Y8002 Microprocessor

Technical Manual

Systemyde International Corporation

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 Y8002 design defines “proper operation”.

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

Chapter 1

Introduction

This book documents the operation of the Y8002 microprocessor. The Y8002 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 15K 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.

The Y8002 CPU was designed in a clean-room environment and is a clone of the Zilog Z8002 microprocessor. Only publicly available documentation was used to create this design so there may be minor differences where the public documentation is misleading or lacking. With only a couple of exceptions, the instruction execution times are identical between the two designs. All known differences for individual instructions are listed in the instruction description chapter as notes.

The Y8002 design, depending on the version, may not implement all of the instructions, features or operating modes of the Z8000 architecture. The specific differences, for any given version of the design, are covered in the various appendices.

The Z8002 CPU is one of four variants of the Z8000 architecture, introduced by Zilog in 1979. 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 and are identical except that the Z8003 added support for virtual memory. All of these devices were implemented in NMOS technology and the Z8001 CPU and the Z8002 CPU were available for -55C to +125C operation. Manufacturing of these devices ceased around 1990.

This document should always be used as the final word on the operation of the Y8002 CPU, but it is useful to refer to the Zilog documentation if the description given here is too cryptic. The Z8000 architecture is over twenty years old, so it is assumed that it is already at least somewhat familiar to the reader, but an overview is presented here. This document will make no attempt to describe the segmented addressing mode of the Z8000 architecture because it is not present in the Y8002 CPU.

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 byte strings. The architecture supports eight different types of addressing modes. A Z8000 architecture CPU (like the Y8002) has 111 instruction types in its instruction set. The multiple addressing modes and data types, when coupled with the instruction types, produces 447 different instructions for the Y8002 processor 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, each space being dedicated to a specific purpose.

The 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 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 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 is used to provide access to each of the segments. As mentioned previously, the Y8002 processor does not implement the segmented memory scheme. It is capable of addressing six separate memory spaces, each being a maximum of 64K bytes.

The architecture includes a refresh-control block that is designed to generate memory refresh cycles for dynamic random access memory that may be used in conjunction with the processor.

The 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 architecture 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 extended processing unit 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.

Chapter 2

Programming Model

The 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 as R0 through R15. Sixteen byte registers (RH0-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.

The Z8000 architecture includes 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 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.

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 Y8002 CPU fetches the FCW from memory address 0x0002 and the PC from memory address 0x0004 before starting execution. Both of these addresses are in the Program address space.

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 the processor instructions to control program branching and looping. The high order byte contains processor control bits that are used to enable and disable the processor interrupt system and control certain processor 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 the processor is the complement of this bit.

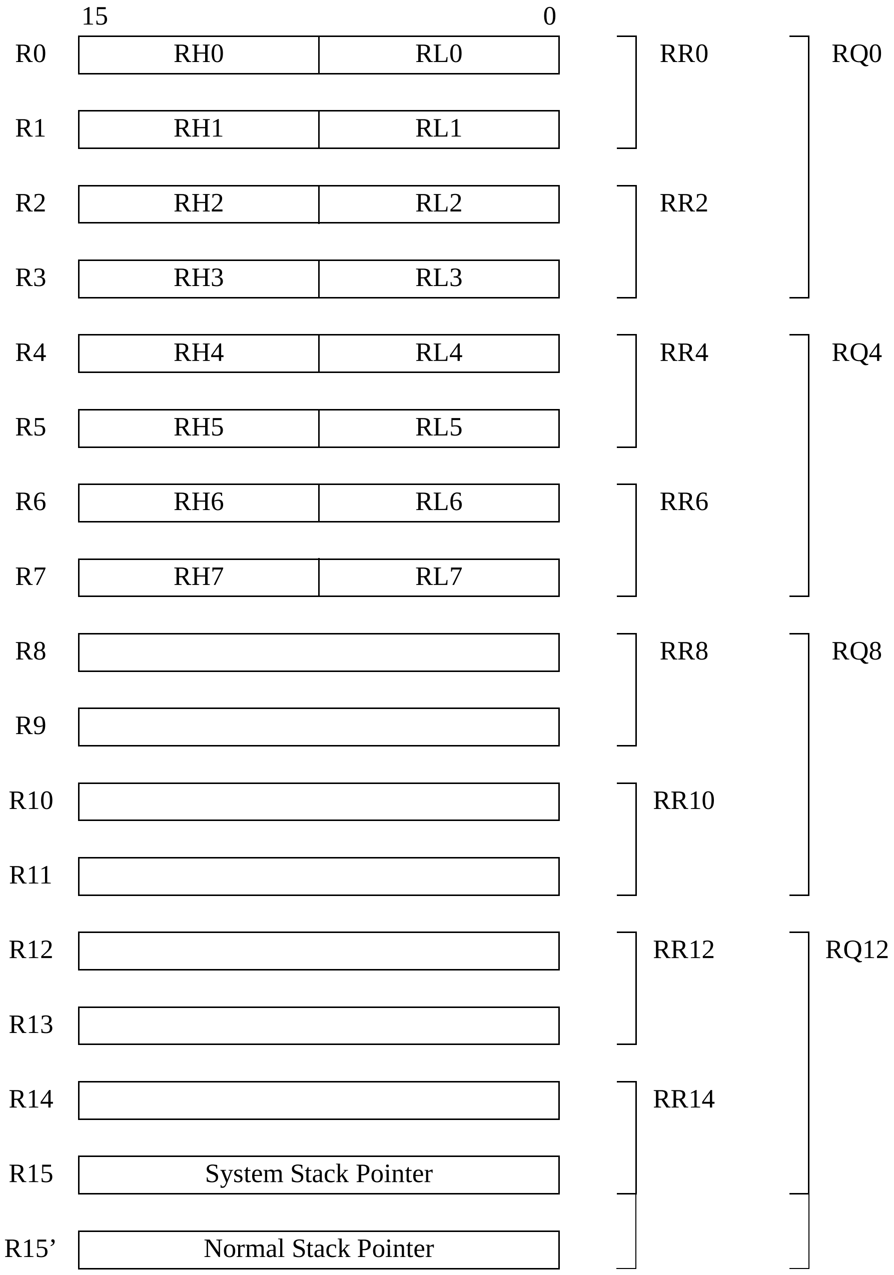
- Extended Processor Architecture (EPA) -- This bit indicates the presence or absence of Extended Processing Units (EPU) in the system architecture. If EPUs are present, this bit should be set to one, and the processor will execute extended instructions as they are encountered. If EPUs are not present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

- Segmentation Mode (SEG) -- This bit is present only in a segmented Z8000 processor. 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 is permanently set to 0 in the Y8002 processor.

The Program Counter 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 or checks that this is true.

The Program Status Area Pointer (PSAP) contains the address of the Program Status Area, 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 lower byte of the PSAP is always zero. The PSAP points to an area in the Program memory address space and is cleared to all zeros by reset.

The Z8000 architecture contains hardware that can be used to automatically refresh dynamic memory in the system. The Refresh Control register contains a 9-bit row counter, and a six-bit rate counter, as well as an enable/disable control bit. Bit 15 of this register is cleared by reset. Modern dynamic RAMs have no need of this feature.



General Purpose Registers

	15															0
FCW	SEG	SYS	EPA	VI	NVI	0	0	0	C	Z	S	V	D	H	0	0
PC	Address															

Program Status Registers

Upper Pointer								0	0	0	0	0	0	0	0

Program Status Area Pointer

REN	Rate						Row								

Refresh Control

Chapter 3

Addressing and 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 that is 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.

- 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 controller (DMA).

In this architecture, each of the memory address spaces can be a maximum of 64K bytes, allowing the Y8002 CPU to address 384 KB of memory. Each I/O address space is a maximum of 64K port addresses.

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.

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 long words

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

bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bytes	address “n”								address “n+1”							
word	address “n” (“n” is even)															
double	address “n” (“n” is even)															
word	address “n+2”															

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, long word) is the same as that of the byte with the lowest memory address within the word or long word. This byte is the leftmost, highest-order (most significant) byte of the word or long word.

Word and long 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 0x0000-0x0005 in the Program memory space are reserved for the FCW 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 address
- Byte of packed BCD (binary coded decimal) integer
- Dynamic-length string of byte data
- Dynamic-length of word data
- Dynamic-length of stack 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 double register 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.
- 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 plus the displacement in the instruction.
- Base Index: the contents of the location addressed by the sum of the contents of one register plus the contents of another register.

Chapter 4

Instruction Format

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

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.

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>Byte</td><td colspan="2">ad mode</td><td colspan="5">opcode</td><td>0</td><td colspan="4">source or destination</td><td colspan="4">source or destination</td></tr>
<tr><td>Word</td><td colspan="2">ad mode</td><td colspan="5">opcode</td><td>1</td><td colspan="4">source or destination</td><td colspan="4">source or destination</td></tr>
<tr><td>Long</td><td colspan="2">ad mode</td><td colspan="6">opcode</td><td colspan="4">source or destination</td><td colspan="4">source or destination</td></tr>
</table>

General Instruction Format (first word of instruction)

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.

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>LDB</td><td>1</td><td>1</td><td>0</td><td>0</td><td colspan="4">destination</td><td colspan="8">byte data</td></tr>
<tr><td>CALR</td><td>1</td><td>1</td><td>0</td><td>1</td><td colspan="12">offset</td></tr>
<tr><td>JR</td><td>1</td><td>1</td><td>1</td><td>0</td><td colspan="4">condition code</td><td colspan="8">offset</td></tr>
<tr><td>DJNZ</td><td>1</td><td>1</td><td>1</td><td>1</td><td colspan="4">register</td><td>W</td><td colspan="7">offset</td></tr>
</table>

Special Compact Instruction Format

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

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	register				register				condition code			
	0	0	0	0	register				0	0	0	0	identifier			
	0	0	0	0	register				0	0	0	0	0	0	0	0
EPU	reserved for EPU use												iteration count			
EPU	reserved for EPU use				register				reserved for EPU use				iteration count			

General Instruction Format (second word of instruction)

Chapter 5

Instruction Set

This chapter presents the assembly language syntax, addressing modes, flag settings, binary encoding, and execution time for the Y8002 instruction set. The entire instruction set is presented in alphabetical order, without regard to whether or not a particular instruction has been implemented in a particular version of the design. For a list of implemented versus unimplemented instructions, refer to the appropriate Appendix.

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, employing the following shortcuts:

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 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. The illegal cases for RRdd, RRss and RQdd should not be used. The instructions will still execute with an illegal register encoding, but the results will be scrambled in the registers because of the way the register addresses are treated internally in the design

encoding in opcode	Rbss or Rbdd	Rddd, Rsss, Rrrr or Rxxx	Rsnz, Rdnz, R1nz or 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. Note that 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:

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. Note that 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 execution times are listed here only as a number of clock cycles. These numbers assume no wait states and no interrupts during execution of an iterative instruction. The details of both internal and external execution sequences are available in an Appendix.

