is not present in the Y8000 design.

3. Zilog documentation from 1983 lists execution times of 14/14/17 and 21/21/24 for DA addressing mode. Zilog documentation from 1985 lists execution times of 14/14/16 and 21/21/23 for DA addressing mode.

4. The instruction PUSHL @Rd, #data is not present in the original Z8000, but the opcode is backwards-compatible with the 32-bit version of the Z8000 architecture and its third-party clones.

RES

Reset Bit Static

RES dst, src
RESB

dst: R, IR, DA, X
src: IM

Operation: dst[src] <= 0

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	RES Rd, #b RESB Rbd, #b	`1010001W_Rddd_bbbb`	4
IR:	RES @Rd, #b RESB @Rd, #b	`0010001W_Rdnz_bbbb`	11
DA:	RES address, #b RESB address, #b	`0110001W_0000_bbbb` `1 or 2 address words`	13
X:	RES addr(Rd), #b RESB addr(Rd), #b	`0110001W_Rdnz_bbbb` `1 or 2 address words`	14

Notes:

1. Only bits 2-0 of the opcode are used to select the bit in the case of RESB, and bit 3 of the opcode is ignored.

RES

Reset Bit Dynamic

RES dst, src
RESB

dst: R
src: R

Operation: dst[src] <= 0

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	RES Rd, Rs RESB Rbd, Rs	0010001W_0000_Rsss 0000_Rddd_0000_0000	10

Notes:

1. The Z8000 microprocessor restricts the source register to be one of R0 - R7 for RESB. This restriction does not apply to the Y8000 design. Any register may be used as the source.

2. Only bits 3-0 of the source operand are used for the bit select for RES; only bits 2-0 of the source operand are used for the bit select for RESB.

3. Only bits 11-8 of the second word of the opcode are used. All other bits in the second word of the opcode are ignored.

RESFLG

Reset Flag

RESFLG flags flag: C, Z, S, P, V

Operation: FCW[7:4] <= FCW[7:4] & ~inst[7:4]

Flags:
C: Cleared if specified; unaffected otherwise.
Z: Cleared if specified; unaffected otherwise.
S: Cleared if specified; unaffected otherwise.
V: Cleared if specified; unaffected otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	RESFLG flags	10001101_CZSV_0011	7

RET
Return

RET cc

Operation: if cc is true: PC <= @SP
SP <= SP + 2

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	RET cc	10011110_0000_cccc	7 or 10/13

Notes:

1. The execution time is 7 clocks if cc is false and 10/13 (NS/S) clocks if cc is true.

RL

Rotate Left

RL dst, src
RLB

dst: R
src: IM

Operation: Word: {C, dst[15:0]} <= {dst[15:0], dst[15]}; repeat if by 2
Byte: {C, dst[7:0]} <= {dst[7:0], dst[7]}; repeat if by 2

Flags:
C: Set if carry from MSB for last rotate; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow during rotate; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	RL Rd, #1 RLB Rbd, #1	`1011001W_Rddd_0000`	6
	RL Rd, #2 RLB Rbd, #2	`1011001W_Rddd_0010`	7

RLC

Rotate Left through Carry

RLC dst, src
RLCB

dst: R
src: IM

Operation: Word: {C, dst[15:0]} <= {dst[15:0], C}; repeat if by 2
Byte: {C, dst[7:0]} <= {dst[7:0], C}; repeat if by 2

Flags:
C: Set if carry from MSB for last rotate; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow during rotate; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	RLC Rd, #1 RLCB Rbd, #1	`1011001W_Rddd_1000`	6
	RLC Rd, #2 RLCB Rbd, #2	`1011001W_Rddd_1010`	7

RLDB

Rotate Left Digit

RLDB dst, src

dst: R
src: R

Operation: {dst[7:0], src[7:0]} <= {dst[7:4], src[7:0], dst[3:0]};

Flags:

C: Unaffected.
Z: Set if dst is zero after rotate; cleared otherwise.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	RLDB Rbd, Rbs	`10111110_Rbss_Rbdd`	9

RR

Rotate Right

RRC dst, src
RRCB

dst: R
src: IM

Operation: Word: {dst[15:0], C} <= {dst[0], dst[15:0]}; repeat if by 2
Byte: {dst[7:0], C} <= {dst[0], dst[7:0]}; repeat if by 2

Flags:
C: Set if carry from LSB for last rotate; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow during rotate; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	RR Rd, #1 RRB Rbd, #1	`1011001W_Rddd_0100`	6
	RR Rd, #2 RRB Rbd, #2	`1011001W_Rddd_0110`	7

RRC

Rotate Right through Carry

RRC dst, src
RRCB

dst: R
src: IM

Operation: Word: {dst[15:0], C} <= {C, dst[15:0]}; repeat if by 2
Byte: {dst[7:0], C} <= {C, dst[7:0]}; repeat if by 2

Flags:
C: Set if carry from LSB for last rotate; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow during rotate; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	RRC Rd, #1 RRCB Rbd, #1	`1011001W_Rddd_1100`	6
	RRC Rd, #2 RRCB Rbd, #2	`1011001W_Rddd_1110`	7

RRDB

Rotate Right Digit

RLDB dst, src

dst: R
src: R

Operation: {dst[7:0], src[7:0]} <= {dst[7:4], src[3:0], dst[3:0], src[7:4]};

Flags:

C: Unaffected.
Z: Set if dst is zero after rotate; cleared otherwise.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	RRDB Rbd, Rbs	`10111100_Rbss_Rbdd`	9

SBC

Subtract With Carry

SBC dst, src
SBCB

dst: R
src: R

Operation: dst <= dst - src - C

Flags:

C: Cleared if arithmetic carry from result MSB;
Set otherwise, indicating borrow.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow; cleared otherwise.
D: Set (SBCB);
Unaffected (SBC).
H: Cleared if arithmetic carry from bit 3;
Set otherwise, indicating borrow (SBCB);
Unaffected (SBC).

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SBC Rd, Rs SBCB Rbd, Rbs	`1011011W_Rsss_Rddd`	5

SC

System Call

SC src src: IM

Operation:

```
SP <= SP - 2
@SP <= PC
SP <= SP - 2
if (!z8002) begin
   @SP <= PCSEG
   SP <= SP - 2
   end
@SP <= FCW
SP <= SP - 2
@SP <= inst
if (!z8002) begin
   FCW <= @{PSAP, 0x1A}
   PCSEG <= @{PSAP, 0x1C}
   PC <= @{PSAP, 0x1E}
   end
else begin
   FCW <= @{PSAP, 0x0C}
   PC <= @{PSAP, 0x0E}
   end
```

Flags:

C: Loaded from SC Program Status Area entry.
Z: Loaded from SC Program Status Area entry.
S: Loaded from SC Program Status Area entry.
V: Loaded from SC Program Status Area entry.
D: Loaded from SC Program Status Area entry.
H: Loaded from SC Program Status Area entry.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IM:	SC #src	`01111111_ssssrccc`	33/39

Notes:

1. The System Stack Pointer is used, independent of the current System/Normal mode.

2. The current SEG bit is ignored; only the state of the `z8002` input determines whether the SC is executed in Segmented mode or Non-segmented mode.

3. Bits 7-0 of the opcode can be used by the trap handler software to encode the type of System Call being requested. The entire opcode is pushed onto the system stack.

SDA

Shift Dynamic Arithmetic

SDA dst, src
SDAB
SDAL

dst: R
src: R

Operation:

src zero:
  Word: {C, dst[15:0]} <= {C, dst[15:0]}
  Byte: {C, dst[7:0]} <= {C, dst[7:0]}
  Long: {C, dst[31:0]} <= {C, dst[31:0]}
src positive:
  Word: (do src times): {C, dst[15:0]} <= {dst[15:0], 0}
  Byte: (do src times): {C, dst[7:0]} <= {dst[7:0], 0}
  Long: (do src times): {C, dst[31:0]} <= {dst[31:0], 0}
src negative:
  Word: (do src times): {dst[15:0], C} <= {dst[15], dst[15:0]}
  Byte: (do src times): {dst[7:0], C} <= {dst[7], dst[7:0]}
  Long: (do src times): {dst[31:0], C} <= {dst[31], dst[31:0]}

Flags:

C: Unaffected (src zero, Long only); set if carry from LSB (src negative) or MSB (src positive) on last shift; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow during shift; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SDA Rd, Rs SDAB Rbd, Rs	1011001W_Rddd_1011 0000_Rsss_0000_0000	15 + 3n
	SDAL RRd, Rs	10110011_RRdd_1111 0000_Rsss_0000_0000	15 + 3n

Notes:

1. Zilog Z8000 documentation lists the C flag as undefined for zero shift. In fact it is always cleared for Byte or Word shifts of zero bit positions, and unaffected for a Long shift of zero bit positions. The Y8000 design matches this behavior

2. The shift count should be restricted to range from -16 to 16 for SDA, from -8 to 8 for SDAB and from -32 to 32 for SDAL. Values outside of these ranges will still execute, but the hardware truncates any out-of-range value to between -127 and +127.

3. Only bits 11-8 of the second word of the opcode are used. All other bits in the second word of the opcode are ignored.

SDL

Shift Dynamic Logical

SDL dst, src
SDLB
SDLL

dst: R
src: R

Operation:

src zero:
  Word: {C, dst[15:0]} <= {C, dst[15:0]}
  Byte: {C, dst[7:0]} <= {C, dst[7:0]}
  Long: {C, dst[31:0]} <= {C, dst[31:0]}
src positive:
  Word: (do src times): {C, dst[15:0]} <= {dst[15:0], 0}
  Byte: (do src times): {C, dst[7:0]} <= {dst[7:0], 0}
  Long: (do src times): {C, dst[31:0]} <= {dst[31:0], 0}
src negative:
  Word: (do src times): {dst[15:0], C} <= {0, dst[15:0]}
  Byte: (do src times): {dst[7:0], C} <= {0, dst[7:0]}
  Long: (do src times): {dst[31:0], C} <= {0, dst[31:0]}

Flags:

C: Unaffected (src zero); set if carry from LSB (src negative) or MSB (src positive) on last shift; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow during shift; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SDL Rd, Rs SDLB Rbd, Rs	1011001W_Rddd_0011 0000_Rsss_0000_0000	15 + 3n
	SDLL RRd, Rs	10110011_RRdd_0111 0000_Rsss_0000_0000	15 + 3n

Notes:

1. Zilog Z8000 documentation lists the V flag as undefined. In fact it signals arithmetic overflow, even though this is a logical operation. The Y8000 design matches this behavior.

1. Zilog Z8000 documentation lists the C flag as undefined for zero shift. In fact it is always cleared for Byte or Word shifts of zero bit positions, and unaffected for a Long shift of zero bit positions. The Y8000 design matches this behavior

3. The shift count should be restricted to range from -16 to 16 for SDL, from -8 to 8 for SDLB and from -32 to 32 for SDLL. Values outside of these ranges will still execute, but the hardware truncates any out-of-range value to between -127 and +127.

4. Only bits 11-8 of the second word of the opcode are used. All other bits in the second word of the opcode are ignored.

SET

Set Bit Static

SET dst, src
SETB

dst: R, IR, DA, X
src: IM

Operation: dst[src] <= 1

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SET Rd, #b SETB Rbd, #b	`1010010W_Rddd_bbbb`	4
IR:	SET @Rd, #b SETB @Rd, #b	`0010010W_Rdnz_bbbb`	11
DA:	SET address, #b SETB address, #b	`0110010W_0000_bbbb` `1 or 2 address words`	13/14/16
X:	SET addr(Rd), #b SETB addr(Rd), #b	`0110010W_Rdnz_bbbb` `1 or 2 address words`	14/14/17

Notes:

1. Only bits 2-0 of the opcode are used to select the bit in the case of SETB, and bit 3 of the opcode is ignored.

SET

Set Bit Dynamic

SET dst, src
SETB

dst: R
src: R

Operation: dst[src] <= 1

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SET Rd, Rs SETB Rbd, Rs	0010010W_0000_Rsss 0000_Rddd_0000_0000	10

Notes:

1. The Z8000 CPUs restrict the source register to be one of R0 - R7 for SETB. This restriction does not apply to the Y8000 design. Any register may be used as the source.

1. Only bits 3-0 of the source operand are used for the bit select for SET; only bits 2-0 of the source operand are used for the bit select for SETB.

2. Only bits 11-8 of the second word of the opcode are used. All other bits in the second word of the opcode are ignored.

SETFLG

Set Flag

SETFLG flags flag: C, Z, S, P, V

Operation: FCW[7:4] <= FCW[7:4] | inst[7:4]

Flags:
C: Set if specified; unaffected otherwise.
Z: Set if specified; unaffected otherwise.
S: Set if specified; unaffected otherwise.
V: Set if specified; unaffected otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	SETFLG flags	10001101_CZSV_0001	7

SLA

Shift Left Arithmetic

SLA dst, src
SLAB
SLAL

dst: R
src: IM

Operation: For src zero:
Word: {C, dst[15:0]} <= {C, dst[15:0]}
Byte: {C, dst[7:0]} <= {C, dst[7:0]}
Long: {C, dst[31:0]} <= {C, dst[31:0]}
Else:
Word: (do src times): {C, dst[15:0]} <= {dst[15:0], 0}
Byte: (do src times): {C, dst[7:0]} <= {dst[7:0], 0}
Long: (do src times): {C, dst[31:0]} <= {dst[31:0], 0}

Flags: **C:** Unaffected (src zero); set if carry from MSB on last shift; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow during shift; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SLA Rd, #b	10110011_Rddd_1001 b	13 + 3b
	SLAB Rbd, #b	10110010_Rbdd_1001 0 | b	13 + 3b
	SLAL RRd, #b	10110011_RRdd_1101 b	13 + 3b

Notes:

1. The shift count should be restricted to range from 0 to 16 for SLA, from 0 to 8 for SLAB and from 0 to 32 for SLAL. Values outside of these ranges will still execute, but the hardware limits any out-of-range value to 255.

SLL

Shift Left Logical

SLL dst, src
SLLB
SLLL

dst: R
src: IM

Operation: For src zero:
Word: {C, dst[15:0]} <= {C, dst[15:0]}
Byte: {C, dst[7:0]} <= {C, dst[7:0]}
Long: {C, dst[31:0]} <= {C, dst[31:0]}
Else:
Word: (do src times): {C, dst[15:0]} <= {dst[15:0], 0}
Byte: (do src times): {C, dst[7:0]} <= {dst[7:0], 0}
Long: (do src times): {C, dst[31:0]} <= {dst[31:0], 0}

Flags: **C:** Set if carry from MSB on last shift; cleared otherwise.
Unaffected (src zero).
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow during shift; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SLL Rd, #b	10110011_Rddd_0001 b	13 + 3b
	SLLB Rbd, #b	10110010_Rbdd_0001 0 | b	13 + 3b
	SLLL RRd, #b	10110011_RRdd_0101 b	13 + 3b

Notes:

1. Zilog Z8000 documentation lists the V flag as undefined. In fact it signals arithmetic overflow, even though this is a logical operation. The Y8000 design matches this behavior.

2. The shift count should be restricted to range from 0 to 16 for SLL, from 0 to 8 for SLLB and from 0 to 32 for SLLL. Values outside of these ranges will still execute, but the hardware limits any out-of-range value to 255.

SRA

Shift Right Arithmetic

SRA dst, src
SRAB
SRAL

dst: R
src: IM

Operation: Word: (do src times): {dst[15:0], C} <= {dst[15], dst[15:0]}
Byte: (do src times): {dst[7:0], C} <= {dst[7], dst[7:0]}
Long: (do src times): {dst[31:0], C} <= {dst[31], dst[31:0]}

Flags:
C: Set if carry from LSB on last shift; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Cleared.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SRA Rd, #b	10110011_Rddd_1001 -b	13 + 3b
	SRAB Rbd, #b	10110010_Rbdd_1001 0 | -b	13 + 3b
	SRAL RRd, #b	10110011_RRdd_1101 -b	13 + 3b

Notes:

1. The shift count should be restricted to range from 1 to 16 for SRA, from 1 to 8 for SRAB and from 1 to 32 for SRAL. Values outside of these ranges will still execute, but the hardware limits any out-of-range value to 255.

2. The ***b*** is negative in the opcode.

SRL

Shift Right Logical

SRL dst, src
SRLB
SRLL

dst: R
src: IM

Operation: Word: (do src times): {dst[15:0], C} <= {0, dst[15:0]}
Byte: (do src times): {dst[7:0], C} <= {0, dst[7:0]}
Long: (do src times): {dst[31:0], C} <= {0, dst[31:0]}

Flags:
C: Set if carry from LSB on last shift; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow during shift; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SRL Rd, #b	`10110011_Rddd_0001` `-b`	13 + 3b
	SRLB Rbd, #b	`10110010_Rbdd_0001` `0 | -b`	13 + 3b
	SRLL RRd, #b	`10110011_RRdd_0101` `-b`	13 + 3b

Notes:

1. Zilog Z8000 documentation lists the V flag as undefined. In fact it signals arithmetic overflow, even though this is a logical operation. The Y8000 design matches this behavior.

Notes (continued):

2. The shift count should be restricted to range from 1 to 16 for SRL, from 1 to 8 for SRLB and from 1 to 32 for SRLL. Values outside of these ranges will still execute, but the hardware limits any out-of-range value to 255.