ADC

Add With Carry

ADC dst, src	dst: R
ADCB	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	10110101_Rsss_Rddd	5
	ADCB Rbd, Rbs	10110100_Rbss_Rbdd	5

ADD

Add

ADD dst, src	dst: R
ADDB	src: R, IM, IR, DA, X
ADDL	

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).
H: Set if arithmetic carry from bit 3; cleared otherwise (ADDB);
Unaffected (ADD).

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	ADD Rd, Rs	10000001_Rsss_Rddd	4
	ADDB Rbd, Rbs	10000000_Rbss_Rbdd	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	00000001_Rsnz_Rddd	7
	ADDB Rbd, @Rs	00000000_Rsnz_Rbdd	7
	ADDL RRd, @Rs	00010110_Rsnz_RRdd	14
DA:	ADD Rd, address	01000001_0000_Rddd address	9
	ADDB Rbd, address	01000000_0000_Rbdd address	9
	ADDL RRd, address	01010110_0000_RRdd address	15
X:	ADD Rd, addr(Rs)	01000001_Rsnz_Rddd address	10
	ADDB Rbd, addr(Rs)	01000000_Rsnz_Rbdd address	10
	ADDL RRd, addr(Rs)	01010110_Rsnz_RRdd address	16

AND

Logical AND

AND dst, src	dst: R
ANDB	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	10000111_Rsss_Rddd	4
	ANDB Rbd, Rbs	10000110_Rbss_Rbdd	4
IM:	AND Rd, #data	00000111_0000_Rddd #data	7
	ANDB Rbd, #data	00000110_0000_Rbdd #data | #data	7
IR:	AND Rd, @Rs	00000111_Rsnz_Rddd	7
	ANDB Rbd, @Rs	00000110_Rsnz_Rbdd	7
DA:	AND Rd, address	01000111_0000_Rddd address	9
	ANDB Rbd, address	01000110_0000_Rbdd address	9

Addressing Modes	Assembly Syntax	Encoding	Clocks
X:	AND Rd, addr(Rs)	01000111_Rsnz_Rddd address	10
	ANDB Rbd, addr(Rs)	01000110_Rsnz_Rbdd address	10

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	10100111_Rddd_bbbb	4
	BITB Rbd, #b	10100110_Rbdd_0bbb	4
IR:	BIT @Rd, #b	00100111_Rdnz_bbbb	8
	BITB @Rd, #b	00100110_Rdnz_0bbb	8
DA:	BIT address, #b	01100111_0000_bbbb address	10
	BITB address, #b	01100110_0000_0bbb address	10
X:	BIT addr(Rd), #b	01100111_Rdnz_bbbb address	11
	BITB addr(Rd), #b	01100110_Rdnz_0bbb address	11

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	00100111_0000_Rsss 0000_Rddd_0000_0000	10
	BITB Rbd, Rs	00100110_0000_Rsss 0000_Rbdd_0000_0000	10

Notes:

1. The Z8000 microprocessor restricts the source register to be one of R0 - R7 for BITB. This restriction does not apply to the Y8002 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
DA:	CALL address	01011111_0000_0000 address	12
X:	CALL address(Rd)	01011111_Rdnz_0000 address	13

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 CALL @Rd a data (or stack) memory access at the address in Rd is performed but the data is discarded.

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

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

Notes:

1. The Zilog documentation incorrectly states that the displacement is *added* to the Program Counter. The Zilog Z8000 devices actually *subtract* the displacement from the Program Counter. The Y8002 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.

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	10001101_Rddd_1000	7
	CLRB Rbd	10001100_Rbdd_1000	7
IR:	CLR @Rd	00001101_Rdnz_1000	8
	CLRB @Rd	00001100_Rdnz_1000	8
DA:	CLR address	01001101_0000_1000 address	11
	CLRB address	01001100_0000_1000 address	11
X:	CLR addr(Rd)	01001101_Rdnz_1000 address	12
	CLRB addr(Rd)	01001100_Rdnz_1000 address	12

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	10001101_Rddd_0000	7
	COMB Rbd	10001100_Rbdd_0000	7
IR:	COM @Rd	00001101_Rdnz_0000	12
	COMB @Rd	00001100_Rdnz_0000	12
DA:	COM address	01001101_0000_0000 address	15
	COMB address	01001100_0000_0000 address	15
X:	COM addr(Rd)	01001101_Rdnz_0000 address	16
	COMB addr(Rd)	01001100_Rdnz_0000 address	16

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. The Z8000 documentation lists the H flag as undefined, when in fact it is always complemented. The Y8002 design matches this behavior.

CP

Compare Register

CP dst, src	dst: R
CPB	src: R, IR, DA, X
CPL	

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	10001011_Rsss_Rddd	4
	CPB Rbd, Rbs	10001010_Rbss_Rbdd	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	00001011_Rsnz_Rddd	7
	CPB Rbd, @Rs	00001010_Rsnz_Rbdd	7
	CPL RRd, @Rs	00010000_Rsnz_RRdd	14
DA:	CP Rd, address	01001011_0000_Rddd address	9
	CPB Rbd, address	01001010_0000_Rbdd address	9
	CPL RRd, address	01010000_0000_RRdd address	15
X:	CP Rd, addr(Rs)	01001011_Rsnz_Rddd address	10
	CPB Rbd, addr(Rs)	01001010_Rsnz_Rbdd address	10
	CPL RRd, addr(Rs)	01010000_Rsnz_RRdd address	16

CP

Compare Immediate

CP dst, src dst: IR, DA, X
CPB 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 address #data	14
	CPB address, #data	01001100_0000_0001 address #data | #data	14
X:	CP (addr)Rd, #data	01001101_Rdnz_0001 address #data	15
	CPB (addr)Rd, #data	01001100_Rdnz_0001 address #data | #data	15

CPD

Compare and Decrement

CPD dst, src, r, cc	dst: R
CPDB	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	10111011_Rsnz_1000 0000_Rrrr_Rddd_cccc	20
	CPDB Rd, @Rs, r, cc	10111010_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	dst: R
CPDRB	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	10111011_Rsnz_1100 0000_Rrrr_Rddd_cccc	11 + 9n
	CPDRB Rd, @Rs, r, cc	10111010_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.

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	dst: R
CPIB	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	10111011_Rsnz_0000 0000_Rrrr_Rddd_cccc	20
	CPIB Rd, @Rs, r, cc	10111010_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	dst: R
CPIRB	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	10111011_Rsnz_0100 0000_Rrrr_Rddd_cccc	11 + 9n
	CPIRB Rd, @Rs, r, cc	10111010_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.

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 Increment

CPSD dst, src, r, cc	dst: IR
CPSDB	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	10111011_Rsnz_1010 0000_Rrrr_Rdnz_cccc	25
	CPSDB @Rd, @Rs, r, cc	10111010_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	dst: IR
CPSDRB	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	10111011_Rsnz_1110 0000_Rrrr_Rdnz_cccc	11 + 14n
	CPSDRB @Rd, @Rs, r, cc	10111010_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	dst: IR
CPSIB	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	10111011_Rsnz_0010 0000_Rrrr_Rdnz_cccc	25
	CPSIB @Rd, @Rs, r, cc	10111010_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	dst: IR
CPSIRB	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	10111011_Rsnz_0110 0000_Rrrr_Rdnz_cccc	11 + 14n
	CPSIRB @Rd, @Rs, r, cc	10111010_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
ADDB or ADCB	0	0-9	0	0-9	00	0
	0	0-8	0	A-F	06	0
	0	0-9	1	0-3	06	0
	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
SUBB or SBCB	0	0-9	0	0-9	00	0
	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	dst: R, IR, DA, X
DECB	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	10101011_Rddd_nnnn	4
	DECB Rbd, #n	10101010_Rbdd_nnnn	4
IR:	DEC @Rd, #n	00101011_Rdnz_nnnn	11
	DECB @Rd, #n	00101010_Rdnz_nnnn	11
DA:	DEC address, #n	01101011_0000_nnnn address	13
	DECB address, #n	01101010_0000_nnnn address	13
X:	DEC addr(Rd), #n	01101011_Rdnz_nnnn address	14
	DECB addr(Rd), #n	01101010_Rdnz_nnnn address	14

DI

Disable Interrupt

DI int   int: VI, NVI

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 address	10 + div
	DIVL RQd, address	01011010_0000_RQdd address	13 + divl
X:	DIV RRd, addr(Rs)	01011011_Rsnz_RRdd address	11 + div
	DIVL RQd, addr(Rs)	01011010_Rsnz_RQdd address	14 + divl

Notes:

1. The Zilog 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 Y8002 design matches the behavior of the Zilog devices.

2. In the case of overflow, the destination register(s) and S flag are undefined. The Y8002 design does not attempt to match the Z8000 microprocessor’s undefined value for this case, and the Y8002 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.

5. The execution time and different flag cases for DIV are shown in the table below:

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

Notes (continued):

6. The execution time and different flag cases for DIVL are shown in the table below:

DIVL case	C	Z	S	V	divl cycles
1. Quotient is a 32-bit twos-complement number.	0	zero	sign	0	494
2. Divisor is zero. The destination register is unchanged in this case.	0	1	0	1	8
3. Quotient is too large to represent in 33 bits. The destination register is undefined in this case.	0	0	0	1	30
4. 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.

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	1111Rrrr_1ddd_dddd	11
	DBJNZ Rb, address	1111Rrrr_0ddd_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

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	10101101_Rsss_Rddd	6
	EXB Rbd, Rbs	10101100_Rbss_Rbdd	6
IR:	EX Rd, @Rs	00101101_Rsnz_Rddd	12
	EXB Rbd, @Rs	00101100_Rsnz_Rbdd	12
DA:	EX Rd, address	01101101_0000_Rddd address	15
	EXB Rbd, address	01101100_0000_Rbdd address	15
X:	Ex, Rd, addr(Rs)	01101101_Rsnz_Rddd address	16
	EXB Rbd, addr(Rs)	01101100_Rsnz_Rbdd address	16

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

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. 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
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	00111101_Rsnz_Rddd	10
	INB Rbd, @Rs	00111100_Rsnz_Rbdd	10
DA:	IN Rd, port	00111011_Rddd_0100 port	12
	SIN Rd, port	00111011_Rddd_0101 port	12
	INB Rbd, port	00111010_Rbdd_0100 port	12
	SINB Rbd, port	00111010_Rbdd_0101 port	12

Notes:

1. Data is read from the I/O or Special I/O address space. Only the status code is different between IN and SIN.

2. I/O reads always have one automatic WAIT state. This is included in the Cycles number above.

INC

Increment

INC dst, src	dst: R, IR, DA, X
INCB	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	10101001_Rddd_nnnn	4
	INCB Rbd, #n	10101000_Rbdd_nnnn	4
IR:	INC @Rd, #n	00101001_Rdnz_nnnn	11
	INCB @Rd, #n	00101000_Rdnz_nnnn	11
DA:	INC address, #n	01101001_0000_nnnn address	13
	INCB address, #n	01101000_0000_nnnn address	13
X:	INC addr(Rd), #n	01101001_Rdnz_nnnn address	14
	INCB addr(Rd), #n	01101000_Rdnz_nnnn address	14

(S)IND

(Special) Input and Decrement

IND dst, src, r, cc	dst: IR
INDB	src: IR
SIND	
SINDB	

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	00111011_Rsnz_1000 0000_Rrrr_Rdnz_1000	21
	SIND @Rd, @Rs, r	00111010_Rsnz_1001 0000_Rrrr_Rdnz_1000	21
	INDB @Rd, @Rs, r	00111011_Rsnz_1000 0000_Rrrr_Rdnz_1000	21
	SINDB @Rd, @Rs, r	00111010_Rsnz_1001 0000_Rrrr_Rdnz_1000	21

Notes:

1. The Z8000 documentation lists the Z flag as undefined. In the Y8002 design this flag is unaffected.

Notes (continued):

2. Data is read from the I/O or Special I/O address space. Only the status code is different between IND and SIND.

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

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

5. Bits 3-0 of the second word of the opcode are actually a condition code. The result of this condition code check is ORed with the r = 0 check to determine if the instruction is complete. The bit combination 1000 is the encoding for “True”, making this a non-repeating instruction.

(S)INDR

(Special) Input, Decrement and Repeat

INDR dst, src, r, cc	dst: IR
INDRB	src: IR
SINDR	
SINDRB	

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	00111011_Rsnz_1000 0000_Rrrr_Rdnz_0000	11 + 10n
	SINDR @Rd, @Rs, r	00111010_Rsnz_1001 0000_Rrrr_Rdnz_0000	11 + 10n
	INDRB @Rd, @Rs, r	00111011_Rsnz_1000 0000_Rrrr_Rdnz_0000	11 + 10n
	SINDR @Rd, @Rs, r	00111010_Rsnz_1001 0000_Rrrr_Rdnz_0000	11 + 10n

Notes:

1. The Z8000 documentation lists the Z flag as undefined. In the Y8002 design this flag is unaffected.

Notes (continued):

2. Data is read from the I/O or Special I/O address space. Only the status code is different between IND and SIND.

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

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

5. Bits 3-0 of the second word of the opcode are actually a condition code. The result of this condition code check is ORed with the r = 0 check to determine if the instruction is complete. The bit combination 0000 is the encoding for “False”, making this a repeating 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

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	00111011_Rsnz_0000 0000_Rrrr_Rdnz_1000	21
	SINI @Rd, @Rs, r	00111010_Rsnz_0001 0000_Rrrr_Rdnz_1000	21
	INIB @Rd, @Rs, r	00111011_Rsnz_0000 0000_Rrrr_Rdnz_1000	21
	SINIB @Rd, @Rs, r	00111010_Rsnz_0001 0000_Rrrr_Rdnz_1000	21

Notes:

1. The Z8000 documentation lists the Z flag as undefined. In the Y8002 design this flag is unaffected.

Notes (continued):

2. Data is read from the I/O or Special I/O address space. Only the status code is different between INI and SINI.

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

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

5. Bits 3-0 of the second word of the opcode are actually a condition code. The result of this condition code check is ORed with the r = 0 check to determine if the instruction is complete. The bit combination 1000 is the encoding for “True”, making this a non-repeating instruction.

(S)INIR

(Special) Input, Increment and Repeat

INIR dst, src, r, cc
INIRB
SINIR
SINIRB

dst: IR
src: IR

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	00111011_Rsnz_0000 0000_Rrrr_Rdnz_0000	11 + 10n
	SINIR @Rd, @Rs, r	00111010_Rsnz_0001 0000_Rrrr_Rdnz_0000	11 + 10n
	INIRB @Rd, @Rs, r	00111011_Rsnz_0000 0000_Rrrr_Rdnz_0000	11 + 10n
	SINIR @Rd, @Rs, r	00111010_Rsnz_0001 0000_Rrrr_Rdnz_0000	11 + 10n

Notes:

1. The Z8000 documentation lists the Z flag as undefined. In the Y8002 design this flag is unaffected.

Notes (continued):

2. Data is read from the I/O or Special I/O address space. Only the status code is different between IND and SIND.

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

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

5. Bits 3-0 of the second word of the opcode are actually a condition code. The result of this condition code check is ORed with the r = 0 check to determine if the instruction is complete. The bit combination 0000 is the encoding for “False”, making this a repeating 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