3. The ***b*** is negative in the opcode.

SUB

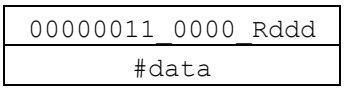
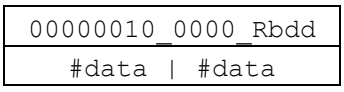
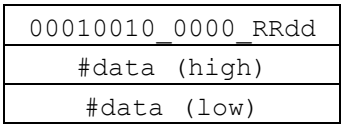
Subtract

SUB dst, src
SUBB
SUBL

dst: R
src: R, IM, IR, DA, X

Operation: dst <= dst - src

Flags:
C: Cleared if arithmetic carry from MSB;
Set otherwise, indicating borrow.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if arithmetic overflow; cleared otherwise.
D: Set (SUBB);
Unaffected (SUB, SUBL).
H: Cleared if arithmetic carry from bit 3;
Set otherwise, indicating borrow (SUBB);
Unaffected (SUB, SUBL).

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SUB Rd, Rs SUBB Rbd, Rbs		4
	SUBL RRd, RRs		8
IM:	SUB Rd, #data		7
	SUBB Rbd, #data		7
	SUBL RRd, #data		14

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	SUB Rd, @Rs SUBB Rbd, @Rs	`0000001W_Rsnz_Rddd`	7
	SUBL RRd, @Rs	`00010010_Rsnz_RRdd`	14
DA:	SUB Rd, address SUBB Rbd, address	`0100001W_0000_Rddd` `1 or 2 address words`	9/10/12
	SUBL RRd, address	`01010010_0000_RRdd` `1 or 2 address words`	15/16/18
X:	SUB Rd, addr(Rs) SUBB Rbd, addr(Rs)	`0100001W_Rsnz_Rddd` `1 or 2 address words`	10/10/13
	SUBL RRd, addr(Rs)	`01010010_Rsnz_RRdd` `1 or 2 address words`	16/16/19

TCC

Test Condition Code

TCC cc, dst
TCCB

dst: R

Operation: if cc true: dst[0] <= 1

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	TCC cc, Rd TCCB cc, Rbd	`1010111W_Rddd_cccc`	5

TEST

Test

TEST dst
TESTB
TESTL

dst: R, IR, DA, X

Operation: dst | 0

Flags:
C: Unaffected.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if result parity is even; cleared otherwise (TESTB); Unaffected (TEST or TESTL).
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	TEST Rd TESTB Rbd	`1000110W_Rddd_0100`	7
	TESTL RRd	`10011100_RRdd_1000`	9
IR:	TEST @Rd TESTB @Rd	`0000110W_Rdnz_0100`	8
	TESTL @Rd	`00011100_Rdnz_1000`	13
DA:	TEST address TESTB address	`0100110W_0000_0100` `1 or 2 address words`	11/12/14
	TESTL address	`01011100_0000_1000` `1 or 2 address words`	16/17/19

Addressing Modes	Assembly Syntax	Encoding	Clocks
X:	TEST addr(Rd) TESTB addr(Rd)	0100110W_Rdnz_0100 1 or 2 address words	12/12/15
	TESTL addr(Rd)	01011100_Rddd_1000 1 or 2 address words	17/17/20

Notes:

1. Zilog Z8000 documentation lists 13 clocks for TESTL of a register.

TRDB

Translate and Decrement

TRDB dst, src, r

dst: IR
src: IR

Operation:
dst <= src[dst]
Rd <= Rd - 1
r <= r - 1

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	TRDB @Rd, @Rs, r	10111000_Rdnz_1000 0000_Rrrr_Rsnz_0000	25

Notes:

1. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

2. Bits 15-12 and 3-0 of the second word of the opcode are ignored.

TRDRB

Translate, Decrement and Repeat

TRDRB dst, src, r

dst: IR
src: IR

Operation:
dst <= src[dst]
Rd <= Rd - 1
r <= r - 1
repeat until r = 0

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	TRDRB @Rd, @Rs, r	10111000_Rdnz_1100 0000_Rrrr_Rsnz_0000	11 + 14n

---

Notes:

1. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

2. Bits 15-12 and 3-0 of the second word of the opcode are ignored.

3. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

TRIB

Translate and Increment

TRIB dst, src, r

dst: IR
src: IR

Operation:
dst <= src[dst]
Rd <= Rd + 1
r <= r - 1

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	TRIB @Rd, @Rs, r	10111000_Rdnz_0000 0000_Rrrr_Rsnz_0000	25

Notes:

1. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

2. Bits 15-12 and 3-0 of the second word of the opcode are ignored.

TRIRB

Translate, Increment and Repeat

TRIRB dst, src, r

dst: IR
src: IR

Operation:
dst <= src[dst]
Rd <= Rd + 1
r <= r - 1
repeat until r = 0

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	TRIRB @Rd, @Rs, r	`10111000_Rdnz_0100` `0000_Rrrr_Rsnz_0000`	11 + 14n

Notes:

1. Because the Rd, Rs and r registers are changed by the instruction, they must be separate and non-overlapping.

2. Bits 15-12 and 3-0 of the second word of the opcode are ignored.

3. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

TRTDB

Translate, Test and Decrement

TRTDB src1, src2, r

src1: IR
src2: IR

Operation:
RH1 <= src2[src1] | 0
Rs1 <= Rs1 - 1
r <= r - 1

Flags:
C: Unaffected.
Z: Set if the translated value is zero; cleared otherwise.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	TRTDB @Rs1, @Rs2, r	10111000_R1nz_1010 0000_Rrrr_R2nz_0000	25

Notes:

1. Because the Rs1, Rs2 and r registers are changed by the instruction, they must be separate and non-overlapping.

2. Bits 15-12 and 3-0 of the second word of the opcode are ignored.

TRTDRB

Translate, Test, Decrement and Repeat

TRTDRB src1, src2, r

src1: IR
src2: IR

Operation:
RH1 <= src2[src1] | 0
Rs1 <= Rs1 - 1
r <= r - 1
repeat until translated value is zero or r is zero after decrement; cleared otherwise.

Flags:
C: Unaffected.
Z: Set if the translated value is zero; cleared otherwise.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	TRTDRB @Rs1, @Rs2, r	10111000_R1nz_1110 0000_Rrrr_R2nz_1110	11 + 14n

---

Notes:

1. Because the Rs1, Rs2 and r registers are changed by the instruction, they must be separate and non-overlapping.

2. Bits 15-12 and 3-0 of the second word of the opcode are ignored.

3. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

TRTIB

Translate, Test and Increment

TRTIB src1, src2, r

src1: IR
src2: IR

Operation:
RH1 <= src2[src1] | 0
Rs1 <= Rs1 + 1
r <= r - 1

Flags:
C: Unaffected.
Z: Set if the translated value is zero; cleared otherwise.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	TRTIB @Rs1, @Rs2, r	10111000_R1nz_0010 0000_Rrrr_R2nz_0000	25

Notes:

1. Because the Rs1, Rs2 and r registers are changed by the instruction, they must be separate and non-overlapping.

2. Bits 15-12 and 3-0 of the second word of the opcode are ignored.

TRTIRB

Translate, Test, Increment and Repeat

TRTIRB src1, src2, r

src1: IR
src2: IR

Operation:
RH1 <= src2[src1] | 0
Rs1 <= Rs1 + 1
r <= r - 1
repeat until translated value is zero or r is zero after decrement; cleared otherwise.

Flags:
C: Unaffected.
Z: Set if the translated value is zero; cleared otherwise.
S: Unaffected.
V: Set if r is zero after the decrement; cleared otherwise.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	TRTIRB @Rs1, @Rs2, r	10111000_R1nz_0110 0000_Rrrr_R2nz_1110	11 + 14n

---

Notes:

1. Because the Rs1, Rs2 and r registers are changed by the instruction, they must be separate and non-overlapping.

2. Bits 15-12 and 3-0 of the second word of the opcode are ignored.

3. This instruction samples interrupts during each iteration. If an interrupt is pending, the instruction is stopped and the interrupt accepted. The PC saved during the interrupt acknowledge cycle in this case is the PC of the running instruction, allowing the instruction to restart after the interrupt has been serviced.

TSET

Test and Set

TSET dst
TSETB

dst: R, IR, DA, X

Operation: Word: S <= dst[15], dst <= 0xFFFF
Byte: S <= dst[7], dst <= 0xFF

Flags:
C: Unaffected.
Z: Unaffected.
S: Set if MSB of dst was 1; cleared otherwise.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	TSET Rd TSETB Rbd	1000110W_Rddd_0110	7
IR:	TSET @Rd TSETB @Rd	0000110W_Rdnz_0110	11
DA:	TSET address TSETB address	0100110W_0000_0110 1 or 2 address words	14
X:	TSET addr(Rd) TSETB addr(Rd)	0100110W_Rdnz_0110 1 or2 address words	15

Notes:

1. BUSREQ is not accepted between the read and write of the destination.

2. The Z8003/4 CPUs signal a special status code (0xF) for the read and write of a TSET instruction. This is a logic compile option in the Y8000 design.

XOR

Logical Exclusive-OR

XOR dst, src
XORB

dst: R
src: R, IM, IR, DA, X

Operation: dst <= dst ^ src

Flags:
C: Unaffected.
Z: Set if result is zero; cleared otherwise.
S: Set if result is negative; cleared otherwise.
V: Set if result parity is even; cleared otherwise (XORB); Unaffected (XOR).
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	XOR Rd, Rs XORB Rbd, Rbs	`1000100W_Rsss_Rddd`	4
IM:	XOR Rd, #data	`00001001_0000_Rddd` `#data`	7
	XORB Rbd, #data	`00001000_0000_Rddd` `#data | #data`	7
IR:	XOR Rd, @Rs XORB Rbd, @Rs	`0000100W_Rsnz_Rddd`	7
DA:	XOR Rd, address XORB Rbd, address	`0100100W_0000_Rddd` `1 or 2 address words`	9/10/12
X:	XOR Rd, addr(Rs) XORB Rbd, addr(Rs)	`0100100W_Rsnz_Rddd` `1 or 2 address words`	10/10/13

Extended Instruction

Load from EPU

LD dst, EPU, #n dst: R, IR, DA, X

Operation: dst <= EPU (n words)

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	LD Rd, EPU, #n	`10001111_xxxx_00xx` `xxxx_Rddd_xxxx_nnnn`	11 + 4n
IR:	LD @Rd, EPU, #n	`0000111W_Rdnz_11xx` `xxxxxxxx_xxxx_nnnn`	11 + 3n

DA:	LD address, EPU, #n	`0100111W_0000_11xx`
		`xxxxxxxx_xxxx_nnnn`
		`1 or 2 address words`
			14/15/17 + 3n

X:	LD addr(Rd), EPU, #n	`0100111W_Rdnz_11xx`
		`xxxxxxxx_xxxx_nnnn`
		`1 or 2 address words`
			15/15/18 + 3n

Notes:

1. Bits 1-0 of the first word of the opcode are ignored by the processor, but normally select one of up to four EPUs in the system.

Notes (continued):

2. EPU data is stored at the destination address and increasing addresses. CPU Registers are loaded starting with Rd and increasing. R0 follows R15 in the case of a wrap-around.

3. Bits marked as "x" are ignored by the CPU but may be used by the EPU.

Extended Instruction

Load to EPU

LD EPU, src, #n src:R, IR, DA, X

Operation: EPU <= src (n words)

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	LD EPU, Rs, #n	10001111_xxxx_10xx xxxx_Rsss_xxxx_nnnn	11 + 4n
IR:	LD EPU, @Rs, #n	0000111W_Rsnz_01xx xxxxxxxx_xxxx_nnnn	11 + 3n
DA:	LD EPU, address, #n	0100111W_0000_01xx xxxxxxxx_xxxx_nnnn 1 or 2 address words	14/15/17 + 3n
X:	LD EPU, addr(Rs), #n	0100111W_Rsnz_01xx xxxxxxxx_xxxx_nnnn 1 or 2 address words	15/15/18 + 3n

---

Notes:

1. Bits 1-0 of the first word of the opcode are ignored by the processor, but normally select one of up to four EPUs in the system.

Notes (continued):

2. EPU data is read from the source address and increasing addresses. CPU Registers are stored starting with Rs and increasing. R0 follows R15 in the case of a wrap-around.

3. Bits marked as "x" are ignored by the CPU but may be used by the EPU.

Extended Instruction

Load FCW from EPU

LDB dst, EPU dst: Flags

Operation: FCW[7:0] <= EPU

Flags:
C: Loaded from EPU.
Z: Loaded from EPU.
S: Loaded from EPU.
V: Loaded from EPU.
D: Loaded from EPU.
H: Loaded from EPU.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	LDB FCW, EPU	`10001110_xxxx_00xx` `xxxx_xxxx_xxxx_0000`	15

Notes:

1. Bits 1-0 of the first word of the opcode are ignored by the processor, but normally select one of up to four EPUs in the system.

2. Bits marked as "x" are ignored by the CPU but may be used by the EPU.

Extended Instruction

Load EPU from FCW

LDB EPU, src src: Flags

Operation: EPU <= FCW[7:0]

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	LDB EPU, FCW	10001110_xxxx_10xx xxxx_xxxx_xxxx_0000	15

Notes:

1. Bits 1-0 of the first word of the opcode are ignored by the processor, but normally select one of up to four EPUs in the system.

2. Bits marked as "x" are ignored by the CPU but may be used by the EPU.

Extended Instruction

Internal EPU Operation

EPUI

Operation: EPU internal operation

Flags:
C: Unaffected.
Z: Unaffected.
S: Unaffected.
V: Unaffected.
D: Unaffected.
H: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	EPUI	`10001110_xxxx_01xx` `xxxx_xxxx_xxxx_xxxx`	14

Notes:

1. Bits 1-0 of the first word of the opcode are ignored by the processor, but normally select one of up to four EPUs in the system.

2. All sixteen bits of the second word of the opcode are ignored by the Y8000 CPU but may be used by the EPU. Z8000 documentation shows "`nnnn`" for bits 3-0 of the second word, encoding a loop count. The Z8070 FPU does not use these bits for a loop count, and the Y8000 ignores these four bits.

External Interface/Timing

This chapter presents external interface signals and timing for the Y8000 design. The user has the option of selecting either the normal multiplexed address/data implementation of the design or a special non-multiplexed address/data implementation. The table below shows the external interface signals available for both versions.

Name	Description	Direction	Active	NonMu	Muxed
AD[15:0]	Address/Data Bus	Bidirectional	High		X
ADDR[15:0]	Address Bus	Output	High	X	
ADOUT_EN	Address Output Enable	Output	High	X	
ASB	Address Strobe	Output	Low		X
B_WB	Byte/Word	Output	Low = W	X	X
BUSACKB	Bus Acknowledge	Output	Low	X	X
BUSREQB	Bus Request	Input	Low	X	X
CLK	Processor Clock	Input		X	X
DIN	Data Input	Input	High	X	
DOUT	Data Output	Output	High	X	
DSB	Data Strobe	Output	Low	X	X
EPA	EPU Available	Output	High	X	X
MIB	Multi-micro In	Input	Low	X	X
MOB	Multi-micro Out	Output	Low	X	X
MREQB	Memory Request	Output	Low	X	X
N_SB	Normal/System	Output	Low = Sys	X	X
NMIB	Non-maskable Interrupt Req	Input	Low	X	X
NVIB	Non-vectored Interrupt Req	Input	Low	X	X
R_WB	Read/Write	Output	Low = Wr	X	X
RESETF	Device Reset	Input	Low	X	X
SEGTB	Segment Fault Request	Input	Low	X	X
SN[7:0]	Segment Number	Output	High	X	X
ST[3:0]	Status	Output	High	X	X
STOPB	Stop Processor	Input	Low	X	X
T1_CYCLE	T1 Cycle Identifier	Output	High	X	
VIB	Vectored Interrupt Request	Input	Low	X	X
WAITB	Wait Request	Input	Low	X	X

The user has two further options beyond the Multiplexed/Non-multiplexed decision mentioned previously. Selecting the ASIC option implements the normal high-impedance operation for those Y8000 outputs that require the ability to go high-impedance. Because implementing any design in an FPGA can be a problem when 3-state signals are present, the FPGA option removes this capability, and the user must implement this functionality outside of the Y8000 implementation. These options are described more fully in the chapter that covers the actual Verilog code for the design. The Y8000 interface signals are:

- AD[15:0] (Address/Data). These signals are the multiplexed address and data bus. This bus is driven during the address portion of a bus transaction, and during the data portion of a write bus transaction. The bus is high-impedance at all other times, including during a Bus Acknowledge.

- ADDR[15:0] (Address). These signals are the address bus in the non-multiplexed option. They are always driven, but are guaranteed valid only during the portion of the bus transaction signalled by the T1 Cycle signal.

- ADOUT_EN (Address Output Enable). This signal is present only in the non-multiplexed option and signals when the normal Address/Data bus should be driven. This is useful when recreating the multiplexed bus interface externally to the Y8000 design itself.

- ASB (Address Strobe). This signal indicates that the address on the Address/Data bus, the Transaction Status, and the signals Normal/System, Read/Write and Byte/Word are all valid and signals the start of a bus transaction. The rising edge of Address Strobe can be used to latch the address for use throughout the transaction. This signal is high-impedance during a Bus Acknowledge (except in the FPGA option).

- B_WB (Byte/Word). This signal indicates whether the current bus transaction is a byte (high) or word (low) transaction, and is valid throughout the bus transaction. This signal is high-impedance during a Bus Acknowledge (except in the FPGA option).

- BUSACKB (Bus Request Acknowledge). This signal is asserted by the processor in response to the bus request signal being asserted. It indicates, when active, that the processor has given up control of the bus and its control signals.

- BUSREQB (Bus Request). This signal is asserted by an external device to indicate that the external device wants to take control of the processor bus and its associated control signals.

- CLK (System Clock). The master clock for all device timing is applied to this input.

- DIN[15:0] (Data Input). These sixteen lines are the data input bus in the non-multiplexed option. This bus is sampled at the normal time during read bus transactions.