Operation:

```
SP <= SP + 2
FCW <= @SP
SP <= SP + 2
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

Notes:

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

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	10/7
DA:	JP cc, address	01011110_0000_cccc address	7
X:	JP cc, address(Rd)	01011110_Rdnz_cccc address	8

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 JP cc, @Rd a data (or stack) memory access at the address in Rd is performed but the data is discarded.

3. The execution time is 7 clocks if cc is false and 10 clocks if cc is true.

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

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	10100001_Rsss_Rddd	3
	LDB Rbd, Rbs	10100000_Rbss_Rbdd	3
	LDL RRd, RRs	10010100_RRss_RRdd	5
IR:	LD Rd, @Rs	00100001_Rsnz_Rddd	7
	LDB Rbd, @Rs	00100000_Rsnz_Rbdd	7
	LDL RRd, @Rs	00010100_Rsnz_RRdd	11
DA:	LD Rd, address	01100001_0000_Rddd address	9
	LDB Rbd, address	01100000_0000_Rbdd address	9
	LDL RRd, address	01010100_0000_RRdd address	12

Addressing Modes	Assembly Syntax	Encoding	Clocks
X:	LD Rd, addr(Rs)	01100001_Rsnz_Rddd address	10
	LDB Rbd, addr(Rs)	01100000_Rsnz_Rbdd address	10
	LDL RRd, addr(Rs)	01010100_Rsnz_RRdd address	13
BA:	LD Rd, Rs(#disp)	00110001_Rsnz_Rddd displacement	14
	LDB Rbd, Rs(#disp)	00110000_Rsnz_Rbdd displacement	14
	LDL RRd, Rs(#disp)	00110101_Rsnz_RRdd displacement	17
BX:	LD Rd, Rs(Rx)	01110001_Rsnz_Rddd 0000_Rxxx_0000_0000	14
	LDB Rbd, Rs(Rx)	01110000_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	00101111_Rdnz_Rsss	8
	LDB @Rd, Rbs	00101110_Rdnz_Rbss	8
	LDL @Rd, RRs	00011101_Rdnz_RRss	11
DA:	LD address, Rs	01101111_0000_Rsss address	11
	LDB address, Rbs	01101110_0000_Rbss address	11
	LDL address, RRs	01011101_0000_RRss address	14

Addressing Modes	Assembly Syntax	Encoding	Clocks
X:	LD addr(Rd), Rs	01101111_Rdnz_Rsss address	12
	LDB addr(Rd), Rbs	01101110_Rdnz_Rbss address	12
	LDL addr(Rd), RRs	01011101_Rdnz_RRss address	15
BA:	LD Rd(#disp), Rs	00110011_Rdnz_Rsss displacement	14
	LDB Rd(#disp), Rbs	00110010_Rdnz_Rbss displacement	14
	LDL Rd(#disp), RRs	00110111_Rdnz_RRss displacement	17
BX:	LD Rd(Rx), Rs	01110011_Rdnz_Rsss 0000_Rxxx_0000_0000	14
	LDB Rd(Rx), Rbs	01110010_Rdnz_Rbss 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	dst: R, IR, DA, X
LDB	src: IM
LDL	

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
	LD Rbd, #data	00100000_0000_Rbdd #data | #data	7
		1100Rbdd_data8data	5
	LDL RRd, #data	00010100_0000_RRdd #data (high) #data (low)	11
IR:	LD @Rd, #data	00001101_Rdnz_0101 #data	11
	LDB @Rd, #data	00001100_Rdnz_0101 #data | #data	11

Addressing Modes	Assembly Syntax	Encoding	Clocks
DA:	LD address, Rs	01001101_0000_0101 address #data	14
	LDB address, Rbs	01001100_0000_0101 address #data | #data	14
X:	LD addr(Rd), Rs	01001101_Rdnz_0101 address #data	15
	LDB addr(Rd), Rbs	01001100_Rdnz_0101 address #data | #data	15

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.

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 address	12
X:	LDA Rd, addr(Rs)	01110110_Rsnz_Rddd address	13
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. A data (or stack) memory access at the effective address is performed but the data is discarded.

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

Notes:

1. A program memory access at the effective address is performed but the data is discarded.

LDCTL

Load Into Control Register

LDCTL dst, src

dst: control register
src: R

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 PSAP, Rs	01111101_Rsss_1101	7

Notes:

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

LDCTL

Load From Control Register

LDCTL dst, src

dst: R
src: control register

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	
	LDCTL Rd,NSP	01111101_Rddd_0111	
	LDCTL Rd, PSAP	01111101_Rddd_0101	7

Notes:

1. All unused FCW bits return zeros.

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 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:	LDCTBL Rbd, FLAGS	10001100_Rbdd_0001	7

Notes:

1. All unused FLAGS bits return zeros.

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	10111011_Rsnz_1001 0000_Rrrr_Rdnz_1000	20
	LDDB @Rd, @Rs, r	10111010_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 Y8002 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.

4. Bits 3-0 of the second word of the opcode are actually a condition code. The result of this condition code check is ORed with the r = 0 check to determine if the instruction is complete. The bit combination 1000 is the encoding for “True”, making this a non-repeating instruction.

LDDR

Load, Increment and Repeat

LDDR dst, src, r, cc	dst: IR
LDDRB	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	10111011_Rsnz_1001 0000_Rrrr_Rdnz_0000	11 + 9n
	LDDRB @Rd, @Rs, r	10111010_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 Y8002 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.

Notes (continued):

4. Bits 3-0 of the second word of the opcode are actually a condition code. The result of this condition code check is ORed with the r = 0 check to determine if the instruction is complete. The bit combination 0000 is the encoding for “False”, making this a repeating instruction.

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 dst: IR
LDIB 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	10111011_Rsnz_0001 0000_Rrrr_Rdnz_1000	20
	LDIB @Rd, @Rs, r	10111010_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 Y8002 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.

4. Bits 3-0 of the second word of the opcode are actually a condition code. The result of this condition code check is ORed with the r = 0 check to determine if the instruction is complete. The bit combination 1000 is the encoding for “True”, making this a non-repeating instruction.

LDIR

Load, Increment and Repeat

LDIR dst, src, r, cc	dst: IR
LDIRB	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	10111011_Rsnz_0001 0000_Rrrr_Rdnz_0000	11 + 9n
	LDIRB @Rd, @Rs, r	10111010_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 Y8002 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.

Notes (continued):

4. Bits 3-0 of the second word of the opcode are actually a condition code. The result of this condition code check is ORed with the r = 0 check to determine if the instruction is complete. The bit combination 0000 is the encoding for “False”, making this a repeating instruction.

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 address	14 + 3n
X:	LDM Rd, addr(Rs), #n	01011100_Rsnz_0001 0000_Rddd_0000_nnnn address	15 + 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 address	14 + 3n
X:	LDM addr(Rd), Rs, #n	01011100_Rdnz_1001 0000_Rsss_0000_nnnn address	15 + 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

Operation: PS <= src (FCW from src address, PC from subsequent address)

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
DA:	LDPS address	01111001_0000_0000 address	16
X:	LDPS addr(Rs)	01111001_Rsnz_0000 address	17

LDR

Load Relative Register

LDR dst, src	dst: R
LDRB	src: RA
LDRL	

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	00110001_0000_Rddd displacement	14
	LDRB Rbd, address	00110000_0000_Rbdd 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.

3. Data is read from the program address space.

LDR

Load Relative Memory

LDR dst, src	dst: RA
LDRB	src: R
LDRL	

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	00110011_0000_Rsss displacement	14
	LDRB address, Rbs	00110010_0000_Rbss 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.

3. Data is read from the program address space.

MBIT

Multi-Micro Bit Test

MBIT

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

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) begin
        FCW[6:5] <= 0x11    //request granted
        end
    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 and can take a long time to execute if the value in Rd is very big.

MRES

Multi-Micro Reset

MRES

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

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	dst: R
MULTL	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
DA:	MULT RRd, address	01011001_0000_RRdd address	10 + mult
	MULTL RQd, address	01011000_0000_RQdd address	13 + mult

Addressing Modes	**Assembly Syntax**	**Encoding**	**Clocks**
X:	MULT RRd, addr(Rs)	01011001_Rsnz_RRdd address	11 + mult
	MULTL RQd, addr(Rs)	01011000_Rsnz_RQdd address	14 + multl

Notes:

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

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

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

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

5. 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 cycles
1. Product requires 64 bits.	1	252 + 4*m*
2. Product is zero.	0	15
3. Product can be represented in 32 bits. Most significant two words are merely sign-extension data.	0	252 + 4*m*

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

NEG

Negate

NEG dst dst: R, IR, DA, X
NEGB

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	10001101_Rddd_0010	7
	NEGB Rbd	10001100_Rbdd_0010	7
IR:	NEG @Rd	00001101_Rdnz_0010	12
	NEGB @Rd	00001100_Rdnz_0010	12
DA:	NEG address	01001101_0000_0010 address	15
	NEGB address	01001100_0000_0010 address	15
X:	NEG addr(Rd)	01001101_Rdnz_0010 address	16
	NEGB addr(Rd)	01001100_Rdnz_0010 address	16

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	dst: R
ORB	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	10000101_Rsss_Rddd	4
	ORB Rbd, Rbs	10000100_Rbss_Rbdd	4
IM:	OR Rd, #data	00000101_0000_Rddd #data	7
	ORB Rbd, #data	00000100_0000_Rbdd #data | #data	7
IR:	OR Rd, @Rs	00000101_Rsnz_Rddd	7
	ORB Rbd, @Rs	00000100_Rsnz_Rbdd	7
DA:	OR Rd, address	01000101_0000_Rddd address	9
	ORB Rbd, address	01000100_0000_Rbdd address	9

Addressing Modes	Assembly Syntax	Encoding	Clocks
X:	OR Rd, addr(Rs)	01000101_Rsnz_Rddd address	10
	ORB Rbd, addr(Rs)	01000100_Rsnz_Rbdd address	10

(S)OTDR

(Special) Output, Decrement and Repeat

OTDR dst, src, r	dst: IR
OTDRB	src: IR
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	00111011_Rsnz_1010 0000_Rrrr_Rdnz_0000	11 + 10n
	SOTDR @Rd, @Rs, r	00111011_Rsnz_1011 0000_Rrrr_Rdnz_0000	11 + 10n
	OTDRB @Rd, @Rs, r	00111010_Rsnz_1010 0000_Rrrr_Rdnz_0000	11 + 10n
	SOTDRB @Rd, @Rs, r	00111010_Rsnz_1011 0000_Rrrr_Rdnz_0000	11 + 10n

Notes:

1. The Z8000 documentation lists the Z flag as undefined. In the Y8002 design this flag is unaffected.

Notes (continued):

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.

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.

5. Bits 3-0 of the second word of the opcode are actually a condition code. The result of this condition code check is ORed with the r = 0 check to determine if the instruction is complete. The bit combination 0000 is the encoding for “False”, making this a repeating instruction.

(S)OTIR

(Special) Output, Increment and Repeat

OTIR dst, src, r
OTIRB
SOTIR
SOTIRB

dst: IR
src: IR

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	00111011_Rsnz_0010 0000_Rrrr_Rdnz_0000	11 + 10n
	SOTIR @Rd, @Rs, r	00111011_Rsnz_0011 0000_Rrrr_Rdnz_0000	11 + 10n
	OTIRB @Rd, @Rs, r	00111010_Rsnz_0010 0000_Rrrr_Rdnz_0000	11 + 10n
	SOTIRB @Rd, @Rs, r	00111010_Rsnz_0011 0000_Rrrr_Rdnz_0000	11 + 10n

Notes:

1. The Z8000 documentation lists the Z flag as undefined. In the Y8002 design this flag is unaffected.

Notes (continued):

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.

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.

5. Bits 3-0 of the second word of the opcode are actually a condition code. The result of this condition code check is ORed with the r = 0 check to determine if the instruction is complete. The bit combination 0000 is the encoding for “False”, making this a repeating instruction.

(S)OUT

(Special) Output

OUT dst, src **OUTB**	dst: IR, DA src: R
SOUT dst, src **SOUTB**	dst: DA 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	00111111_Rdnz_Rsss	10
	OUTB @Rd, Rbs	00111110_Rdnz_Rbss	10
DA:	OUT port, Rst	00111011_Rsss_0110 port	12
	SOUT port, Rst	00111011_Rsss_0111 port	12
	OUTB port, Rbs	00111010_Rsss_0110 port	12
	SOUTB port, Rbs	00111010_Rsss_0111 port	12

Notes:

1. The Z8000 documentation lists the Z flag as undefined. In the Y8002 design this flag is unaffected.

2. Data is written to the I/O or Special I/O address space. Only the status code is different between OUT and SOUT.

3. I/O writes always have one automatic WAIT state. This is included in the Cycles number above.

(S)OUTD

(Special) Output and Decrement

OUTD dst, src, r	dst: IR
OUTDB	src: IR
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	00111011_Rsnz_1010 0000_Rrrr_Rdnz_1000	21
	SOUTD @Rd, @Rs, r	00111011_Rsnz_1011 0000_Rrrr_Rdnz_1000	21
	OUTDB @Rd, @Rs, r	00111010_Rsnz_1010 0000_Rrrr_Rdnz_1000	21
	SOUTDB @Rd, @Rs, r	00111010_Rsnz_1011 0000_Rrrr_Rdnz_1000	21

Notes:

1. The Z8000 documentation lists the Z flag as undefined. In the Y8002 design this flag is unaffected.

Notes (continued):

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.

4. Bits 3-0 of the second word of the opcode are actually a condition code. The result of this condition code check is ORed with the r = 0 check to determine if the instruction is complete. The bit combination 1000 is the encoding for “True”, making this a non-repeating instruction.

(S)OUTI

(Special) Output and Increment

OUTI dst, src, r	dst: IR
OUTIB	src: IR
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	00111011_Rsnz_0010 0000_Rrrr_Rdnz_1000	21
	SOUTI @Rd, @Rs, r	00111011_Rsnz_0011 0000_Rrrr_Rdnz_1000	21
	OUTIB @Rd, @Rs, r	00111010_Rsnz_0010 0000_Rrrr_Rdnz_1000	21
	SOUTIB @Rd, @Rs, r	00111010_Rsnz_0011 0000_Rrrr_Rdnz_1000	21

Notes:

1. The Z8000 documentation lists the Z flag as undefined. In the Y8002 design this flag is unaffected.

Notes (continued):

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.

4. Bits 3-0 of the second word of the opcode are actually a condition code. The result of this condition code check is ORed with the r = 0 check to determine if the instruction is complete. The bit combination 1000 is the encoding for “True”, making this a non-repeating instruction.

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 address	16
	POPL address, @Rs	01010101_Rsnz_0000 address	23
X:	POP addr(Rd), @Rs	01010111_Rsnz_Rdnz address	16
	POPL addr(Rd), @Rs	01010101_Rsnz_Rdnz address	23

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.

PUSH

Push

PUSH dst, src	dst: IR
PUSHL	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
IR:	PUSH @Rd, @Rs	00010011_Rdnz_Rsnz	13
	PUSHL @Rd, @Rs	00010001_Rdnz_Rsnz	20
DA:	PUSH @Rd, address	01010011_Rdnz_0000 address	14
	PUSHL @Rd, address	01010001_Rdnz_0000 address	21

Addressing Modes	Assembly Syntax	Encoding	Clocks
X:	PUSH @Rd, addr(Rs)	01010011_Rdnz_Rsnz address	14
	PUSHL @Rd, addr(Rs)	01010001_Rdnz_Rsnz address	21

Notes:

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

2. The Z8000 documentation states that the Rs and Rd registers must be separate and non-overlapping for PUSHL. This restriction is not present in the Y8002 design.

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	10100011_Rddd_bbbb	4
	RESB Rbd, #b	10100010_Rbdd_0bbb	4
IR:	RES @Rd, #b	00100011_Rdnz_bbbb	11
	RESB @Rd, #b	00100010_Rdnz_0bbb	11
DA:	RES address, #b	01100011_0000_bbbb address	13
	RESB address, #b	01100010_0000_0bbb address	13
X:	RES addr(Rd), #b	01100011_Rdnz_bbbb address	14
	RESB addr(Rd), #b	01100010_Rdnz_0bbb address	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	dst: R
RESB	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	00100011_0000_Rsss 0000_Rddd_0000_0000	10
	RESB Rbd, Rs	00100010_0000_Rsss 0000_Rbdd_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 Y8002 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	10/7

Notes:

1. The execution time is 7 clocks if cc is false and 10 clocks if cc is true.

RL

Rotate Left

RL dst, src	dst: R
RLB	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	10110011_Rddd_0000	6
	RL Rd, #2	10110011_Rddd_0010	7
	RLB Rbd, #1	10110010_Rbdd_0000	6
	RLB Rbd, #2	10110010_Rbdd_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	10110011_Rddd_1000	6
	RLC Rd, #2	10110011_Rddd_1010	7
	RLCB Rbd, #1	10110010_Rbdd_1000	6
	RLCB Rbd, #2	10110010_Rbdd_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	dst: R
RRCB	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	10110011_Rddd_0100	6
	RR Rd, #2	10110011_Rddd_0110	7
	RRB Rbd, #1	10110010_Rbdd_0100	6
	RRB Rbd, #2	10110010_Rbdd_0110	7

RRC

Rotate Right through Carry

RRC dst, src	dst: R
RRCB	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	10110011_Rddd_1100	6
	RRC Rd, #2	10110011_Rddd_1110	7
	RRCB Rbd, #1	10110010_Rbdd_1100	6
	RRCB Rbd, #2	10110010_Rbdd_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	dst: R
SBCB	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	10110111_Rsss_Rddd	5
	SBCB Rbd, Rbs	10110110_Rbss_Rbdd	5

SC

System Call

SC src src: IM

Operation:

```
SP <= SP - 2
@SP <= PC
SP <= SP - 2
@SP <= FCW
SP <= SP - 2
@SP <= inst
FCW <= @{PSAP, 0x0C}
PC <= @{PSAP, 0x0E}
```

Flags:

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

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

Notes:

1. 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.

2. Version 1 of the Y8002 uses the System Call entry in the Program Status Area for all Unimplemented and Reserved opcodes. All of these opcodes can be distinguished from an actual System Call by inspecting the opcode word pushed onto the system stack. Refer to the appropriate Appendix for a list of all Unimplemented and Reserved opcodes.

SDA

Shift Dynamic Arithmetic

SDA dst, src	dst: R
SDAB	src: R
SDAL	

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	10110011_Rddd_1011 0000_Rsss_0000_0000	15 + 3n
	SDAB Rbd, Rs	10110010_Rbdd_1011 0000_Rsss_0000_0000	15 + 3n
	SDAL RRd, Rs	10110011_RRdd_1111 0000_Rsss_0000_0000	15 + 3n

Notes:

1. The 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 Y8002 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. However, the entire 16-bit value in the src register is used as the starting count for the shift, so values outside of these ranges will still execute.

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	dst: R
SDLB	src: R
SDLL	

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	10110011_Rddd_0011 0000_Rsss_0000_0000	15 + 3n
	SDLB Rbd, Rs	10110010_Rbdd_0011 0000_Rsss_0000_0000	15 + 3n
	SDLL RRd, Rs	10110011_RRdd_0111 0000_Rsss_0000_0000	15 + 3n

Notes:

1. The Z8000 documentation lists the V flag as undefined. In fact it signals arithmetic overflow, even though this is a logical operation. The Y8002 design matches this behavior.

1. The 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 Y8002 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. However, the entire 16-bit value in the src register is used as the starting count for the shift, so values outside of these ranges will still execute.

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	10100101_Rddd_bbbb	4
	SETB Rbd, #b	10100100_Rbdd_0bbb	4
IR:	SET @Rd, #b	00100101_Rdnz_bbbb	11
	SETB @Rd, #b	00100100_Rdnz_0bbb	11
DA:	SET address, #b	01100101_0000_bbbb address	13
	SETB address, #b	01100100_0000_0bbb address	13
X:	SET addr(Rd), #b	01100101_Rdnz_bbbb address	14
	SETB addr(Rd), #b	01100100_Rdnz_0bbb address	14

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	dst: R
SETB	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	00100101_0000_Rsss 0000_Rddd_0000_0000	10
	SETB Rbd, Rs	00100100_0000_Rsss 0000_Rbdd_0000_0000	10

Notes:

1. The Z8000 microprocessor restricts the source register to be one of R0 - R7 for SETB. This restriction does not apply to the Y8002 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	dst: R
SLAB	src: IM
SLAL	

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	
	SLAB Rbd, #b	10110010_Rbdd_1001 0 | b	
	SLAL RRd, #b	10110011_RRdd_1101 b	

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. However, the entire 16-bit value (Word or Long) or 8-bit value (Byte) in the second word of the instruction is used as the starting count for the shift, so values outside of these ranges will still execute.

2. Only bits 7-0 of the second word of the opcode are used for SLAB. All other bits in the second word of the opcode are ignored in this case.

SLL

Shift Left Logical

SLL dst, src	dst: R
SLLB	src: IM
SLLL	

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_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. The Z8000 documentation lists the V flag as undefined. In fact it signals arithmetic overflow, even though this is a logical operation. The Y8002 design matches this behavior.

Notes: (continued)