- DOUT[15:0] (Data Output). These signals are the data output bus in the non-multiplexed option. This bus is valid during the portion of a write bus transaction where the multiplexed address/data bus would normally contain data.

- DSB (Data Strobe). This signal provides the timing for moving data to or from the processor during a bus transaction. It indicates that valid and stable data is present on the bus. This signal is high-impedance during a Bus Acknowledge (except in the FPGA option).

- EPA (Extended Processing Unit Available). This signal is just the EPA bit in the FCW. It signals that the CPU will not trap on EPU instructions so that an external EPU can execute these instructions.

- MIB (Multi-Micro In). The multi-micro input is asserted by another device on the processor bus to request exclusive use of a system resource.

- MOB (Multi-Micro Out). This signal is asserted by the processor to request exclusive control of a system resource.

- MREQB (Memory Request). This signal is asserted during memory transactions, with slightly different timing than DSB. It provides an extra edge for interfacing to DRAM, for example. This signal is high-impedance during a Bus Acknowledge (except in the FPGA option).

- N_SB (Normal/System). This signal indicates whether the processor is operating in Normal (high) or System (low) mode and is valid throughout the bus transaction. This signal is high-impedance during a Bus Acknowledge (except in the FPGA option).

- NMIB (Non-Maskable Interrupt Request). A high-to-low transition on this signal requests a non-maskable interrupt. This interrupt cannot be disabled by software.

- NVIB (Non-Vectored Interrupt Request). This signal is asserted by a peripheral device to request a non-vectored interrupt. The request should remain active until acknowledged by the processor. The non-vectored interrupt can be enabled or disabled by the processor.

- R_WB (Read/Write). This signal indicates that the current bus transaction is a read (high) or write (low) transaction. This signal is valid throughout the bus transaction and is high-impedance during a Bus Acknowledge (except in the FPGA option).

- RESETF (Reset). Asserting this signal forces the device into a known state, after which it will start up by fetching the starting FCW, starting PC Segment (in Z8001 mode) and starting PC and POC Segment program memory space. Reset must be asserted for at

least three clock cycles to be recognized, and the device is held in the reset state as long as Reset is asserted.

- SEGTB (Segment Trap Request) This signal is asserted by an external Memory Management Unit (MMU) to signal a segmentation fault.. The request should remain active until acknowledged by the processor. This request is masked when operating in Z8002 mode.

- SN[7:0] (Segment Number) This bus carries the Segment Number when the processor is operating in Segmented Mode. This bus is valid one clock cycle before the normal start of a bus transaction to allow time for address translation. This bus is high-impedance during a Bus Acknowledge (except in the FPGA option). SN[7] is a backwards-compatible enhancement.

- ST[3:0] (Transaction Status). The bus encodes the transaction type for the current bus transaction according to the table below. These signals are valid throughout a bus transaction. They are high-impedance during a Bus Acknowledge (except in the FPGA option).

ST[3:0]	Bus Transaction Type
0000	Internal Operation
0001	Refresh
0010	Standard I/O
0011	Special I.//O
0100	Segment Trap Acknowledge
0101	Non-Maskable Interrupt Acknowledge
0110	Non-Vectored Interrupt Acknowledge
0111	Vectored Interrupt Acknowledge
1000	Data Address Space
1001	Stack Address Space
1010	EPU Transfer, Data Address Space
1011	EPU Transfer, Stack Address Space
1100	Program Address Space
1101	Program Address Space, First Word of Instruction
1110	EPU/CPU Transfer
1111	Test and Set Data Access

- STOPB (Stop Request). Asserting this signal causes the processor to suspend execution and issue a continuous stream of Internal Operation bus transactions until the signal is de-asserted. These Internal Operation bus transactions are issued immediately following an Instruction Fetch 1 bus transaction only.

- T1_CYCLE (T1 Cycle). This signal is present only in the non-multiplexed option and signals the start of a bus transaction and that the address is valid on the Address Bus. This is useful when recreating the multiplexed bus interface externally to the Y8000.

- VIB (Vectored Interrupt Request). This signal is asserted by a peripheral device to request a vectored interrupt. The request should remain active until acknowledged by the processor. The vectored interrupt can be enabled or disabled by the processor.

- WAITB (Wait Request). This signal is asserted by devices on the system bus to notify the processor that additional time is needed to complete the current bus transaction.

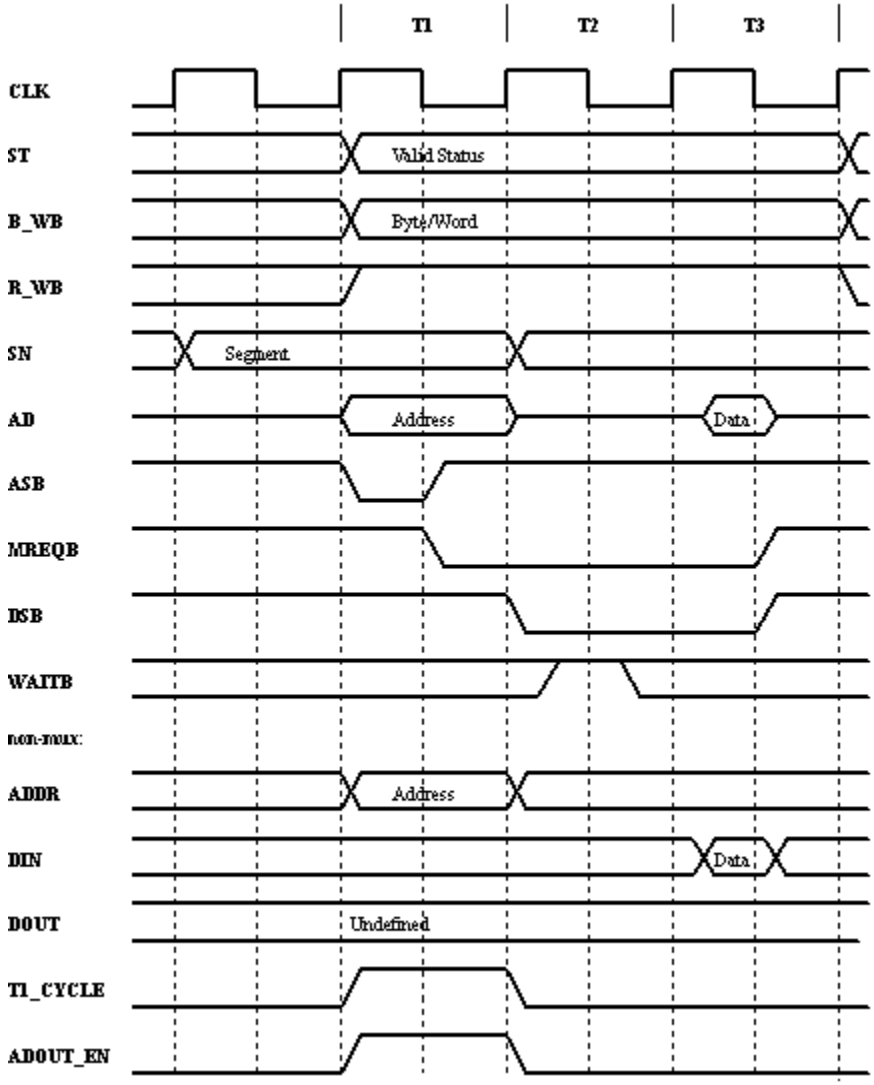
- Z8002 (Non-segmented CPU). This signal is used to select between Z8002 operation (High) and Z8001 operation (Low). This signal is assumed to be static.

All Y8000 signals come either directly from the output of a flip-flop, or through one or two gate delays from the output of a flip-flop. All outputs that come from logic are designed to be glitch-free. The lack of substantial clock-to-signal delays for outputs is different from the original Z8000 design and may affect system design.

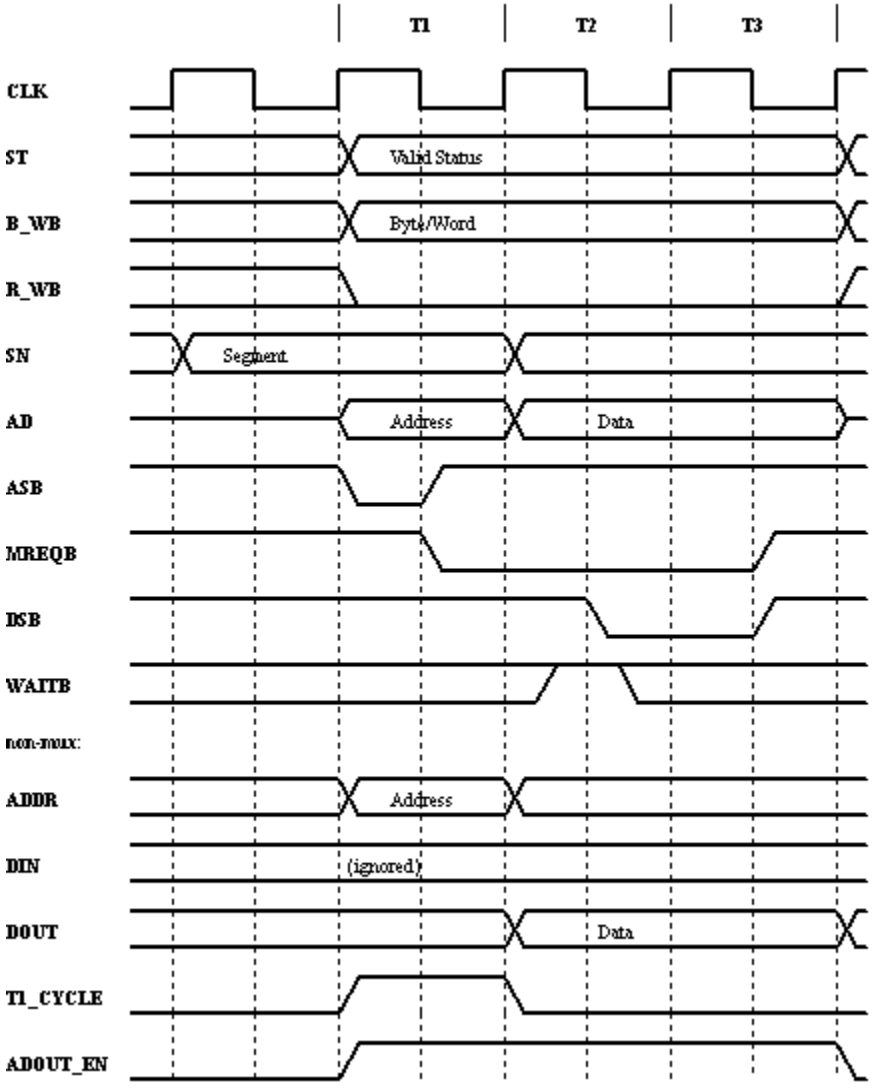
In the figures below individual clock cycles are labelled as T1, T2, etc. The actual number of clock cycles in any bus transaction (also called a machine cycle) may be longer than the typical “T3” shown in the figures.

The SN bus is only used in the Z8001 mode and should be ignored in the Z8002 mode. The SN signals are always valid one clock cycle before the ASB signal goes active to give an external MMU time to translate the segment number into a physical address. The SN signals are not always shown in all of the figures below.

The figure below shows a memory read bus transaction, without wait states:

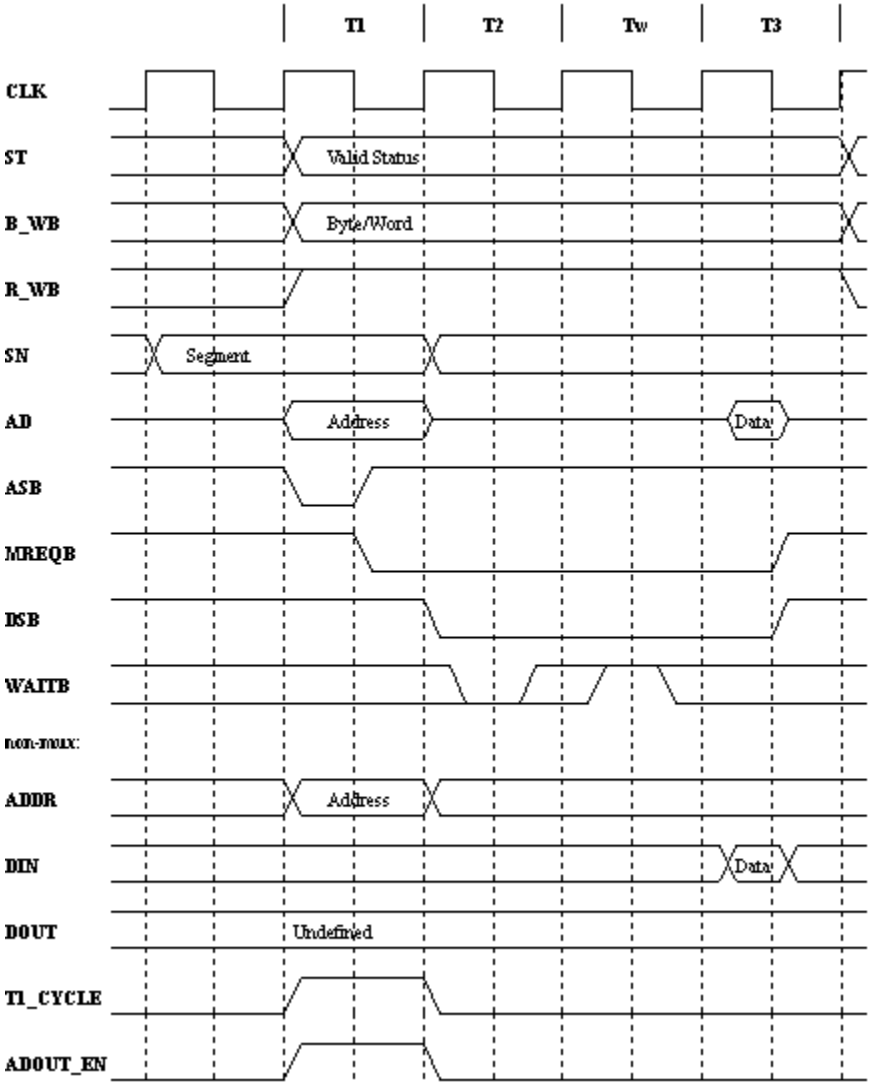


The figure below shows a memory write bus transaction, without wait states:

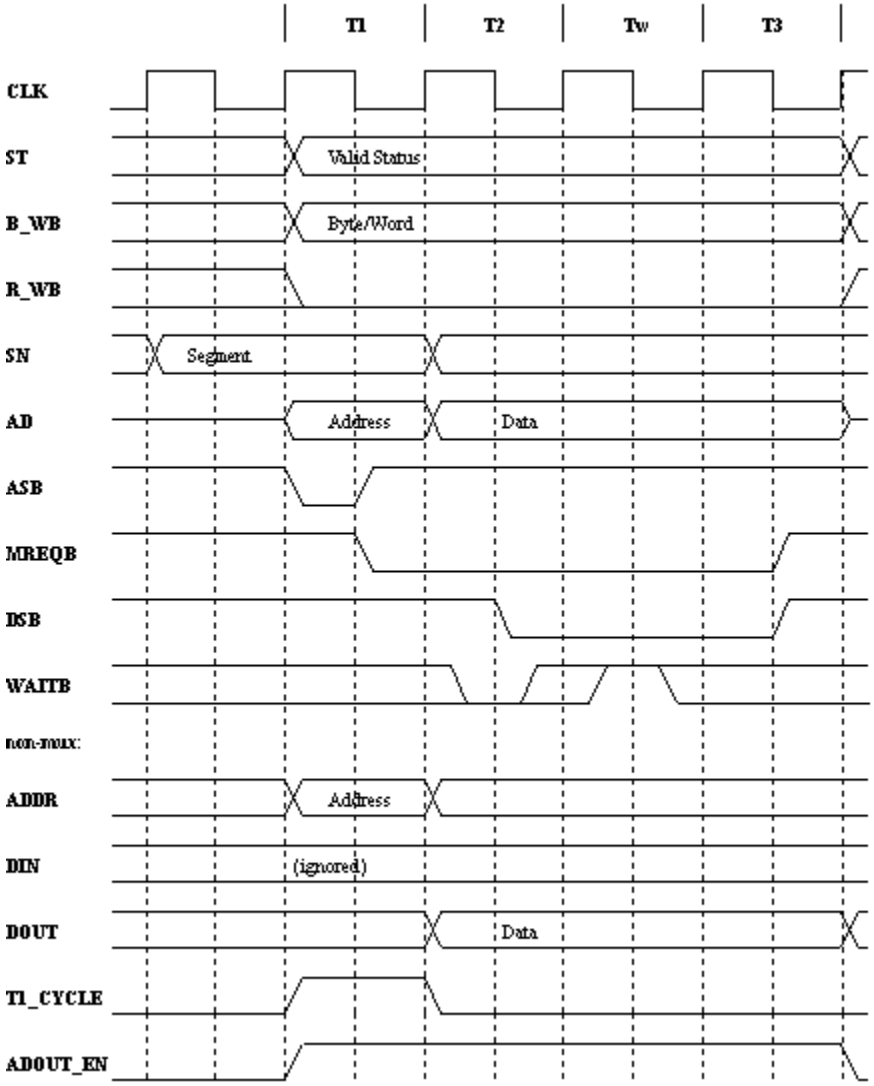


For memory write cycles the DSB signal is delayed by one-half clock cycle relative to the read case to guarantee that the data bus is valid prior to the leading edge of DSB.

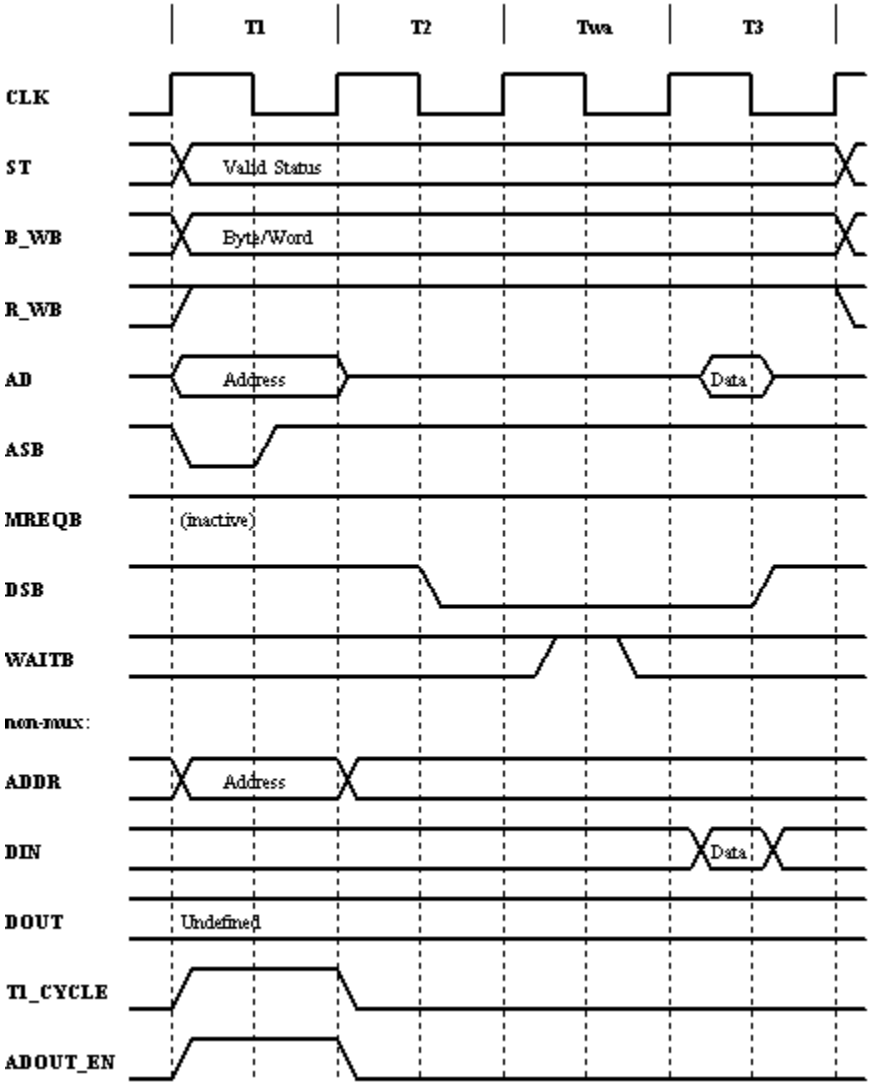
The figure below shows a memory read bus transaction, with one wait state:



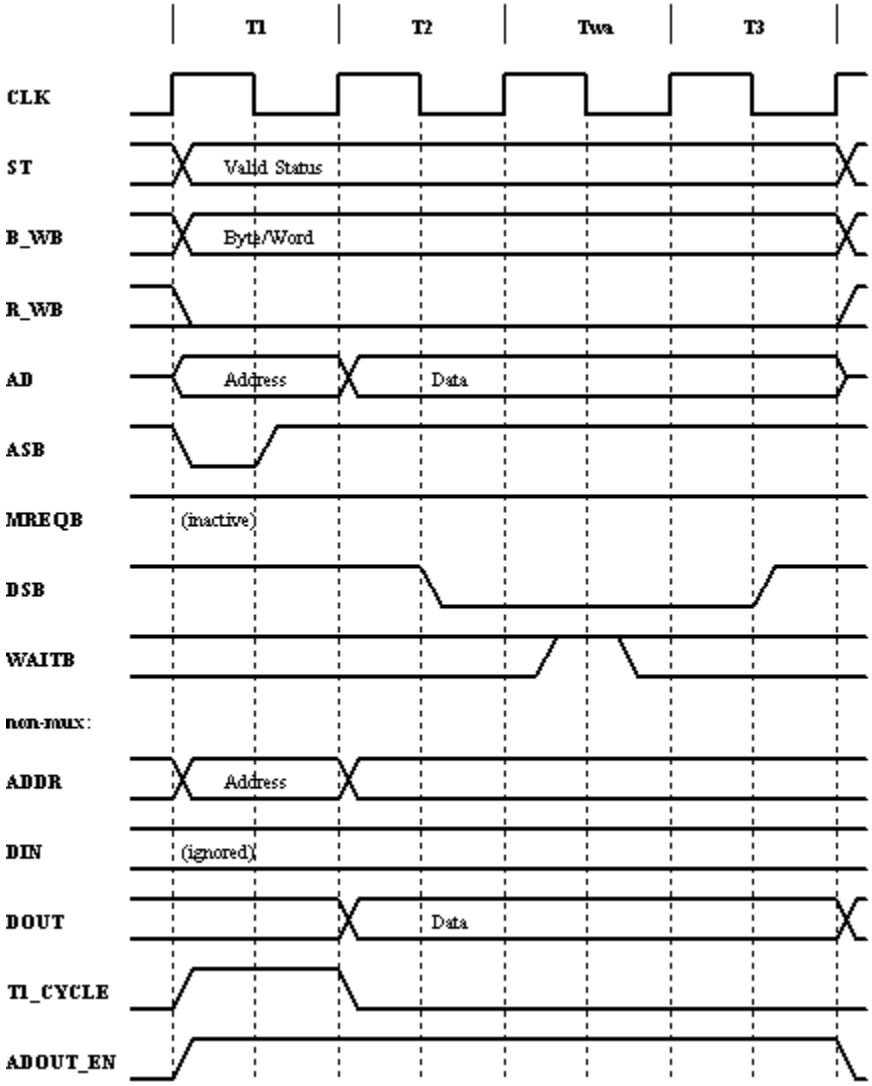
The figure below shows a memory write bus transaction, with one wait state:



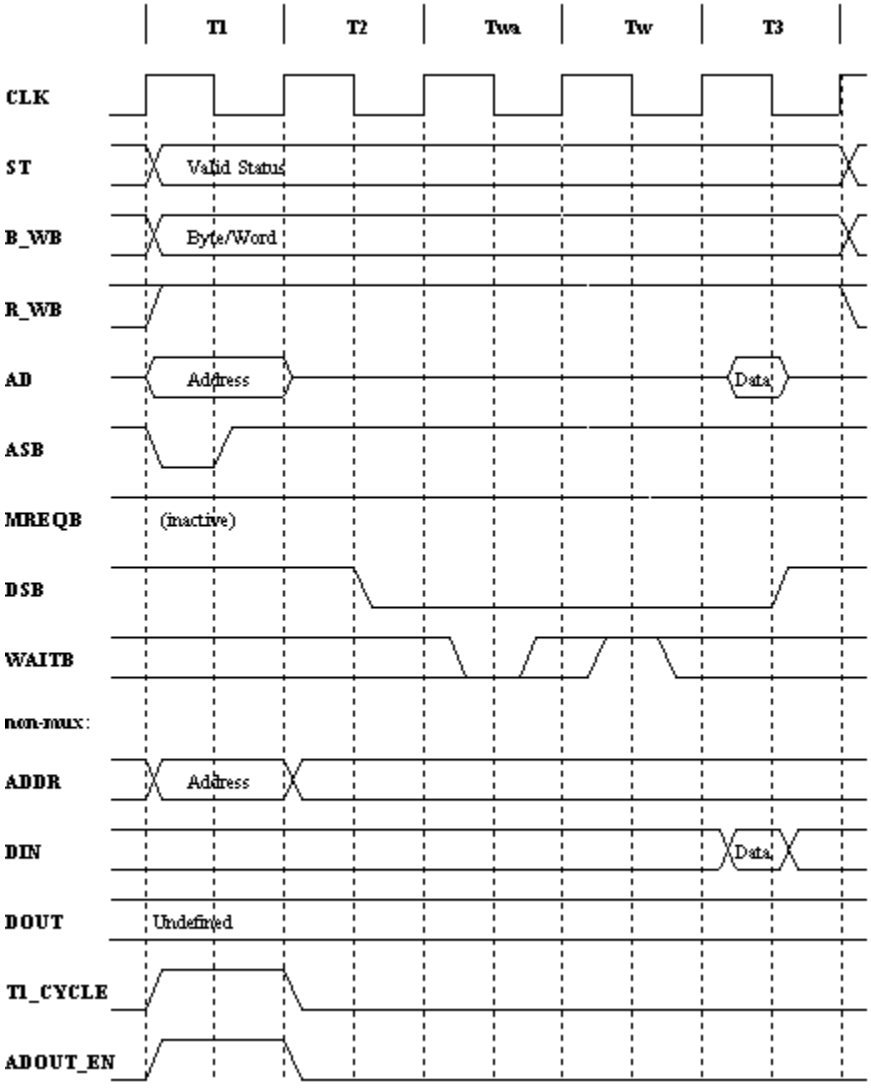
The figure below shows an I/O read bus transaction, with only the automatic wait state:



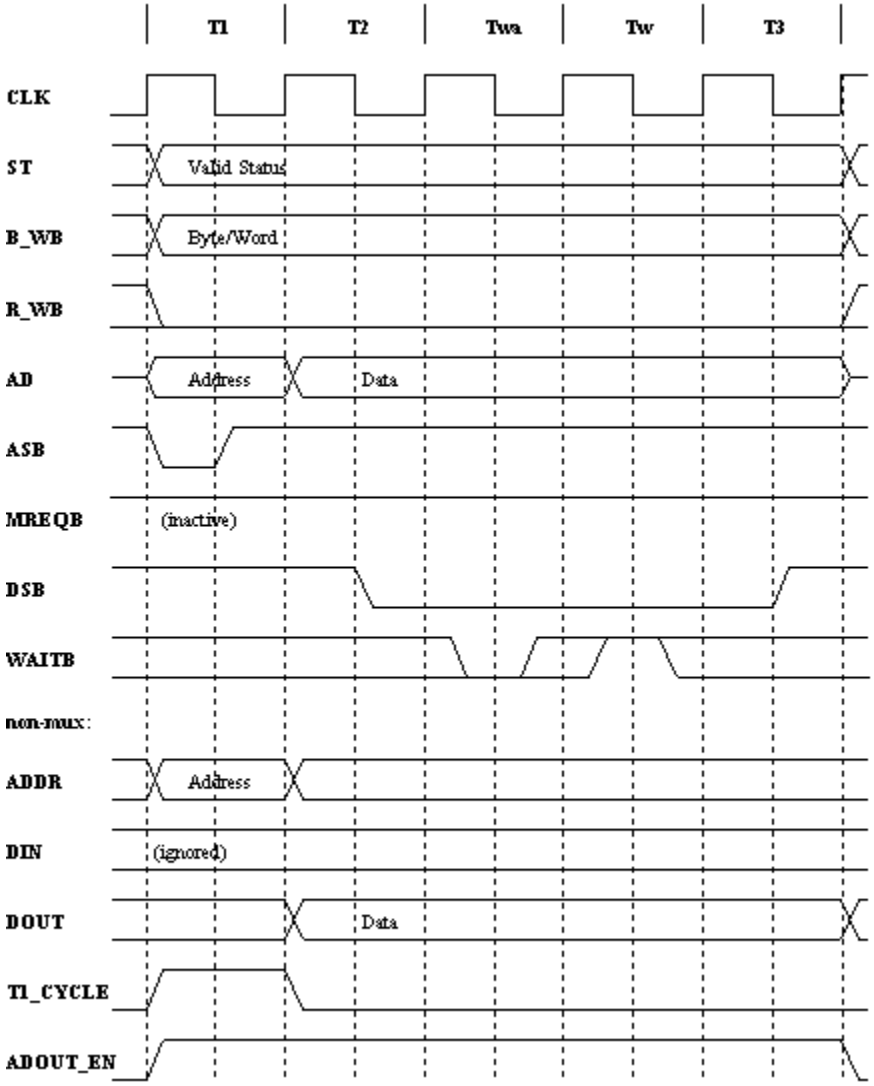
The figure below shows an I/O write bus transaction, with only the automatic wait state:



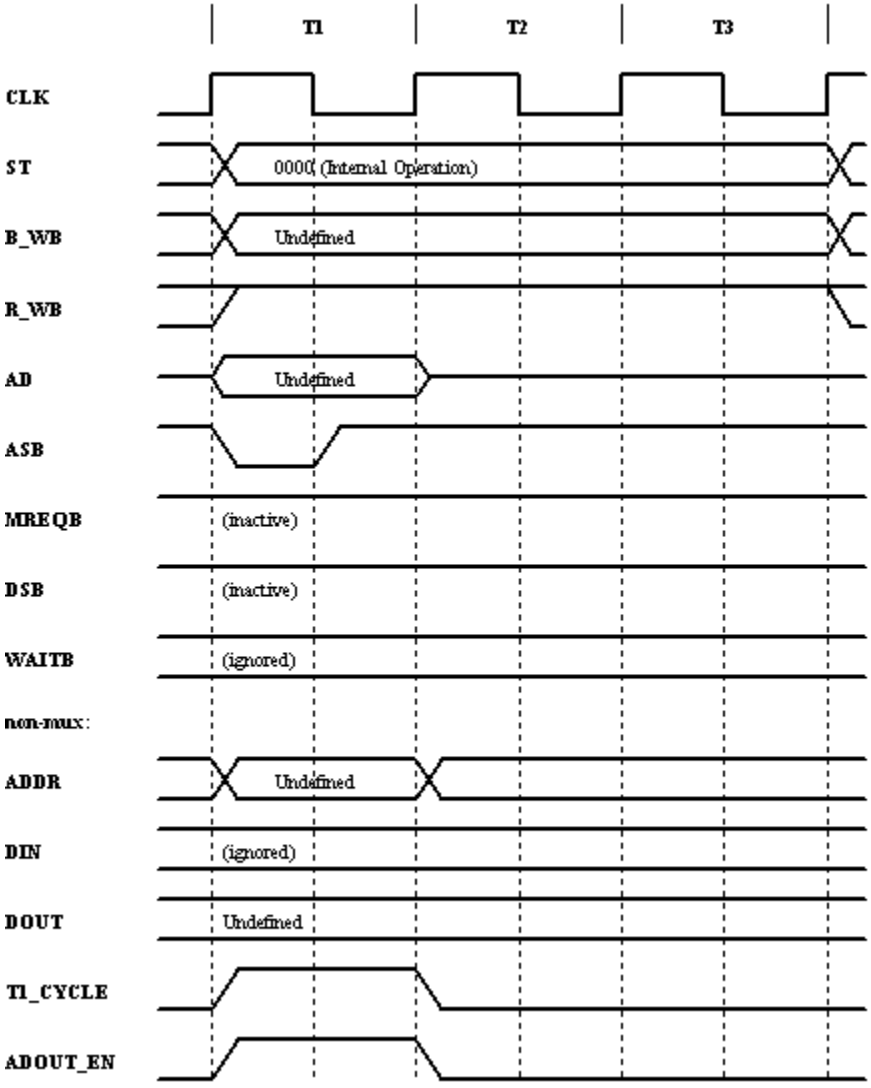
The figure below shows an I/O read bus transaction, with one additional wait state:



The figure below shows an I/O write bus transaction, with one additional wait state:

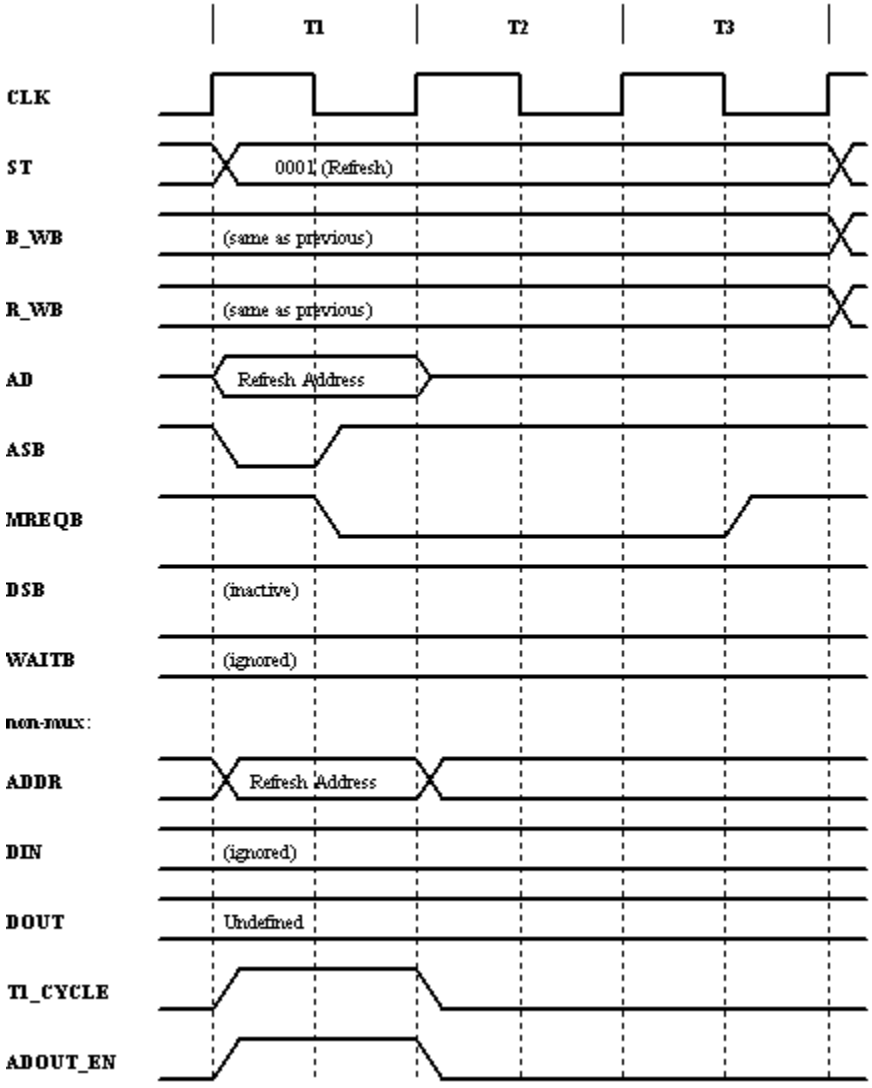


The figure below shows an internal operation bus transaction:



The Wait Request input is never sampled for this type of bus transaction.