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. However, the entire 16-bit value (Word or Long) or 8-bit value (Byte) in the second word of the instruction is used as the starting count for the shift, so values outside of these ranges will still execute.

3. Only bits 7-0 of the second word of the opcode are used for SLLB. All other bits in the second word of the opcode are ignored in this case.

SRA

Shift Right Arithmetic

SRA dst, src	dst: R
SRAB	src: IM
SRAL	

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 16 for SRAB and from 1 to 32 for SRAL. However, the entire 16-bit value (for Word or Long) or 8-bit value (for Byte) in the second word of the instruction is used as the starting count for the shift, so values outside of these ranges will still execute. Note that *b* is negative in the opcode.

2. Only bits 7-0 of the second word of the opcode are used for SRAB. All other bits in the second word of the opcode are ignored in this case.

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. The Z8000 documentation lists the V flag as undefined. In fact it signals arithmetic overflow, even though this is a logical operation. The Y8002 design matches this behavior.

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. However, the entire 16-bit value (Word or Long) or 8-bit value (Byte) in the second word of the instruction is used as the starting count for the shift, so values outside of these ranges will still execute. Note that *b* is negative in the opcode.

Notes (continued):

3. Only bits 7-0 of the second word of the opcode are used for SRLB. All other bits in the second word of the opcode are ignored in this case.

SUB

Subtract

SUB dst, src	dst: R
SUBB	src: R, IM, IR, DA, X
SUBL	

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).
H: Cleared if arithmetic carry from bit 3; set otherwise, indicating borrow (SUBB);
Unaffected (SUB).

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SUB Rd, Rs	10000011_Rsss_Rddd	4
	SUBB Rbd, Rbs	10000010_Rbss_Rbdd	4
	SUBL RRd, RRs	10010010_RRss_RRdd	8
IM:	SUB Rd, #data	00000011_0000_Rddd #data	7
	SUBB Rbd, #data	00000010_0000_Rbdd #data | #data	7
	SUBL RRd, #data	00010010_0000_RRdd #data (high) #data (low)	14

Addressing Modes	**Assembly Syntax**	**Encoding**	**Clocks**
IR:	SUB Rd, @Rs	00000011_Rsnz_Rddd	7
	SUBB Rbd, @Rs	00000010_Rsnz_Rbdd	7
	SUBL RRd, @Rs	00010010_Rsnz_RRdd	14
DA:	SUB Rd, address	01000011_0000_Rddd address	9
	SUBB Rbd, address	01000010_0000_Rbdd address	9
	SUBL RRd, address	01010010_0000_RRdd address	15
X:	SUB Rd, addr(Rs)	01000011_Rsnz_Rddd address	10
	SUBB Rbd, addr(Rs)	01000010_Rsnz_Rbdd address	10
	SUBL RRd, addr(Rs)	01010010_Rsnz_RRdd address	16

TCC

Test Condition Code

TCC cc, dst dst: R
TCCB

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	10101111_Rddd_cccc	5
	TCCB cc, Rbd	10101110_Rbdd_cccc	5

TEST

Test

TEST dst   dst: R, IR, DA, X
TESTB
TESTL

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	10001101_Rddd_0100	7
	TESTB Rbd	10001100_Rbdd_0100	7
	TESTL RRd	10011100_RRdd_1000	9
IR:	TEST @Rd	00001101_Rdnz_0100	8
	TESTB @Rd	00001100_Rdnz_0100	8
	TESTL @Rd	00011100_Rdnz_1000	13
DA:	TEST address	01001101_0000_0100 address	11
	TESTB address	01001100_0000_0100 address	11
	TESTL address	01011100_0000_1000 address	16

Addressing Modes	Assembly Syntax	Encoding	Clocks
X:	TEST addr(Rd)	01001101_Rdnz_0100 address	12
	TESTB addr(Rd)	01001100_Rdnz_0100 address	12
	TESTL addr(Rd)	01011100_Rddd_1000 address	17

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]
Rs <= 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:	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 of the second word of the opcode are ignored.

3. There is no obvious reason why bits 3-0 of the second word of the opcode must be 1110, and the Y8002 actually ignores these four bits.