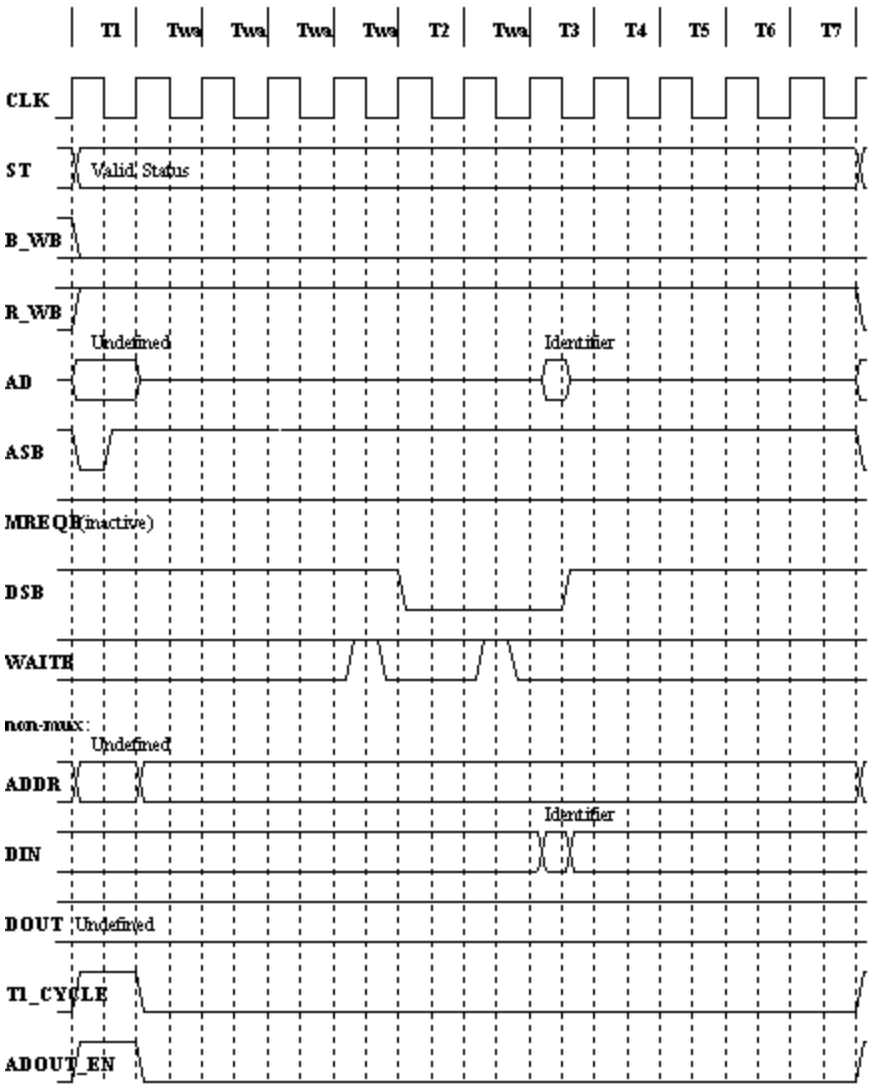
The figure below shows a Refresh bus transaction:



The Wait Request input is never sampled for this type of bus transaction. This type of bus transaction is also issued in response to a Stop request.

A Refresh transaction can be inserted immediately after any type of bus transaction. Normally only one Refresh transaction will be inserted, but the Y8000 accumulates up to eight missed Refresh timeouts, so more than one Refresh transaction is certainly possible. The Refresh address is incremented by two by each Refresh transaction.

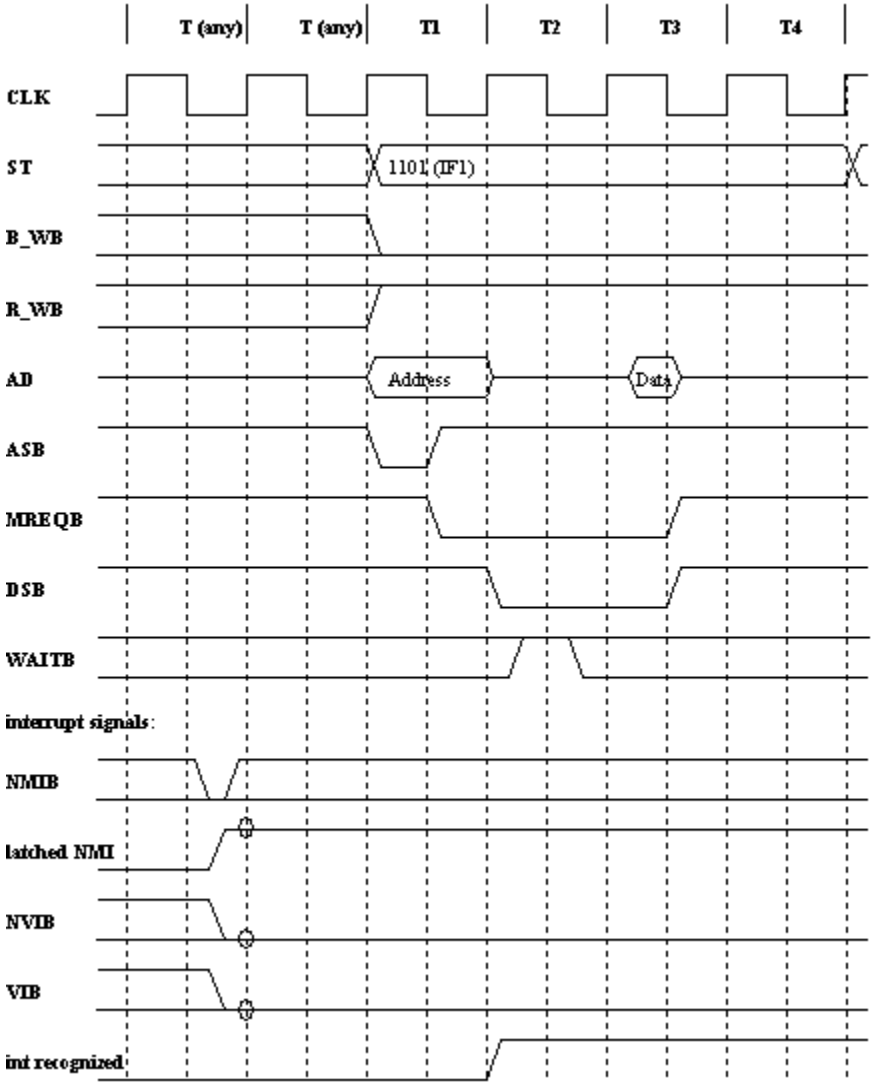
The figure below shows an interrupt acknowledge bus transaction (T1 through T3) and the four idle clock cycles (T4 through T7) that always follow an interrupt acknowledge:



The Wait Request input is sampled twice during an interrupt acknowledge, once before the Data Strobe is activated and once while Data Strobe is active. The first Wait Request sample allows for extra time to be inserted while the interrupt daisy chain (a Z-Bus feature) settles to resolve interrupt priority.

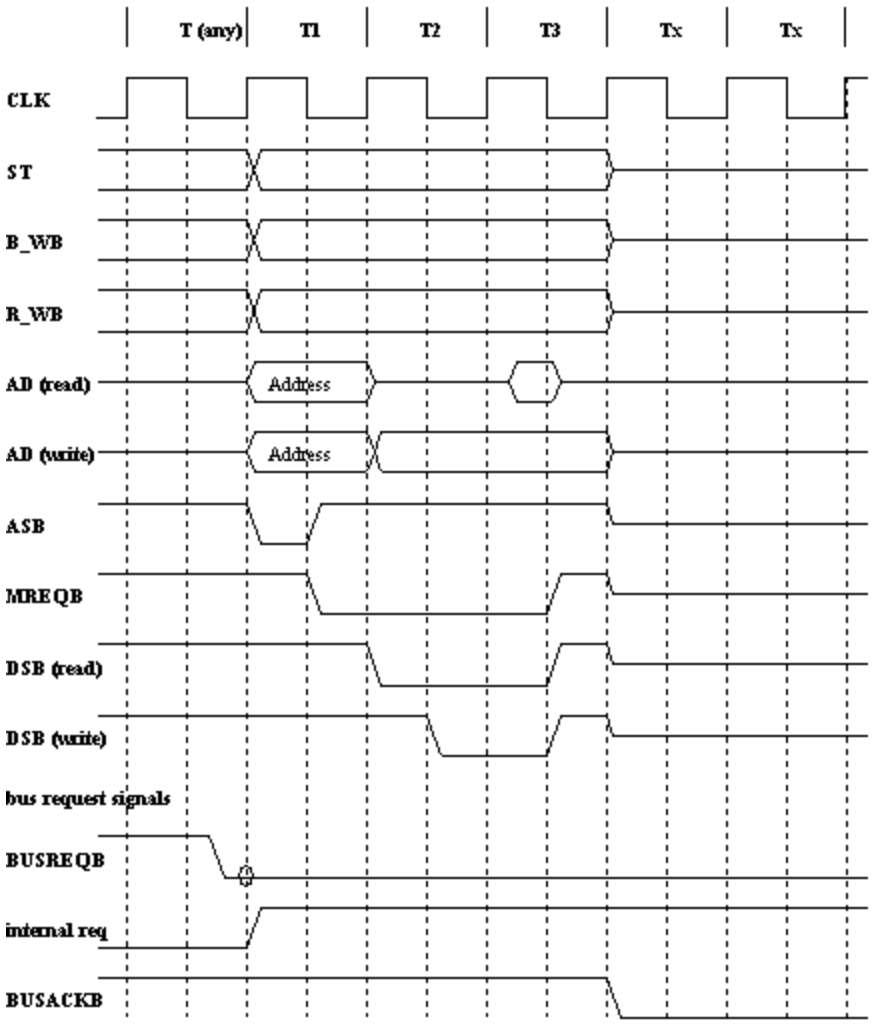
This sequence is followed immediately by the status-saving sequence shown in a later figure.

The figure below shows how interrupts are sampled, with the fastest possible response:



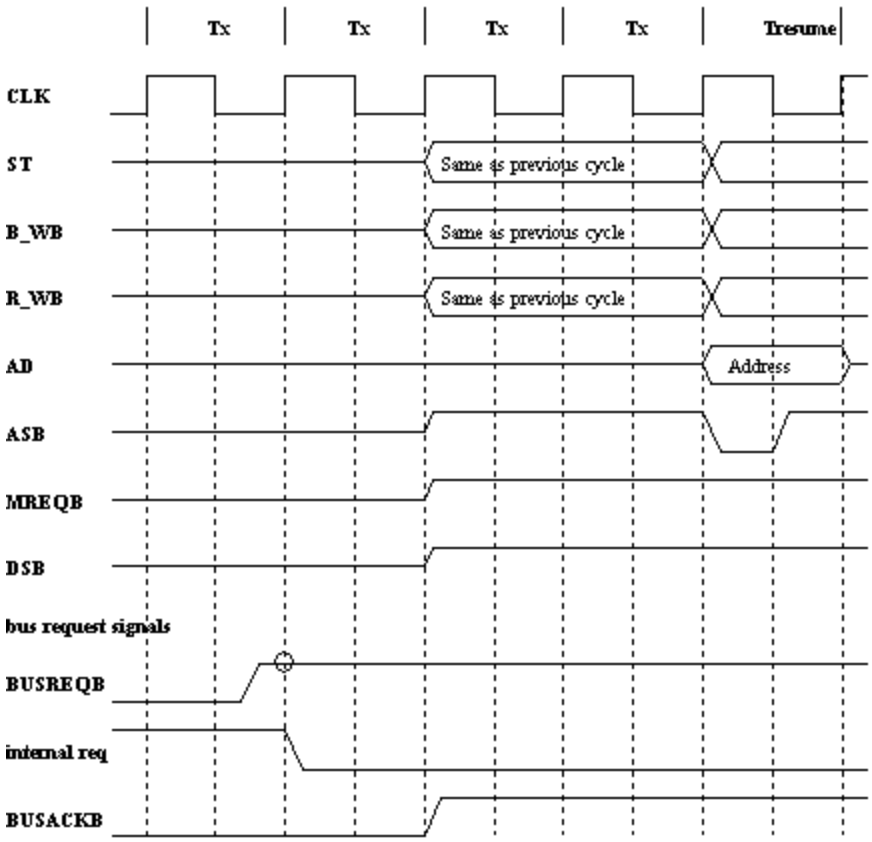
The Z8000 documentation indicates that this aborted IF1 transaction may be from 3 to 7 cycles (plus Wait states) long, but the aborted IF1 transaction is always 4 clock cycles (plus Wait states) long in the Y8000 design. An interrupt acknowledge transaction follows immediately after this aborted IF1 transaction. For some instructions (block instructions, for example) the interrupt sampling is not at the end of instruction execution.

The figure below shows the sampling of Bus Request and the entry into the Bus Acknowledge state:



Processor operation stops completely while Bus Acknowledge is active, even though it is theoretically possible for some internal operations to continue in parallel. This is believed to be consistent with the operation of the Z8000 processor.

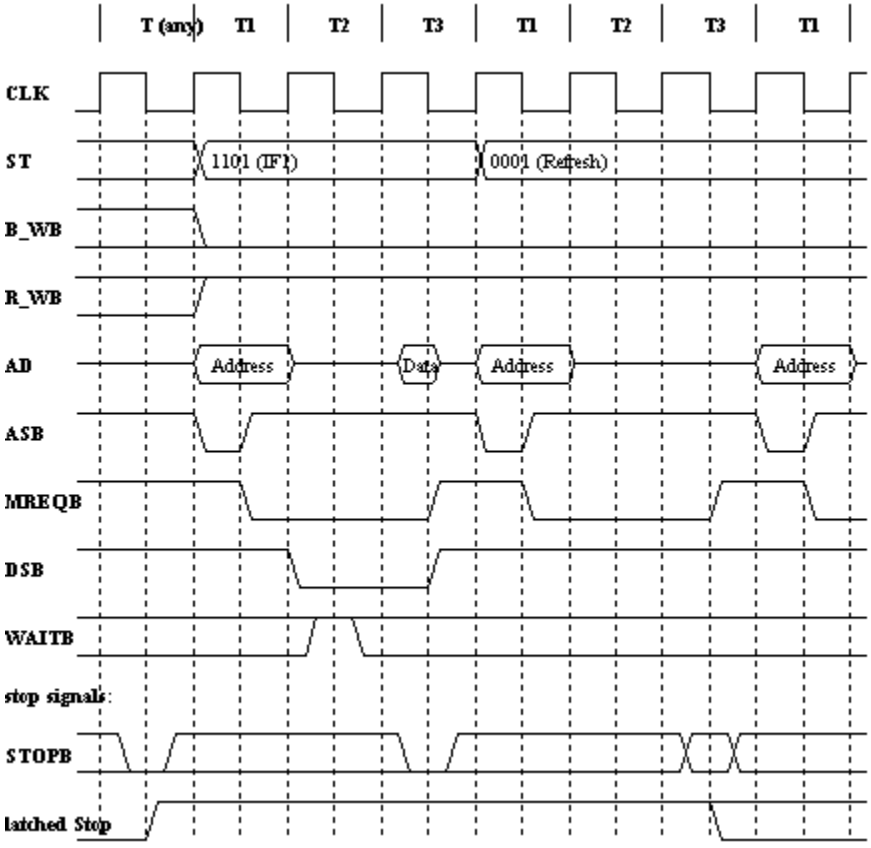
The figure below shows the sampling of Bus Request and the exit from the Bus Acknowledge state:



The first two clock cycles after exiting the Bus Acknowledge state are still part of the Bus Acknowledge sequence, and no processing takes place during these clock cycles. Processor operation resumes where it left off upon entering the Bus Acknowledge state. This may or may not be the start of another bus transaction.

The Bus Request input must remain inactive (High) for at least two clock cycles between successive bus requests. This is consistent with the operation of the Z8000 processor and is a consequence of the two clock cycles required for recovery at the end of a Bus Acknowledge sequence.

The figure below shows how the Stop Request is sampled: This figure shows two inserted Stop (Refresh) transactions. The Refresh address is used, and is incremented by two for every (Stop) (Refresh) transaction.



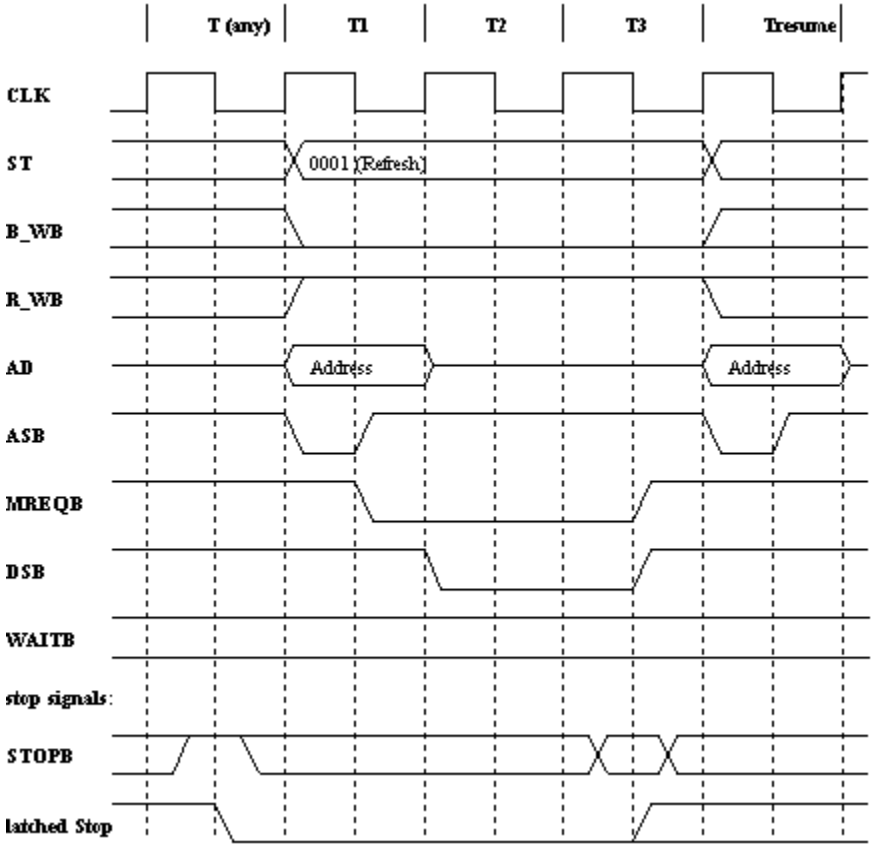
Unlike the internal Refresh request, the Stop Request input is only sampled at specific times.

The first case is just prior to an IF1 bus transaction. The IF1 bus transaction that follows the sampling of the Stop Request active is always exactly three clock cycles long. In this case any extra Instruction Fetch clock cycles follow the inserted Refresh bus transactions.

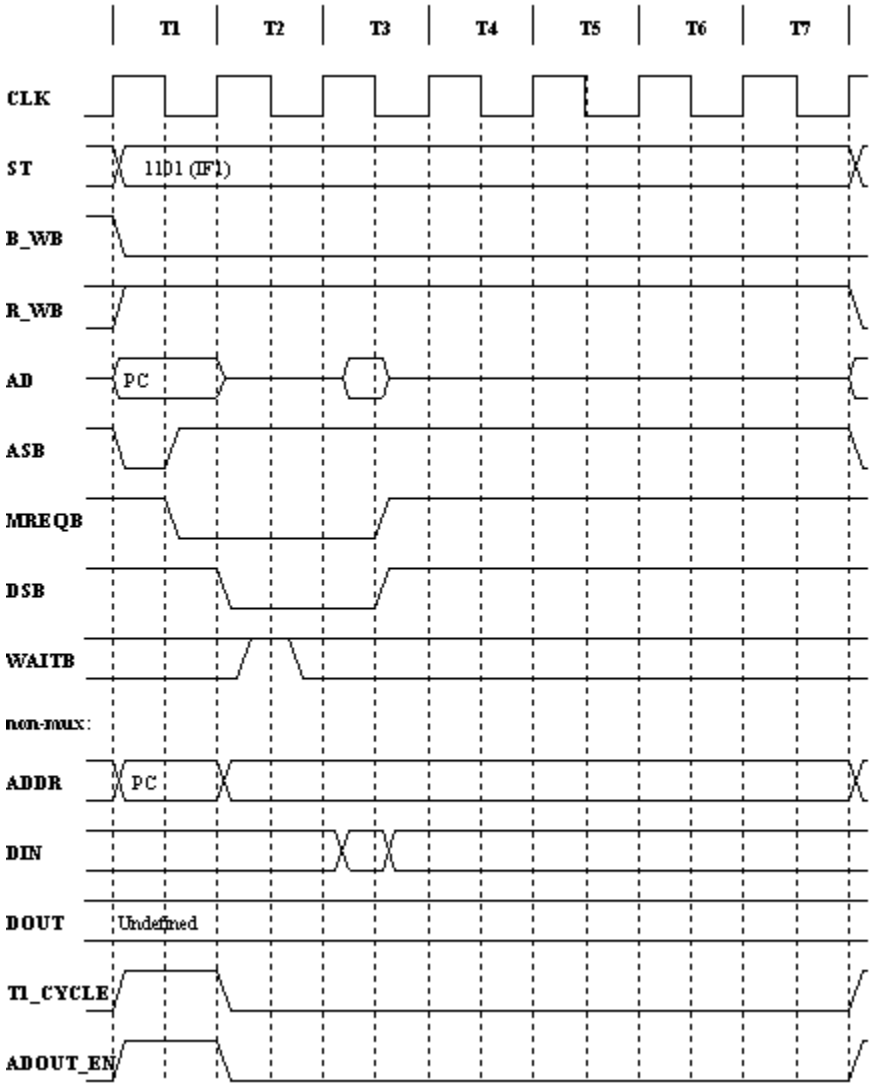
The second case is just prior to a Refresh transaction. In this case the three-clock Refresh transaction will complete, followed immediately by either another Refresh transaction or the resumption of the paused instruction.

The final case occurs only if the EPA bit is set and the instruction fetched during an IF1 transaction is an EPU instruction. In this case the Stop Request is sampled during the clock cycle immediately following the IF1 transaction. The second word of the EPU instruction will be fetched fetched, followed immediately by Refresh transactions if the Stop Request is active. This feature allows an EPU to pause CPU instruction fetches while the EPU catches up to the CPU.

The figure below shows the de-assertion of the Stop Request:

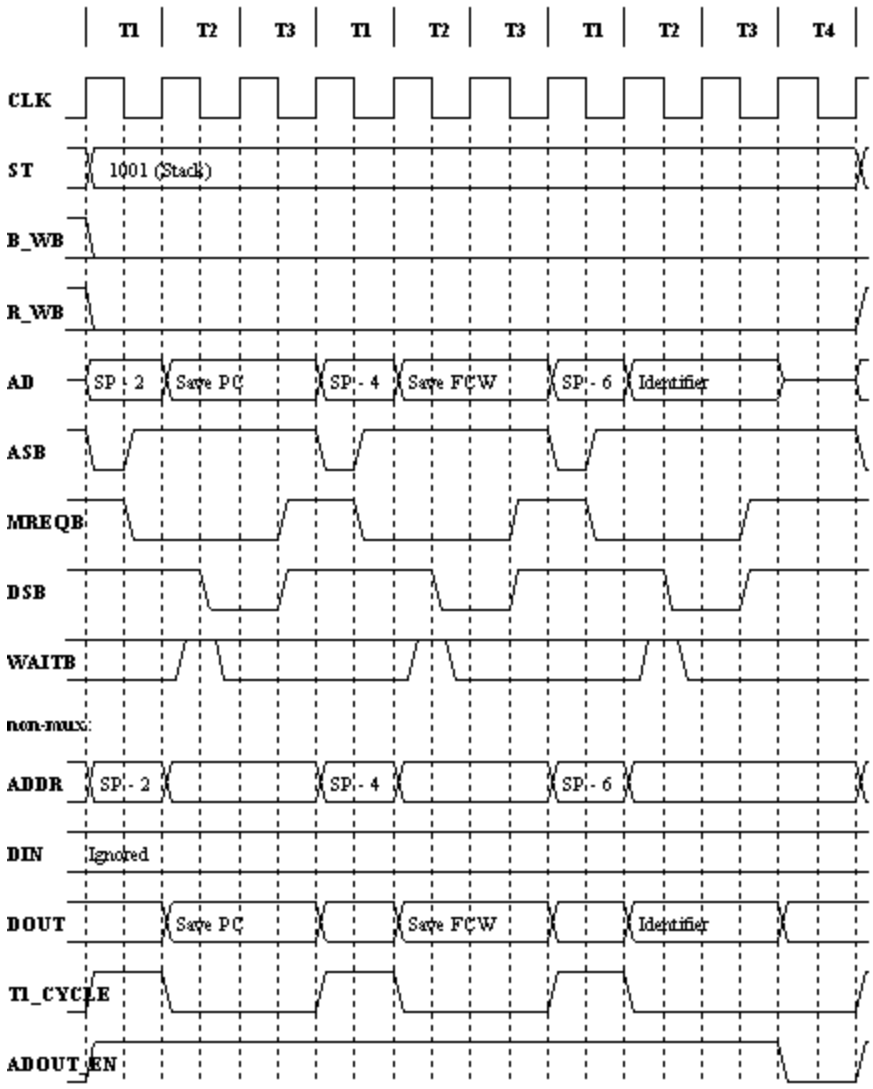


The figure below shows the timing for Traps, which are the System Call instruction, an EPU instruction when the EPA bit in the FCW is clear, or a Privileged Instruction:



This sequence is followed immediately by the status-saving sequence shown in the next two figures.

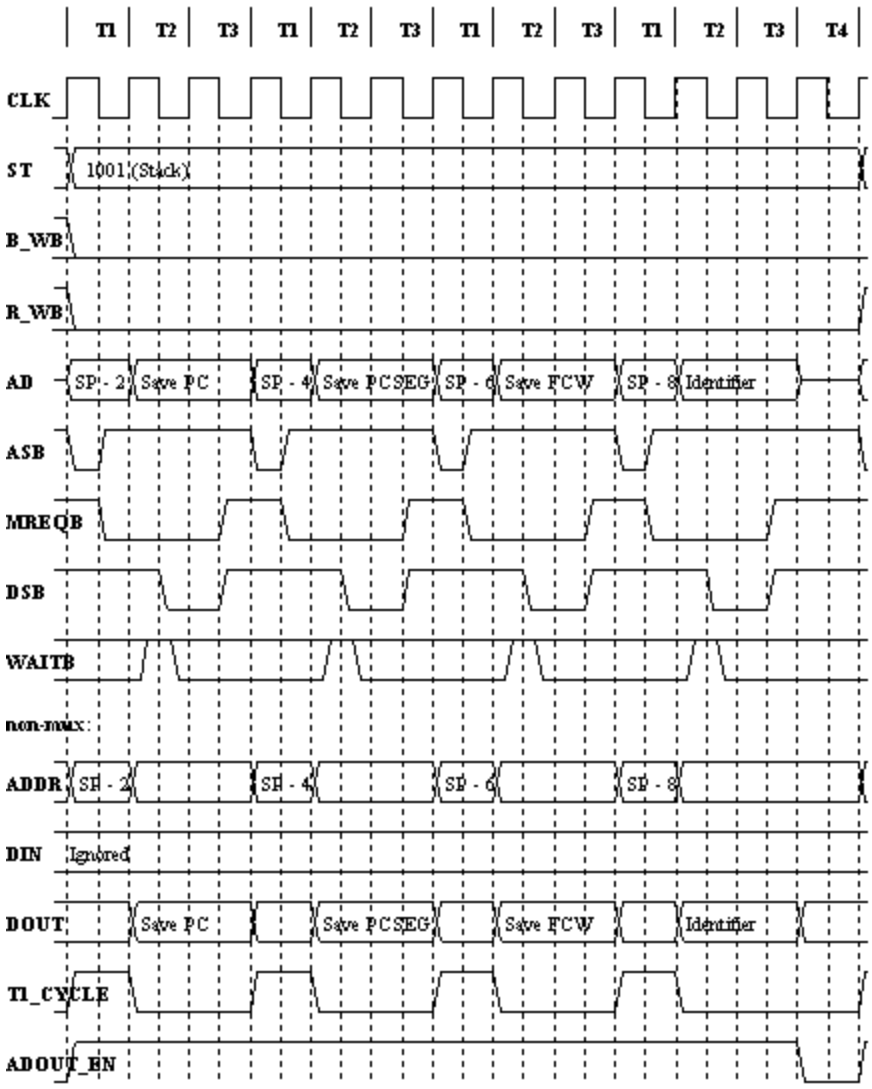
The figure below shows the status saving sequence common to both interrupts and traps when operating in Z8002 mode:



This status-saving sequence immediately follows the interrupt acknowledge sequence or trap sequence. The stack pointer used is the System SP (R15).

Although not shown in the figure, this sequence also includes one 3-clock-cycle Internal Operation bus transaction at the end of the sequence.

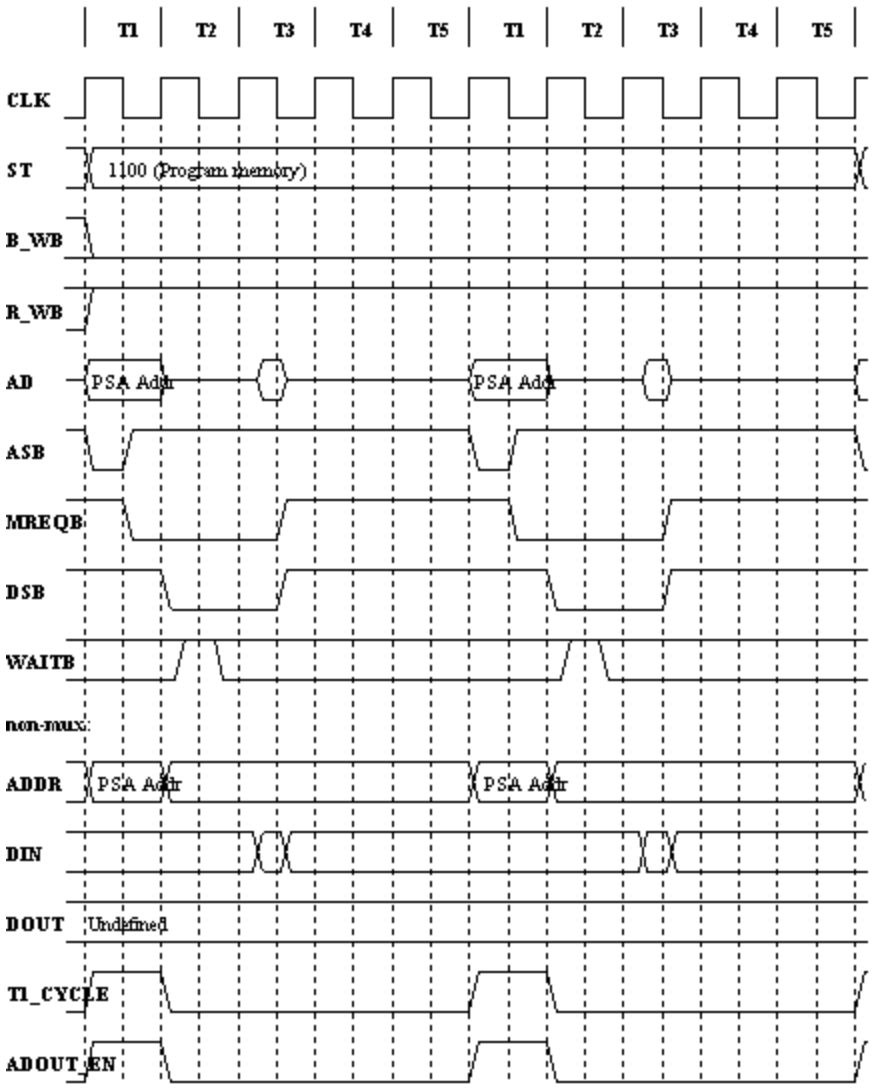
The figure below shows the status saving sequence common to both interrupts and traps when operating in Z8001 mode:



This status-saving sequence immediately follows the interrupt acknowledge sequence or trap sequence. The stack pointer used is the System SP (RR14).

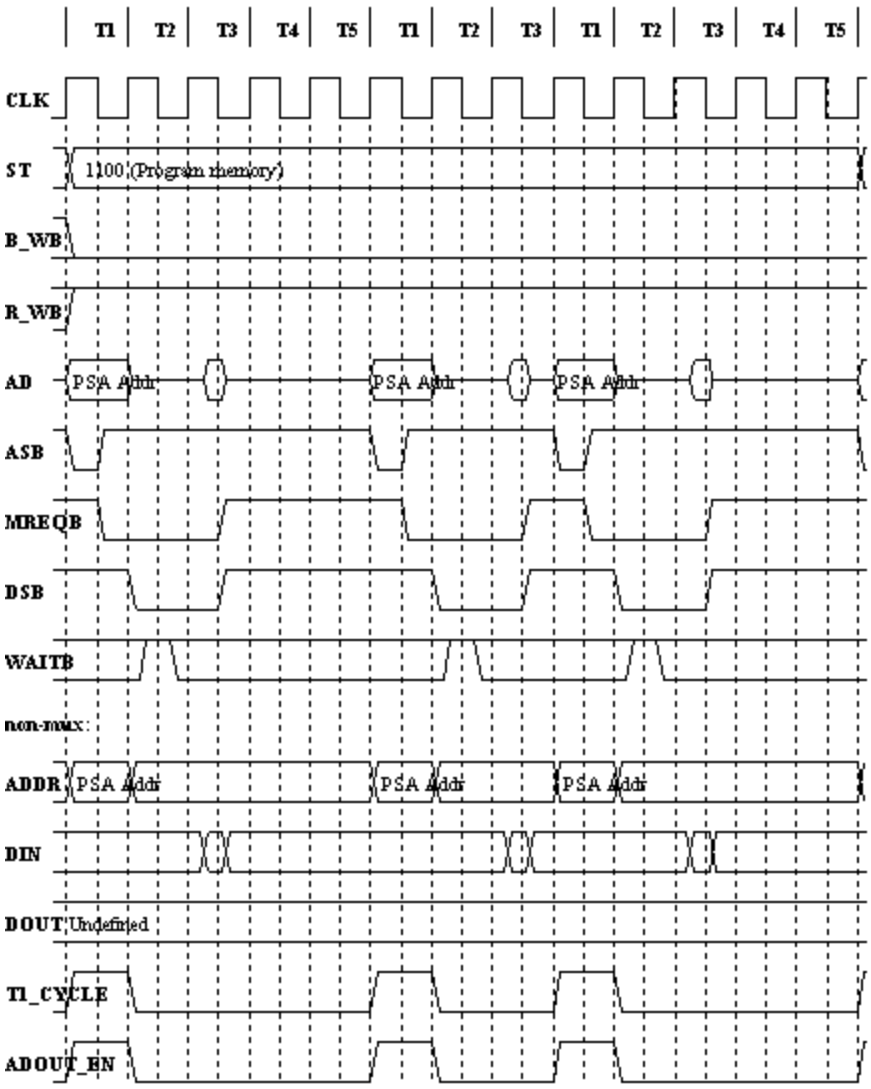
Although not shown in the figure, this sequence also includes one 3-clock-cycle Internal Operation bus transaction at the end of the sequence.