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.

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 of the second word of the opcode are ignored.

3. There is no obvious reason why bits 3-0 of the second word of the opcode must be 1110, and the Y8002 actually ignores these four bits.

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.

TSET

Test and Set

TEST dst
TESTB

dst: R, IR, DA, X

Operation: Word: S <= dst[15], dst <= 0xFFFFh
Byte: S <= dst[7], dst <= 0xFFh

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	10001101_Rddd_0110	7
	TSETB Rbd	10001100_Rbdd_0110	7
IR:	TSET @Rd	00001101_Rdnz_0110	11
	TSETB @Rd	00001100_Rdnz_0110	11
DA:	TSET address	01001101_0000_0110 address	14
	TSETB address	01001100_0000_0110 address	14
X:	TSET addr(Rd)	01001101_Rdnz_0110 address	15
	TSETB addr(Rd)	01001100_Rdnz_0110 address	15

Notes:

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

2. The Z8004 microprocessor actually signals a special status code (0xF) for the read and write of a TSET instruction. The Y8002 design does not do this, instead matching the operation of the Z8002 microprocessor and signalling either Data or Stack memory address space.

XOR

Logical Exclusive-OR

XOR dst, src	dst: R
XORB	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	10001001_Rsss_Rddd	4
	XORB Rbd, Rbs	10001000_Rbss_Rbdd	4
IM:	XOR Rd, #data	00001001_0000_Rddd #data	7
	XORB Rbd, #data	00001000_0000_Rddd #data | #data	7
IR:	XOR Rd, @Rs	00001001_Rsnz_Rddd	7
	XORB Rbd, @Rs	00001000_Rsnz_Rbdd	7
DA:	XOR Rd, address	01001001_0000_Rddd address	9
	XORB Rbd, address	01001000_0000_Rbdd address	9

Addressing Modes	Assembly Syntax	Encoding	Clocks
X:	XOR Rd, addr(Rs)	01001001_Rsnz_Rddd address	10
	XORB Rbd, addr(Rs)	01001000_Rsnz_Rbdd address	10

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_0xxx_00xx xxxx_Rddd_xxxx_nnnn	11 + 4n
IR:	LD @Rd, EPU, #n	00001111_Rdnz_11xx xxxxxxxx_xxxx_nnnn	11 + 3n
DA:	LD address, EPU, #n	01001111_0000_11xx xxxxxxxx_xxxx_nnnn address	14 + 3n
X:	LD addr(Rd), EPU, #n	01001111_Rdnz_11xx xxxxxxxx_xxxx_nnnn address	15 + 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.

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.

Notes (continued):

3. Bits 15-12 and 7-4 of the second word of the opcode 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_0xxx_10xx xxxx_Rsss_xxxx_nnnn	11 + 4n
IR:	LD EPU, @Rs, #n	00001111_Rsnz_01xx xxxxxxxx_xxxx_nnnn	11 + 3n
DA:	LD EPU, address, #n	01001111_0000_01xx xxxxxxxx_xxxx_nnnn address	14 + 3n
X:	LD EPU, addr(Rs), #n	01001111_Rsnz_01xx xxxxxxxx_xxxx_nnnn address	15 + 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.

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.

Notes (continued):

3. Bits 15-12 and 7-4 of the second word of the opcode 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_0000_xxxx_0000	15

Notes:

2. 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.

3. Bits 15-12 and 7-4 of the second word of the opcode 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_Rsss_xxxx_nnnn	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 15-12 and 7-4 of the second word of the opcode 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 #n	10001110_xxxx_01xx xxxx_xxxx_xxxx_nnnn	11 + 4n

Notes:

2. 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.

3. Bits 15-4 of the second word of the opcode are ignored by the CPU but may be used by the EPU.

Chapter 6

External Interface and Timing

This chapter presents external interface signals and timing for the Y8002 design. The user has the option of selecting either the normal multiplexed implementation of the design or a special non-multiplexed implementation. The table below shows the external interface signals available for both versions.