The figure below shows the Program Status Area fetch sequence common to reset, traps and interrupts in Z8002 mode:



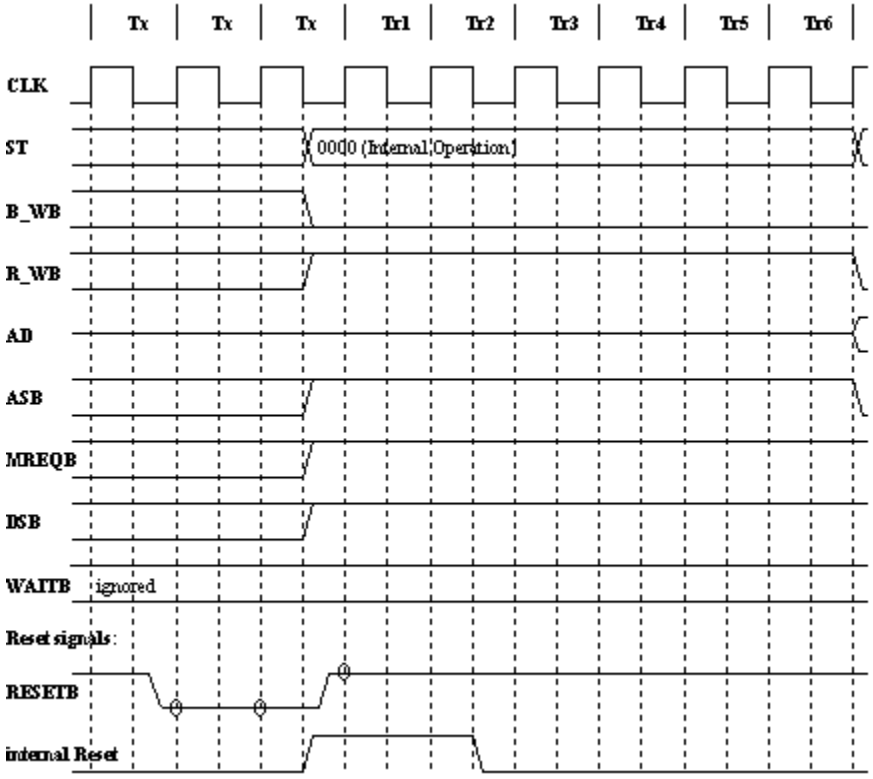
This sequence follows the reset sequence directly. This sequence follows the trap or interrupt status-saving sequence after three clock cycles of no external operation.

The figure below shows the Program Status Area fetch sequence common to reset, traps and interrupts in Z8001 mode:



This sequence follows the reset sequence directly. This sequence follows the trap or interrupt status-saving sequence after three clock cycles of no external operation.

The figure below shows the Reset timing and reset sequence:



The Y8000 design requires that the Reset Request input be sampled active by two successive rising edges of the System Clock to be recognized. This is believed to be consistent with the operation of the Z8000 processor.

This page is intentionally left blank

Interrupts and Traps

The Z8000 architecture supports three different types of interrupts and four different types of trap. Each interrupt and trap type will be discussed below.

The response to an interrupt or trap is nearly identical, the difference being the interrupt acknowledge bus cycle generated in response to an interrupt. The interrupt acknowledge cycle samples the data bus and the sampled data is used as an identifier for the interrupt. A trap uses the first word of the instruction that causes the trap as the identifier.

After the interrupt acknowledge cycle (in the case of an interrupt), or the fetch of the offending instruction (in the case of a trap) the processor pushes the Program Counter, followed by the FCW, followed by an identifier, onto the system stack. The Program Counter is two words (PCS and PC) in Z8001 mode and one word (PC only) in Z8002 mode. The new FCW and Program Counter are then fetched from the Program Status Area in program memory to start the service routine. At the end of the service routine an IRET instruction is used to restore the operating state of the processor. The Program Counter value pushed in various cases is shown below:

Exception	Program Counter value pushed on stack
EPU instruction trap	Address of second word of EPU instruction (all EPU instructions are two words)
Privileged Instruction Trap	Address of second word of privileged instruction (all privileged instructions are two words).
System Call	Address of the word following the SC instruction.
Any Interrupt	Address of next instruction (or current instruction if a block instruction that has not completed is interrupted)

The start of the Program Status Area is pointed to by the PSAP register. This register is cleared by a Reset and should be written with a different value using a LDCTL instruction before the Program Status Area is accessed. The Program Status Area starts on a 256-byte boundary and is organized as shown in the tables below:

The Program Status Area should be relocated away from its reset location because the Reset FCW and PC are fetched from locations that interfere with the Program Status Area entry for the Extended Instruction Trap when using the first 256-byte page as the Program Status Area. The format of the Program Status Area is different for Z8001 mode and Z8002 mode.

Non-segmented (z8002 mode) Program Status Area

PSAP Offset	Contents	Interrupt/Trap type
0x000 0x002		Reserved
0x004 0x006	FCW PC	Extended Instruction Trap
0x008 0x00A	FCW PC	Privileged Instruction Trap
0x00C 0x00E	FCW PC	SC Trap
0x010 0x012		Unused - available for custom use
0x014 0x016	FCW PC	Non-maskable Interrupt
0x018 0x01A	FCW PC	Non-vectored Interrupt
0x01C	FCW	Vectored Interrupts
0x01E	PC (vector = 0x00)	
0x020	PC (vector = 0x01)	
0x022	PC (vector = 0x02)	
.	.	
.	.	
.	.	
0x21A	PC (vector = 0xFE)	
0x21C	PC (vector = 0xFF)	

The Z8002 PSAP offset 0x010-0x012 was originally reserved for use with a bank-switched MMU but this feature was never implemented.

Segmented (z8001 mode) Program Status Area

PSAP Offset	Contents	Interrupt/Trap type
0x000		Reserved
0x002		
0x004		
0x006		
0x008	Reserved	Extended Instruction Trap
0x00A	FCW	
0x00C	PC Segment	
0x00E	PC	
0x010	Reserved	Privileged Instruction Trap
0x012	FCW	
0x014	PC Segment	
0x016	PC	
0x018	Reserved	SC Trap
0x01A	FCW	
0x01C	PC Segment	
0x01E	PC	
0x020	Reserved	Segment Trap
0x022	FCW	
0x024	PC Segment	
0x026	PC	
0x028	Reserved	Non-maskable Interrupt
0x02A	FCW	
0x02C	PC Segment	
0x02E	PC	
0x030	Reserved	Non-vectored Interrupt
0x032	FCW	
0x034	PC Segment	
0x036	PC	
0x038	FCW	Vectored Interrupts
0x03C	PC Segment	
0x03E	PC (vector = 0x00)	
0x040	PC Segment	
0x042	PC (vector = 0x02)	
.	.	
.	.	
.	.	
0x234	PC Segment	
0x236	PC (vector = 0xFC)	
0x238	PC Segment	
0x23A	PC (vector = 0xFE)	

An Extended Instruction Trap occurs when the processor fetches an Extended Instruction while the EPA bit in the FCW is cleared, indicating that no EPU is present in the system. This allows the processor to emulate the operation of an Extended Instruction in software.

The Privileged Instruction Trap occurs when the processor fetches a privileged instruction while the SYS bit in the FCW is cleared, indicating the processor is operating in Normal mode. This prevents Normal mode programs from accessing system resources such as I/O.

The SC Instruction Trap occurs when the CPU fetches an SC instruction, which is usually used to request some type of operating system service.

The Non-maskable Interrupt is triggered by a falling edge on the Non-maskable Interrupt Request input. This interrupt request is edge-triggered and cannot be disabled by software.

The Non-vectored Interrupt is taken when the Non-vectored Interrupt Request input is asserted and the NVIE bit in the FCW is set. Clearing the NVIE bit in the FCW disables non-vectored interrupts. The Non-vectored Interrupt Request input must remain asserted until the corresponding interrupt acknowledge bus cycle is issued by the processor. This interrupt is called non-vectored because even though an identifier is read during the interrupt acknowledge cycle, it is not used by the processor and is merely pushed onto the system stack for use by software.

The Vectored Interrupt is taken when the Vectored Interrupt Request input is asserted and the VIE bit in the FCW is set. Clearing the VIE bit in the FCW disables vectored interrupts. The Vectored Interrupt Request input must also remain asserted until asserted until acknowledged by the processor. This interrupt is called vectored because the processor uses the data returned from the interrupting device during the interrupt acknowledge cycle (the “vector”) to select from a number of new Program Counter values stored in the Program Status Area. Only the lower eight bits of the vector are used for this purpose, selecting from among 256 (Z8002 mode) or 128 (Z8001 mode) different Program Counter values. All vectored interrupts share the same FCW value in the Program Status Area.

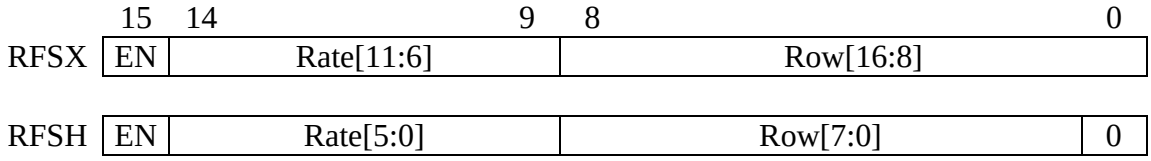
In the case of a simultaneous trap and interrupt or multiple interrupt requests the following prioritization applies. Once the highest priority request has been answered, the new FCW and Program Couinter applies and the remaining pending interrupt requests are accepted as appropriate.

Priority	Interrupt or trap type
Highest	Traps. Only one type of trap can occur at a time.
	Non-maskable Interrupt
	Vectored Interrupt
Lowest	Non-vectored Interrupt

This page is intentionally left blank

Refresh

The 16-bit Z8000 architecture includes hardware to automatically refresh dynamic memory, and this hardware supports the DRAM organizations that were state-of-the-art at the time that the Z8000 family was available. The Y8000 includes this refresh logic, but expands the refresh capabilities to include current DRAM organizations, using a new control register.



Refresh Control

The original Refresh Control (RFSH) register contains an 8-bit row counter, a 6-bit rate counter, and an enable/disable control bit. Bit 15 of this register (the Enable bit) is cleared by reset but the other bits in the register were unaffected by reset. The Y8000 RFSH control register is identical to the original Z8000 register except that all bits of the register are cleared by reset.

The Y8000 adds an Extended Refresh register (RFSX) that adds nine bits to the row counter and six bits to the rate counter, along with a separate enable bit for the expanded features. The entire RFSX register is cleared by reset and can only be accessed by a Y8000-specific LDCTLL instruction.

The Rate value controls the time between refresh transactions. Rate values between 0 and 7 select a fixed time of 256 clocks between refresh requests, while a value of "n" enables a refresh period of "4n" clocks. If the Extended Enable bit is cleared this provides a range of 32 clocks to 252 clocks between each refresh request. If the Extended Enable bit is set a range of 32 clocks to 16,380 clocks between refresh requests is available.

Refresh requests may not result in immediate refresh trasnsactions on the bus if, for example, the CPU does not control the bus. So the Y8000 accumulates up to eight missed refresh requests and will perform the required refresh transactions in a burst as soon as possible. The original Z8000 only accumulated two missed refresh requests.

The seventeen bits of row address are output during a refresh transaction and increment with every refresh transaction. If the Extended Enable bit is clear this increment is only across the eight bits of row address in the RFSH register. The increment carries to the upper bits of row address in the RFSX register only if the Extended Enable bit is set.

Because the Z8000 memory is sixteen bits wide the row address is offset on the AD bus by one bit and AD[0] is always zero during refresh transactions. So Row[14:0] are driven on AD[15:1] and Row[16:15] are driven on SN[1:0]. The remaining SN signals are always Low during refresh transactions. During refresh transactions the SN signals are not driven one clock cycle early, but are driven with the same timing as the AD lines.

Reset

The Device Reset (RESETF) input forces the Y8000 CPU into a known state, irrespective of its current state. This input must be sampled active for at least two successive rising edges of the clock input to be properly recognized, and then on the next falling edge of the clock all processor outputs assume known states. This condition continues as long as the Device Reset input remains active.

The reset state of all processor outputs is shown in the table below:

Name	Description	Reset state
AD[15:0]	Address/Data Bus	3-state
ADDR[15:0]	Address Bus	Unaffected by reset.
ADOUT_EN	Address Output Enable	Low (inactive).
ASB	Address Strobe	High (inactive).
B_WB	Byte/Word	Low (signalling Word).
BUSACKB	Bus Acknowledge	High (inactive).
DOUT	Data Output	Unaffected by reset.
DSB	Data Strobe	High (inactive).
MOB	Multi-micro Out	High (inactive).
MREQB	Memory Request	High (inactive).
N_SB	Normal/System	Low (signalling System).
R_WB	Read/Write	High (signalling Read).
SN[7:0]	Segment Number	All zero.
ST[3:0]	Status	All zero (signalling IOP).
T1_CYCLE	T1 Cycle Identifier	Low (inactive).

Once the Device Reset input is sampled inactive the processor fetches an FCW, PC Segment (in the case of Z8001 mode) and PC value from memory. All of these fetches are from the program memory address space and the processor is in System mode for these fetches. Execution then begins at the address of the fetched PC with the status from the fetched FCW.

The addresses for these initial fetches are shown in the table below:

Address	!z8002	z8002
0x0000	not used	not used
0x0002	initial FCW	initial FCW
0x0004	initial PC segment	initial PC
0x0006	initial PC	not used

The Device Reset does not affect any of the registers in the register file. In fact the only register that is initialized during the reset state is the Program Status Area Pointer, which is cleared to all zeros. The PSAP should be reprogrammed to point elsewhere in memory to avoid a conflict between the reset locations and the Program Status Area entry for the Extended Instruction trap.

Specifying Normal mode in the initial FCW is **not a good idea**. This is because the System Stack Pointer is not initialized by reset, and the System SP is what is used for interrupts and traps. Startup code should execute in System mode and at the very least initialize the System SP and the PSAP before switching to Normal mode.

If the Bus Request input is active upon exiting the reset state the fetch of the FCW will occur before the bus is released by the processor. This is because the Bus Request input is sampled at the start of a bus transaction.

If the Stop Request input is active upon exiting the reset state both the FCW and PC will be fetched, as will the first word of the first instruction, before any stop transactions are inserted. Again, this is consistent with the way that the Stop Request input is sampled at the start of an IF1 bus transaction.

Enhancements

The Y8000 design includes a number of backwards-compatible enhancements that are listed here for convenience. The first seven are completely transparent, meaning that they will only be visible if they are explicitly used.

1. SN[7] output

3. EPA output

3. Instruction: LDL @Rd, #data

4. Instruction: PUSHL @Rd, #data

5. Instruction: LDCTLL dst, RRs with a destination of NSP, PSAP or REFRESH

6. Instruction: LDCTLL RRd, src with a source of NSP, PSAP or REFRESH

7. Extended Refresh register, for a 17-bit refresh row counter and 10-bit rate counter.

The remaining three enhancements will be visible under certain circumstances.The refresh enhancements are easy to remove if complete compatability is required.

8. Improved execution time for Multiply and Divide instructions.

9. Accumulator for up to eight missed refresh transactions. The Z8000 accumulates two missed refresh transactions.

10. Minimum refresh rate of thirty-two clocks. The Z8000 has a minimum refresh rate of just four clocks.

This page is intentionally left blank

Verilog HDL Source

This chapter presents an overview of the Verilog HDL source code for the Y8000 design, sufficient to understand the basic organization of the design. Also described are the options available for the design. No attempt will be made to describe the internal operation of the design in detail. Refer to the comments embedded in the source code for that type of information. The test bench, described in a separate chapter, is covered in much more detail because users may actually find it useful to modify the test bench.

Each file contains just one Verilog HDL module, and may include (using the `` `include `` directive) one or more other files. In all cases the name of the file is identical to the name of the Verilog HDL module that it contains. The only files that do not contain a Verilog HDL module are *defines.v*, which contains all of the `` `define `` statements for the design in one central location, and *version.v*, which selects the options for the design.

The logical organization of the files (and Verilog HDL modules) is in the table below.

logical organization
Y8000.v
version.v
core8001.v
defines.v
ext_int.v
machine.v
control.v
datapath.v
reg_word.v
reg_file.v
ff_byte.v
ff_word.v
refresh.v

Before discussing individual Verilog HDL modules, it is appropriate to describe the versions of the design that are available. These versions are controlled by `` `define `` statements in the file *version.v*. This file is referenced by an `` `include `` statement in the top level of the design.

The first `define selects between an ASIC option and an FPGA option. The primary difference between these two options is that the FPGA option does not include any 3-state function for those outputs that are normally able to go high-impedance. If this functionality is required it can be implemented outside of the design. This option is available because synthesis tools sometimes have difficulty implementing 3-state signals.

The second `define selects between the normal multiplexed option and a special non-multiplexed option. The multiplexed option employs the normal bidirectional Address/Data bus familiar to Z8000 users, while the non-multiplexed option uses separate Address, Data In and Data Out busses. The multiplexed option obviously employs 3-state drivers, so selecting this option in conjunction with the FPGA option doesn't make a lot of sense.

The top level design file (and module) is called *Y8000.v*. This file is the top level "wrapper" for the design and is where the design options are implemented. It is also where the user can implement technology-specific I/O cells or buffers.

Several heavily-loaded signals from the core design are buffered here using instantiated technology-specific cells in the FPGA option. These cells can obviously be replaced with different buffers for different FPGA vendors. Technology-specific buffers for these heavily-loaded signals can also be instantiated in the ASIC option. This level is also where the user can insert I/O cells if desired.

The *core8001.v* file (and module) contains essentially all of the design, but no logic is implemented at this level. Rather, all of the remaining design files are referenced here, using `include statements, and the four main modules are instantiated.

The heavily-loaded signals exit this module and then are input to the module with a slightly different name to allow for technology-specific buffers to be implemented. These signals are the synchronized reset signal (`resetpls_reg` output and `resetb` input), the instruction register (`inst_reg` outputs and `buf_inst_reg` inputs), and the signal which stops all of the state machines for Bus Request, Wait Request and Stop Request (`hold_mach` output and `buf_hold_mach` input). The clock input to this module is called `buf_CLK`, having been buffered outside of the module.

Nearly all of the logic in the design is clocked by the rising edge of this clock. The only exceptions are in the external interface where certain falling-edge operation are required for compatibility with the Z8000.

The file *defines.v* contains all of the encoding for multi-bit control fields and state machines. Most of these control field encodings can be changed to attempt to minimize logic, subject to restrictions listed in the file. However, the current encoding should be close to optimal, given our experience with synthesis tools, so the user is strongly cautioned against any such changes.

The main state machines in the design are one-hot encoded. This type of encoding works well even in ASIC implementations because the control fields are not that wide. One-hot encoding also leads to fewer levels of logic and hence higher operating frequency in most cases.

The `ff_byte` and `ff_word` modules contain a byte-wide register and a word-wide register respectively, for use in the register file. These registers are not affected by reset.

The `reg_word` module contains a word-wide register for use in the datapath. This register is cleared to all zeros by reset and is used for temporary storage in the datapath.

The `reg_file` module contains the Y8000 register file and is instantiated in the datapath. The register file is implemented using standard flip-flops rather than latches or RAM because of its relatively small size. This is also leads to a more robust and technology-independent design.

The `ext_int` module contains the external inteface for the Y8000 processor. This is where all of the external control signals are generated with the proper timing, and all inputs are sampled with the proper timing. This module contains a mixture of rising-edge triggered logic and falling-edge triggered logic as required for compatible timing. Most outputs from this module come directly from the outputs of flip-flops, but where this is not possible, the design guarantees glitch-free operation.

The `machine` module contains the five main state machines for the device. The primary state machine is called `mach_cyc` and this state machine controls everything in the design. The four auxiliary state machines are controlled by the primary state machine and run only in specific circumstances. The `mltl_cyc` state machine runs to perform the multiply algorithm during MULT and MULTL instructions. The `divl_cyc` state machine runs to perform the divide algorithm during DIV and DIVL instructions. The `rti_cyc` state machine runs as part of the reset sequence, the trap sequence and the interrupt sequence to fetch data from the Program Status Area. The `trap_cyc` state machine runs during the trap sequence and the interrupt sequence to perform the state-saving operation that writes the FCW, PC and an identifier to the stack.

The `control` module generates all of the control signals used throughout the design. This module contains only combinatorial logic. This partitioning allows this module to be replaced with a PLA or similar structure in an ASIC implementation. Modern synthesis tools have made this option less attractive than it once was, but the option is still available with this partitioning. Multi-bit control signals are generated using the mnemonics defined in the file *defines.v* to increase the readability of this module.

The `datapath` module contains all of the data manipulation logic for the design. This includes the ALU, the register file, the PC and FCW, the PSAP and the instruction register. This module drives the internal bus, called `intrnl_bus`, which carries all data and

addresses in the design. The design uses a classical 16-bit ALU, without any special ALU logic for the multiply and divide operations. Three small state machines are included in this module. Two are used for the LDM instruction, to count the number of transfers (`inst4_reg`) and to generate the register address (`inst6_reg`). The third, called `mcnt_reg`, is used to count the iterations for the multiply and divide operations.

The `refresh` module contains the refresh controller. This includes both refresh registers, the refreshh address, refresh timer and missed refresh counter.

For more detailed information about the design refer to the design spreadsheet, *execute.xls*, and to the comments embedded in the Verilog HDL source code.

Test Bench

This chapter covers the operation of the test bench included with the design. The test bench instantiates the processor, and using a series of test patterns, verifies the operation of the processor. The included test patterns exercise every instruction and completely verify the operation of all of the flags. The test bench normally runs without any Wait states, Bus Requests, or Stop Requests. However, each of these external stimuli, alone or in combination, can be generated by the test bench.

The overall operation of the test bench is controlled by a set of `` `define `` statements at the top of the test bench file. These high-level options will be discussed first, followed by a section-by-section description of the operation of the test bench. Much of this discussion will make more sense if a copy of the test bench is available, but the general operation can be deduced without the actual Verilog HDL code at hand.

The first `` `define `` selects the type of memory used by the test bench. Normal test bench operation uses two separate memories, one for reading (and it really is read-only) and another that holds compare data for any writes that the processor will be performing. Each test pattern loads both of these memories before starting, and the test bench checks every write operation performed by the processor against the compare memory contents at that address. In this regard the test bench operates much like a piece of Automatic Test Equipment (ATE). The alternative is a uniform memory that is read and written by the processor. This option allows user code to be loaded into the test bench memory for simulation.

The second `` `define `` selects the clock cycle time for the simulation. This option is essentially meaningless when simulating the Verilog HDL source code, but will be useful if the test bench is used with a gate-level implementation of the design, as well as when generating trace files to use on a tester in the case of an ASIC implementation.

The third `` `define `` enables or disables the option of generating a trace file that can be converted into the proper format for ATE. The trace file is generated in a print-on-change format and includes the direction control signal for the address/data bus. All input stimuli are generated assuming that the ATE uses the NRZ format for all signals and that there are two ATE cycles per each processor clock cycle.

The fourth `` `define `` controls the insertion of Wait states by the test bench. This is a global enable, used to select zero, one or two wait states for each and every transaction that samples the `WAITB` input. Finer control, by each transaction type, is also available. This will be discussed in the corresponding section description below.

The fifth `define controls the insertion of Bus Requests. Only a global enable is available, with a number of options as to the length of each bus request. When enabled, a Bus Request is generated between every bus transaction. The test bench is not capable of verifying proper operation of the Bus Request except via a proper cycle count for the pattern.

The sixth `define is also used with Bus Requests. It selects the inactive time for the Bus Request. The processor requires that BUSREQB be inactive for at least two clock cycles between requests, and this is the minimum allowed by this `define. To verify proper functionality, longer inactive times may also be selected here.

The final `define controls the insertion of Stop Requests. Only a global enable is available, with a number of options as to the number of Stop Requests in a burst. When enabled, a Stop Request is generated for every instruction (recall that Stop Requests are only accepted between instructions). The test bench is not capable of verifying the proper operation of the Stop Request except via a proper cycle count for the pattern.

The remainder of this chapter will discuss each section of the Verilog HDL code of the test bench. Each section is identified by a comment block that will be referenced here.

set test cycle time

This section uses the clock cycle time selected by the top level `define to set two parameters to create the proper timing in the test bench. Both the memory access time and the minor cycle time are set here. The minor cycle time will be explained in the relevant section below.

select wait patterns

This section uses the number of wait states selected by the corresponding top level `define to create the actual data patterns which drive the WAITB input. Separate data patterns are available for each transaction type. The data patterns correspond, msb-to-lsb, to the inverse of the value applied to the WAITB starting with the T1 clock cycle of the transaction. One bit is used for each clock cycle, and WAITB can be asserted more than once per bus transaction. This is useful in the case of interrupt acknowledge cycles, where WAITB is sampled at two separate places in the transaction.

select busreq operation

This section uses the Bus Request options selected by the two top level `define statements to set three parameters used by the state machine that generates the BUSREQB input. The operation of the bus request state machine will by covered in the relevant section below.

select stop operation

This section uses the Stop Request option selected by the top level `` `define `` statement to set two parameters used by the state machine that generates the `STOPB` input. The operation of the stop request state machine will by covered in the relevant section below.

instantiate processor

This is the section where the processor is actually instantiated into the test bench. All of the pin variables are also declared here. Recall that there are actually two versions of the processor available, multiplexed and non-multiplexed, selected in the file *version.v*. This section automatically recognizes the version and connects the correct one.

program memory and write compare data

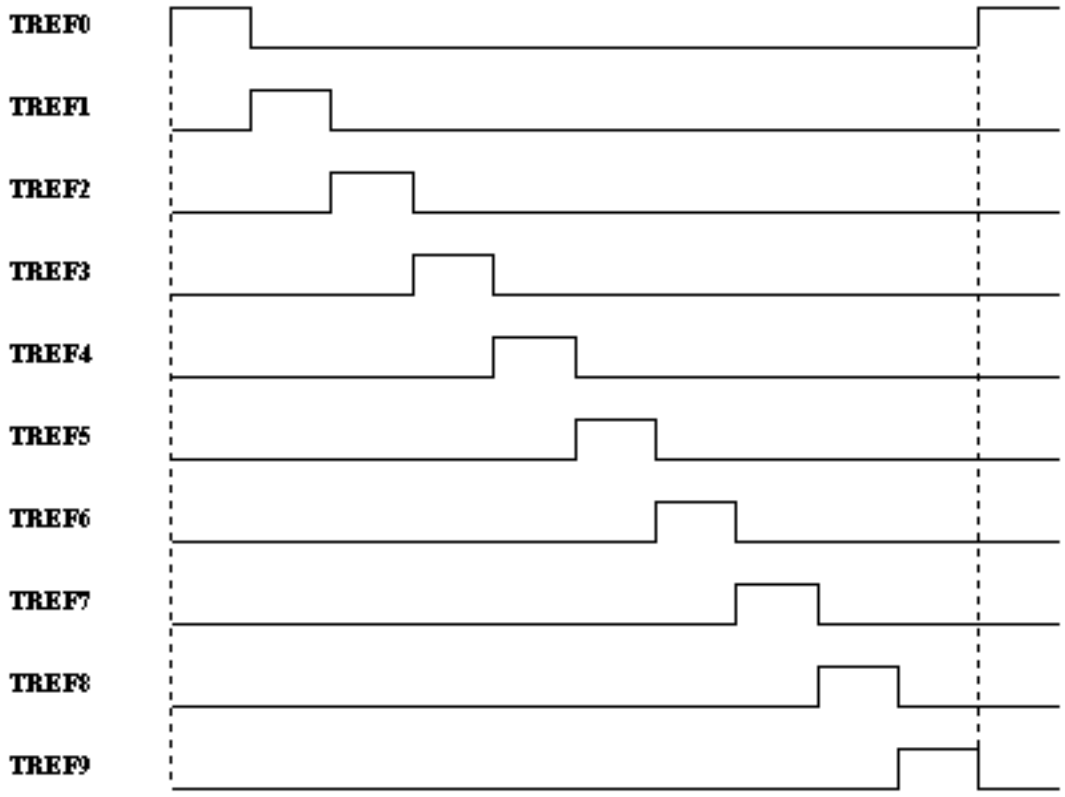
This section contains the program memory and, if necessary, the memory that holds the compare data. Both memories support both byte and word accesses. This is actually done in the memory interface section below.

test bench internal variables

This section declares all of the variables for the test bench. Most of the variables are self-explanatory. However, note that there are actually two variables per input pin. The test bench version has an extension `_var` and is the one that is manipulated by the test bench without regard to the actual timing of when the input is applied to the processor. The actual pin variable takes its value from the test bench version at the appropriate time during each clock cycle.

generate the overall test bench timing

This section contains the main timing generator for the test bench. All test bench variables are initialized and the timing generator is started here. The test bench timing generator consists of a ten bit shift register, with outputs called TREF0 through TREF9. The minor cycle time is the period of the “clock” for this state machine. Only one of these outputs is active at a time, and rising edges on these signals are used to drive state machines and generate the actual timing for the input signals. Each cycle of this timing generator corresponds to one ATE cycle and one-half of a processor clock cycle.



This section also contains a cycle counter that counts the clock cycles for a pattern. This count is not used for anything in the test bench but is provided as a convenience for anyone running the test bench. Recall that the test bench cannot verify proper operation with Wait states, Bus Requests or Stop Requests except for the proper cycle count upon pattern completion.

The end-of-pattern condition is also recognized in this section. The test bench recognizes the end of a test pattern when an address of 0xFFFE is sampled by the `ASB` signal.

peripheral simulation tasks are in a separate file

This short section uses an `` `include `` to bring in a file that contains a number of tasks used by the test bench. There is a reset task and a task to run each test pattern. Each test pattern task initializes the error counter, resets the processor and loads the memories with the test program and compare data. It also loads the interrupt generator memory, if required by the test pattern.

t-cycle tracker to generate inputs

This section contains the state machine that tracks the t-state (the clock cycle within a machine cycle) of the processor, by synchronizing with the `ASB` signal. This synchronization is necessary to assert the various processor inputs at the correct time. This section also decodes the Status lines to select the proper pattern for the Wait State generator.

busreq generator state machine

The bus request state machine uses the parameters selected earlier and the state of the `BUSACKB` pin to sequence through the states necessary to generate the requested on and off times for the Bus Request. This state machine is clocked by the rising edge of TREF4 when the `CLK` input is High. The `BUSREQB` signal is the logical AND of the two most-significant bits of the bus request state machine.

stop generator state machine

The stop request state machine uses the parameters selected earlier to sequence through the states necessary to generate a Stop Request at the correct time. This state machine is clocked by the rising edge of TREF4 when the `ASB` output is Low (in other words, at the start of every machine cycle). The `STOPB` signal is the logical AND of the two most-significant bits of the stop request state machine.

interrupt generator state machine

The interrupt request state machine uses information loaded into the interrupt generator memory at the start of each test pattern to generate interrupts during the test pattern. Every interrupt acknowledge transaction increments the address used to access the interrupt generator memory. One consequence of this is that if two interrupts are generated by an entry, there must be a subsequent dummy entry to account for the second interrupt acknowledge transaction. Similarly, if three interrupts are generated by an entry, two subsequent dummy entries are required.

Each entry in the interrupt generator memory contains one bit for each interrupt type, a four bit transaction count, a four bit transaction status and a sixteen bit address. The interrupt request state machine contains a counter that is reset by an interrupt acknowledge

transaction and then increments each time that a transaction matching the address and status is recognized. When the transaction count, transaction status and address all match those in the interrupt generator memory, the selected interrupt is generated at the correct time during the transaction.

Bit position(s)	Width	Meaning
27	1	Value for SEGTB
26	1	Value for NMIB
25	1	Value for NVIB
24	1	Value for VIB
23:20	4	Count
19:16	4	Transaction Status
15:0	16	Transaction Address

Once asserted, the SEGTB, VIB and NVIB interrupt requests remain active until the test bench recognizes the corresponding interrupt acknowledge transaction. The NMIB interrupt request is different, however. In the case of NMIB, the request is a pulse that is only one-half of a clock cycle wide. This is to verify the edge-triggered operation on this input to the processor.

Every test pattern except for the one which actually tests the interrupts (called int_ops) merely loads a null value into the interrupt generator memory.

processor inputs

This section adds the final timing to the test bench variables to create the actual processor inputs. The CLK input changes on the rising edge of TREF5, and all of the other inputs change on the rising edge of TREF3. In the case of BUSREQB, NVIB, RESETB, SEGTB and VIB it is the rising edge of TREF3 just before a rising edge of CLK. In the case of NMIB, STOPB and WAITB it is the rising edge of TREF3 just before a falling edge of CLK. Note that the NVIB, RESETB, SEGTB and VIB inputs persist. That is, the value is latched by the test bench and persists until the corresponding test bench variable is de-asserted. In all other cases the input is actually driven for only half of a clock cycle and is then returned to the inactive state. The test bench does this to guarantee that the processor input is actually sampled on the correct CLK edge.

memory interface

This section connects the test bench memories to the processor. The address/data bus is latched by the rising edge of TREF4 during the ASB Low time. Input data is applied to the processor on the rising edge of TREF2 during the CLK Low time (after the memory access time specified previously). The memory interfaces support both byte and word transfers properly.

trace file generation

This section generates the print-on-change trace file, if that option is selected. The complete test program requires slightly more than TBD clock cycles to complete. If the tester pattern memory is not deep enough for the entire set of patterns, individual patterns can be enabled or disabled in the next section by commenting out the appropriate task.

load and execute the test patterns

This final section is where the actual test pattern tasks are run. Note that they can be individually commented out if required. The existing pattern sequence is arbitrary, and the patterns can actually be run in any order. Even though each pattern task does a hardware reset, there is an initial reset task to force all processor outputs to be known. This should always be the first task to run for this reason. Below is a list of the test patterns and their functionality:

Pattern Name	Pattern Number	Coverage
dat_ops	0	data movement instructions
alu_ops	1	arithmetic and logical instructions
jmp_ops	2	jumps, calls, and return
bit_ops	3	bit operation instructions
lng_ops	4	multiply, divide and multi-bit shifts
blk_ops	5	block-type instructions
seg_ops	6	all opcodes using segment addresses
int_ops	7	interrupts