Name	Description	Direction	Active State	Non-Muxed	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 = Word	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
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 = System	X	X
NMIB	Non-maskable Interrupt Request	Input	Low	X	X
NVIB	Non-vectored Interrupt Request	Input	Low	X	X
R_WB	Read/Write	Output	Low = Write	X	X
RESETF	Device Reset	Input	Low	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	Input	Low	X	X

Note that 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 Y8002 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 Y8002 implementation. These options are described more fully in the chapter that covers the actual Verilog code for the design. The Y8002 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 Y8002 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 only 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).

- 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 type of interrupt request 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 may 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 and starting PC from 0x0002 and 0x0004, respectively, in the 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.

- 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). The bus transaction type is encoded as shown below:

ST[3:0]	Bus Transaction Type
0000	Internal Operation
0001	Refresh
0010	Standard I/O
0011	Special I.//O
0100	Reserved
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	Reserved

- 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 Y8002 design itself.

- 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 may be enabled or disabled by the processor.

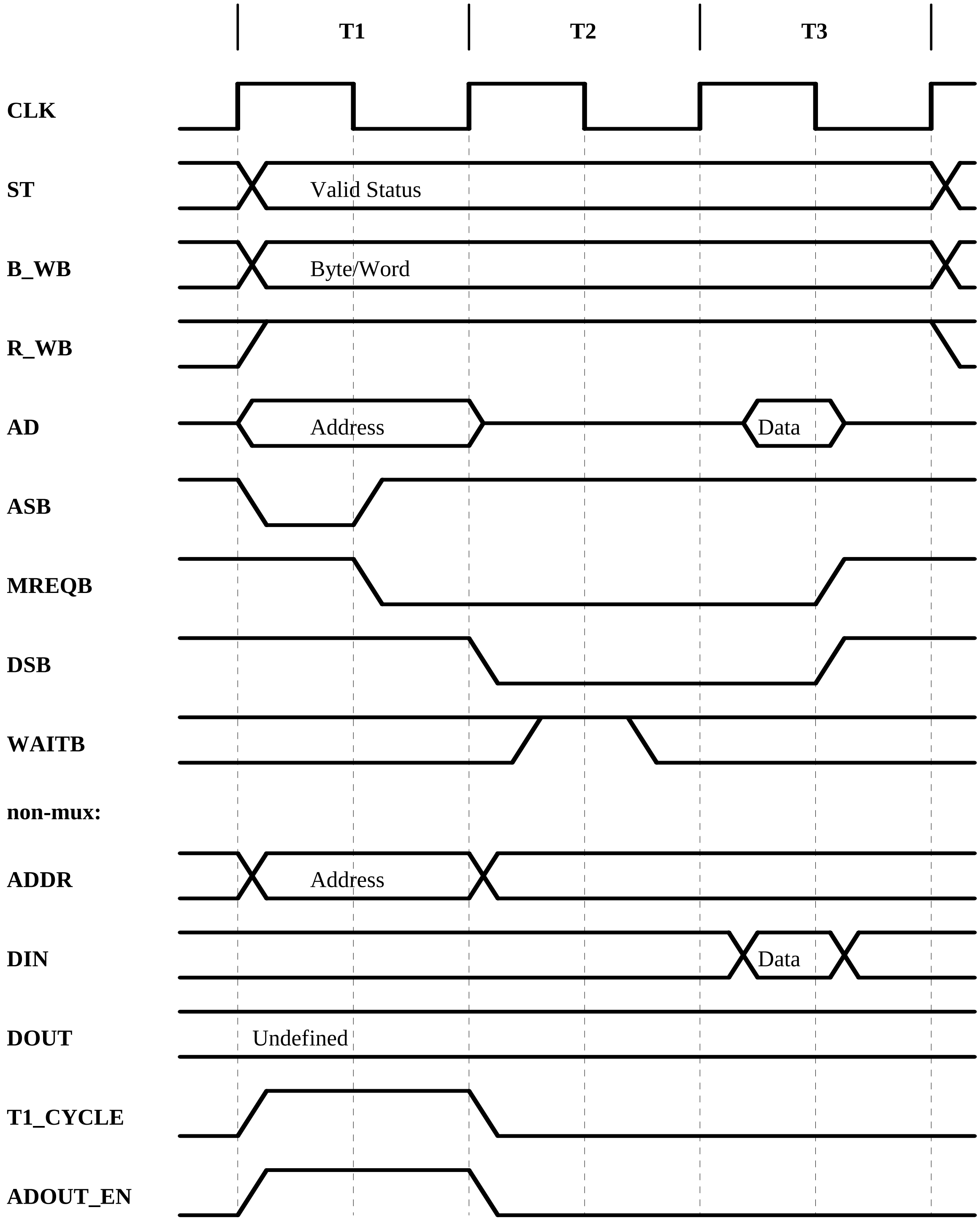
- WAITB (Wait Request). This signal is asserted by devices on the system bus to notify the processor that additional time is needed to complete the current bus transaction.

This section has detailed the entire set of interface signals, without regard to whether or not a particular signal has been implemented in a particular version of the design. For a list of implemented versus unimplemented signals, refer to the appropriate Appendix.

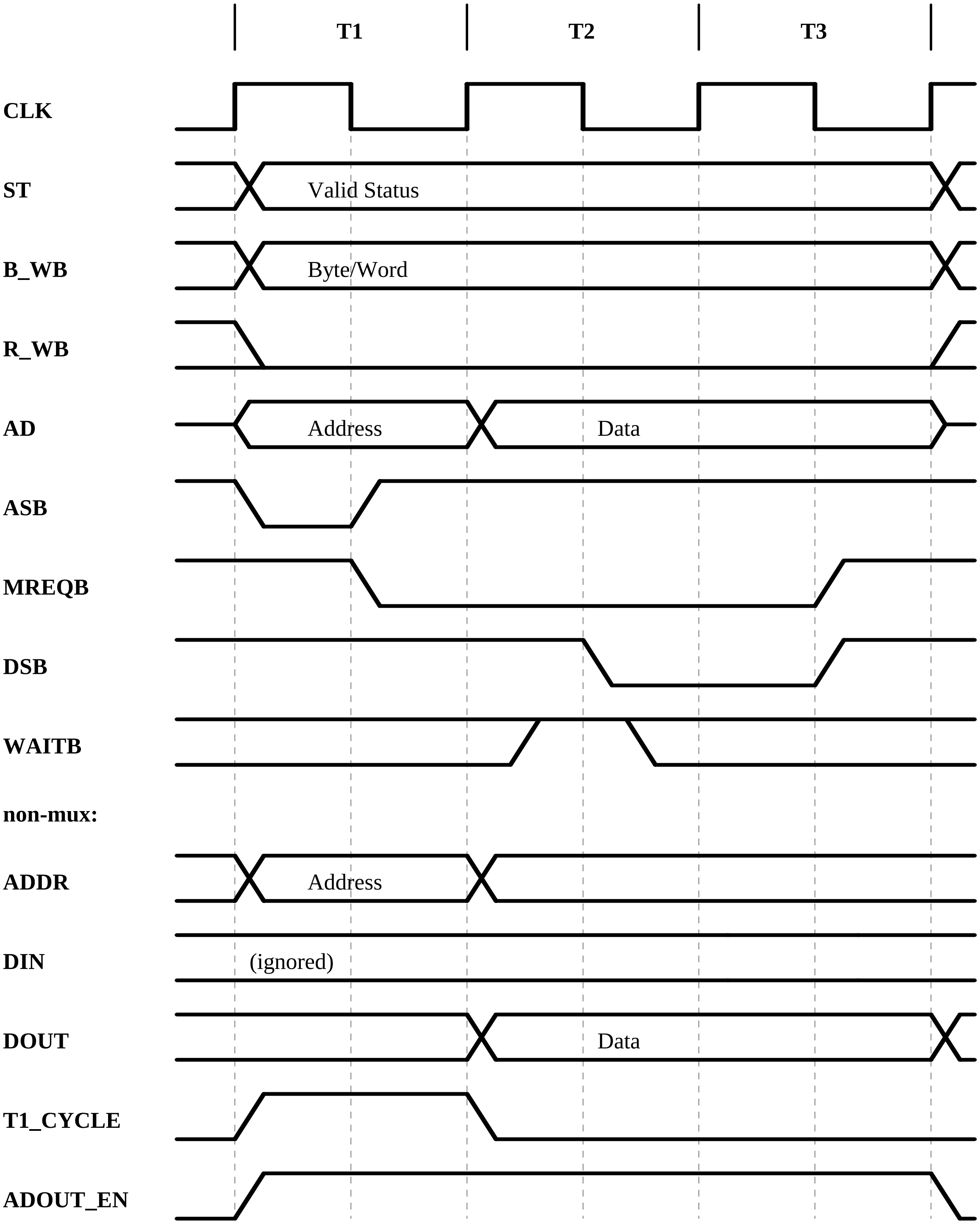
Note that all Y8002 signals come either directly from the output of a flip-flop, or through one or two gate delays from the output of a flip-flop. All outputs that come from logic are designed to be glitch-free. The lack of substantial clock-to-signal delays for outputs is different from the original Z8002 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. Refer to the appropriate Appendix for an instruction-by-instruction table of machine cycle lengths.

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

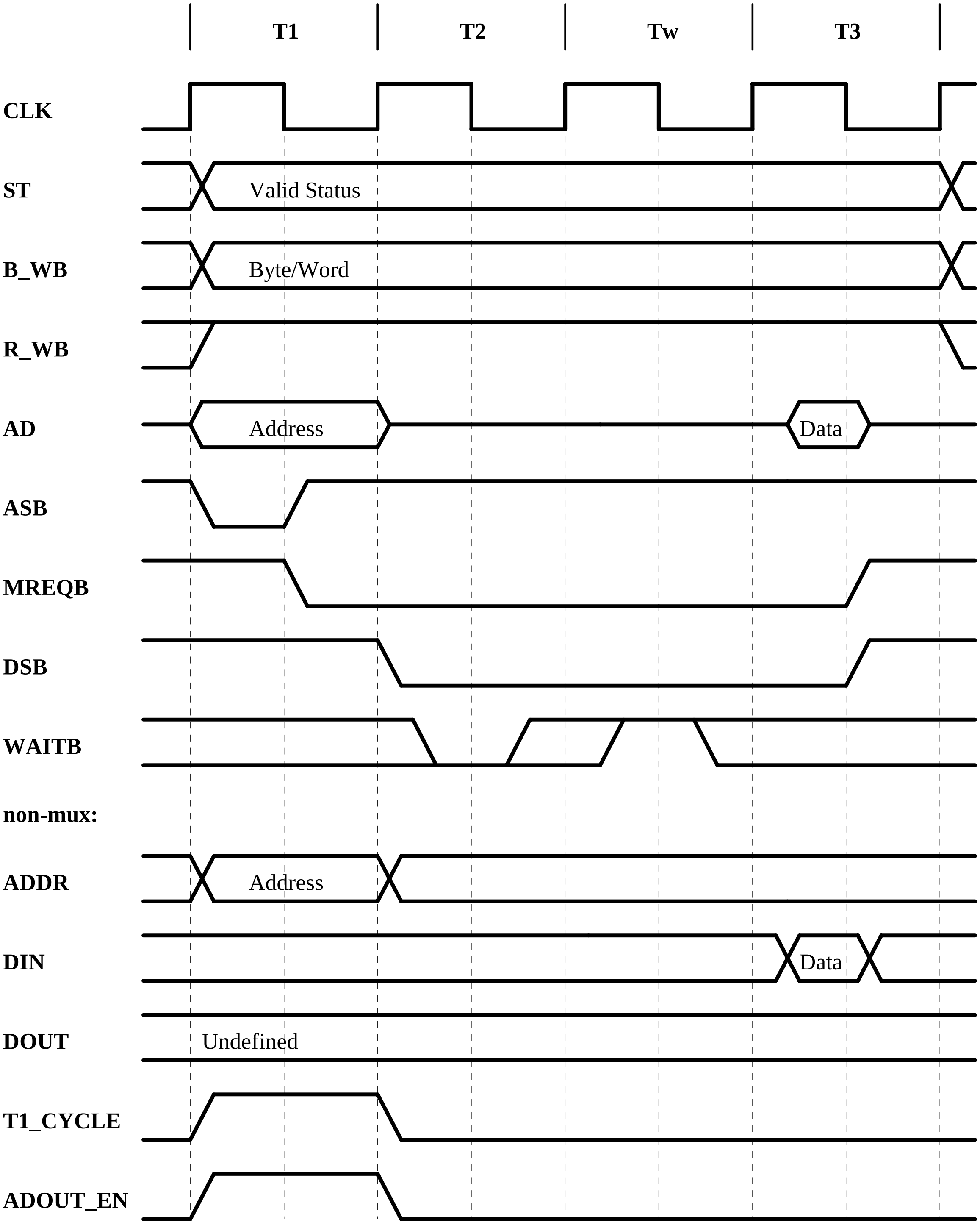


The figure below shows a memory write bus transaction, without wait states:

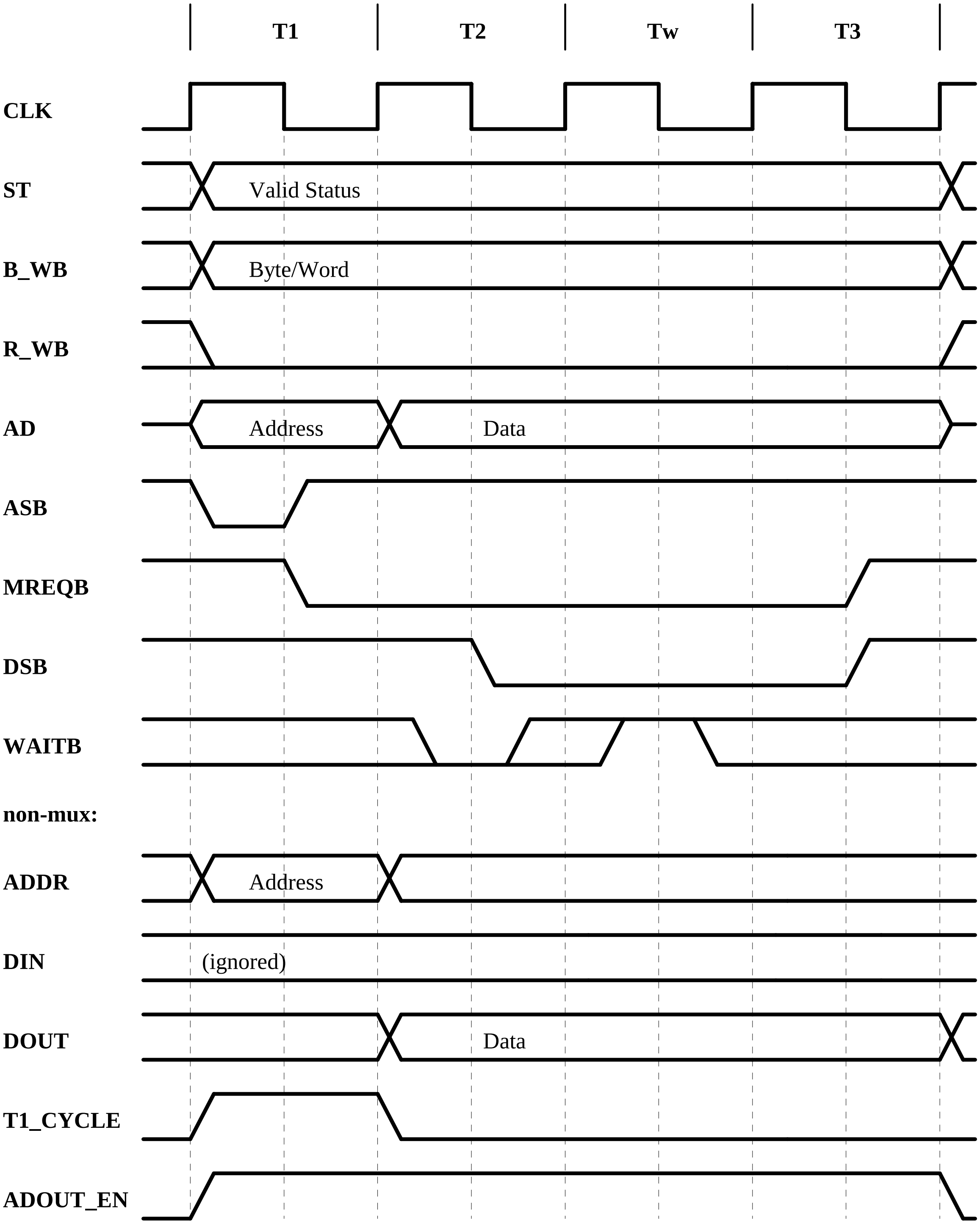


Note that for memory write cycles the DSB signal is delayed by one-half clock cycle to guarantee that the data bus is valid prior to the leading edge of DSB.

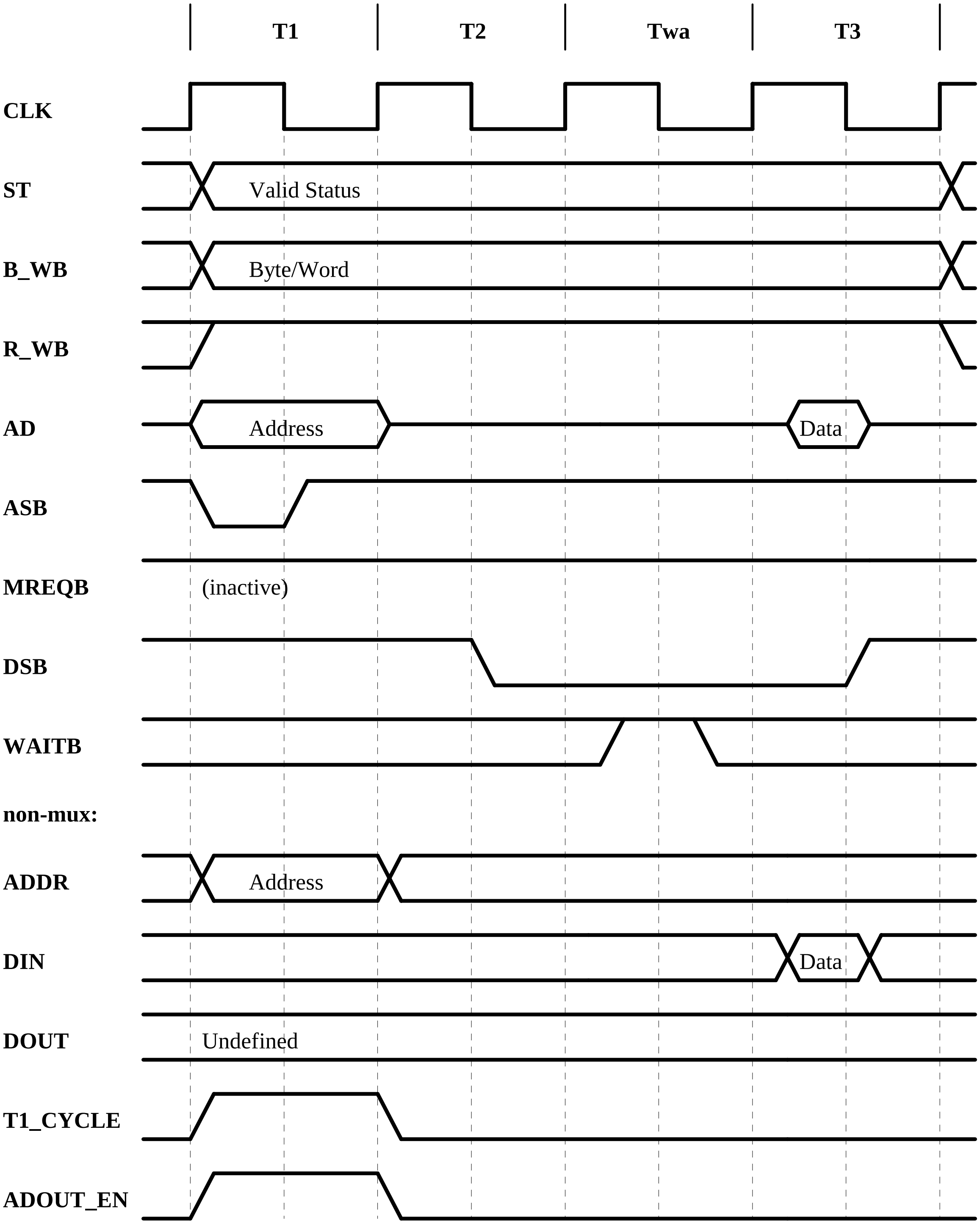
The figure below shows a memory read bus transaction, with one wait state:



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

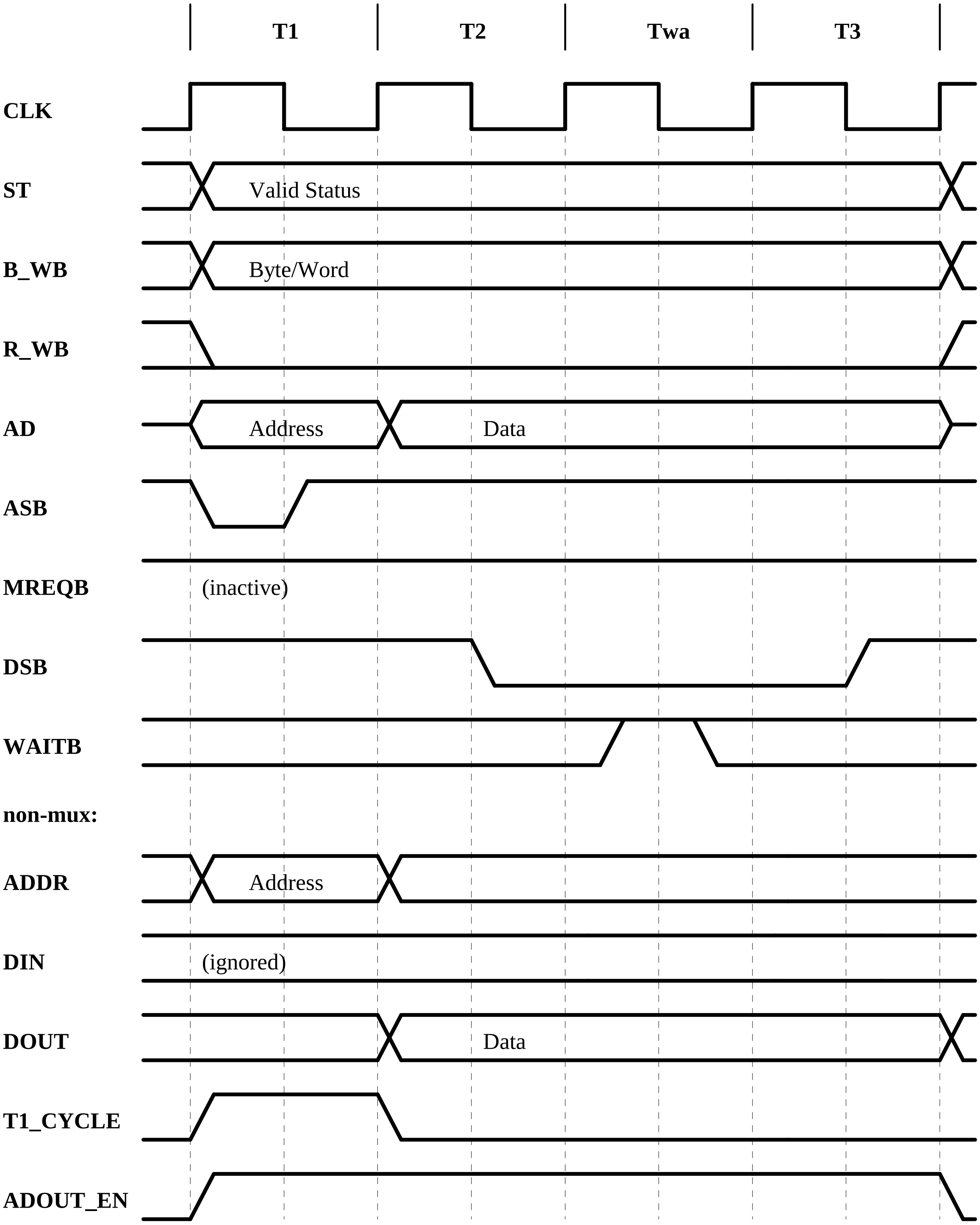


The figure below shows an I/O read bus transaction, with only the automatic wait state:

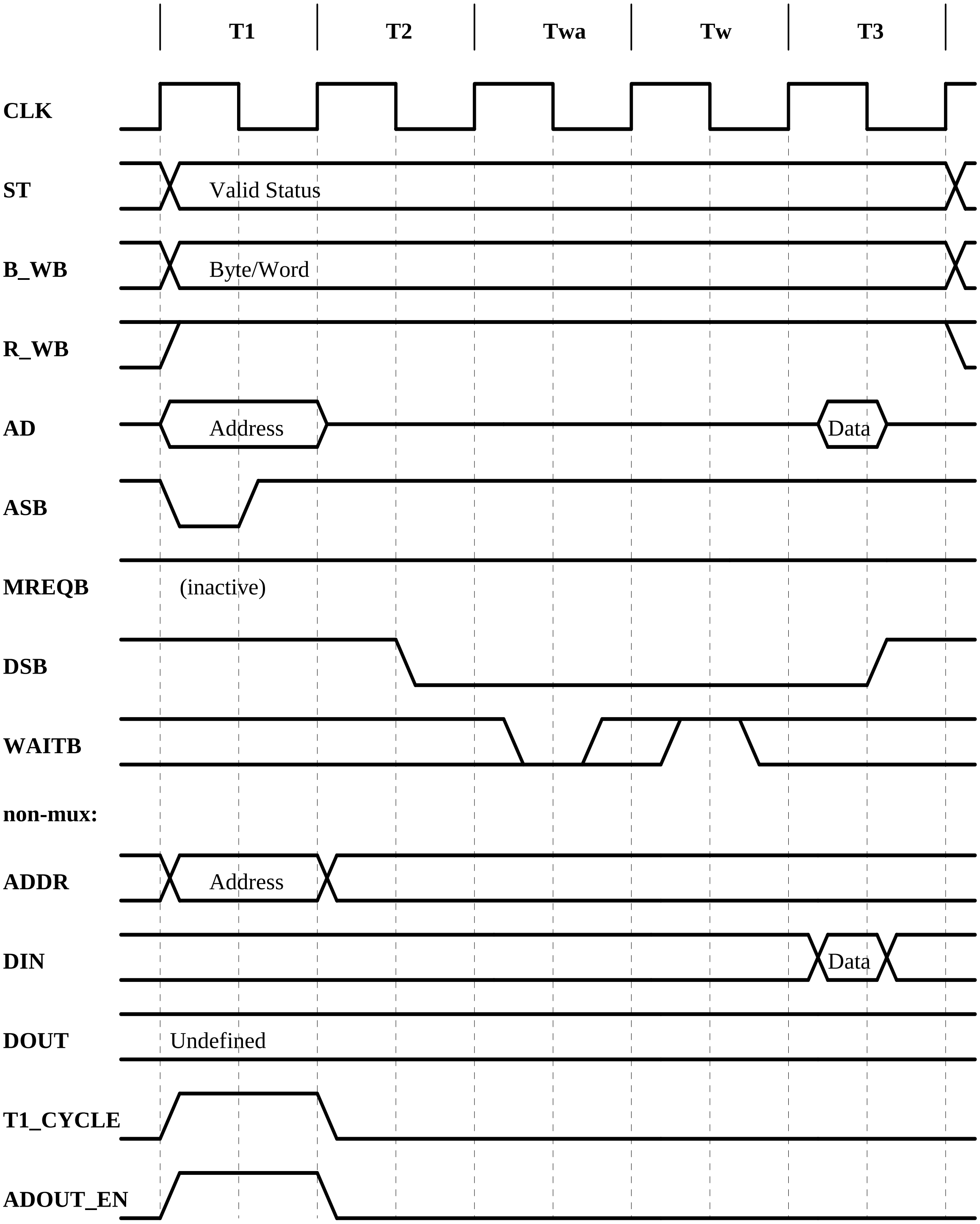


Note that the timing for the falling edge of Data Strobe for an I/O bus transaction is different form the corresponding timing for a Memory Read bus transaction.

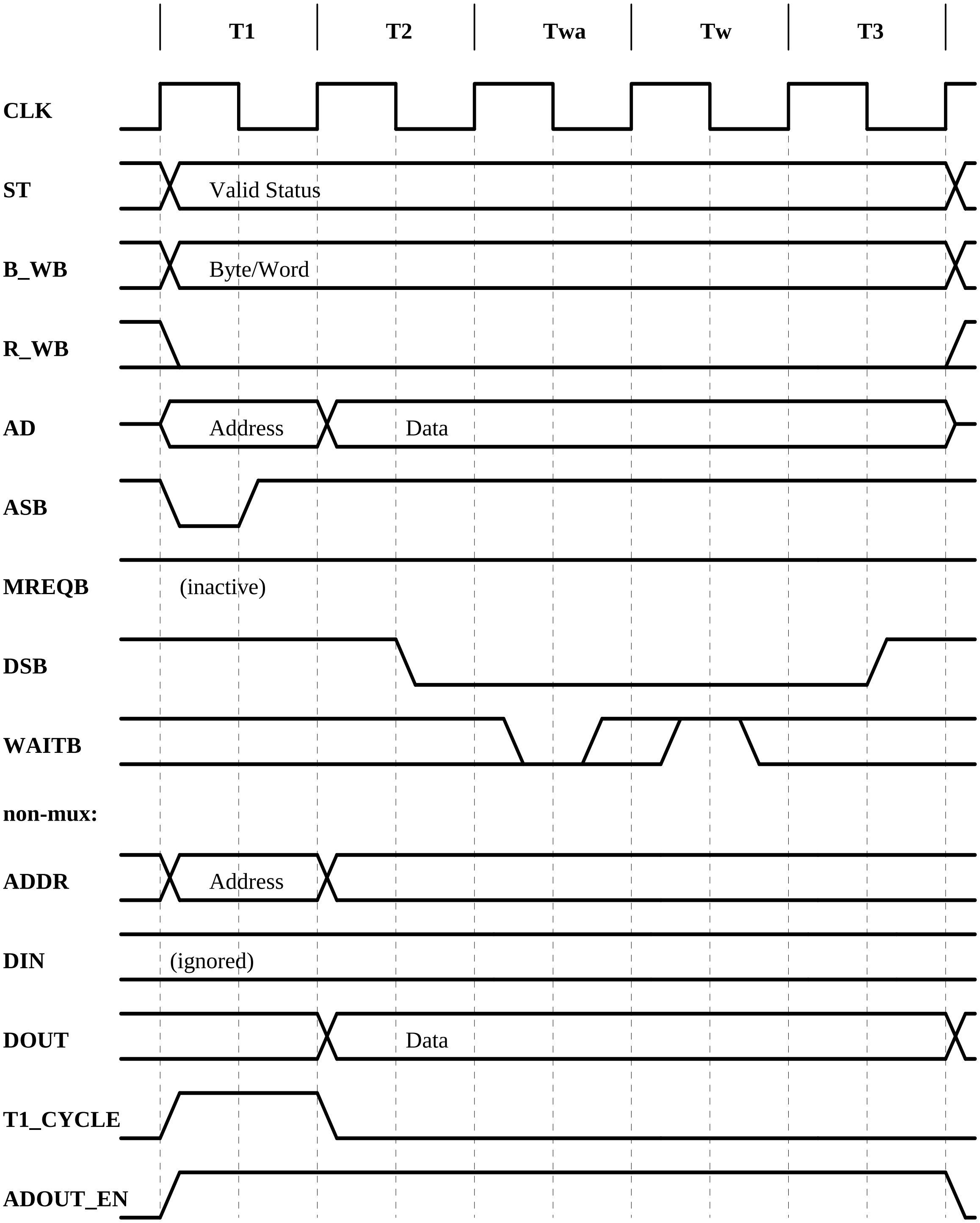
The figure below shows an I/O write bus transaction, with only the automatic wait state:



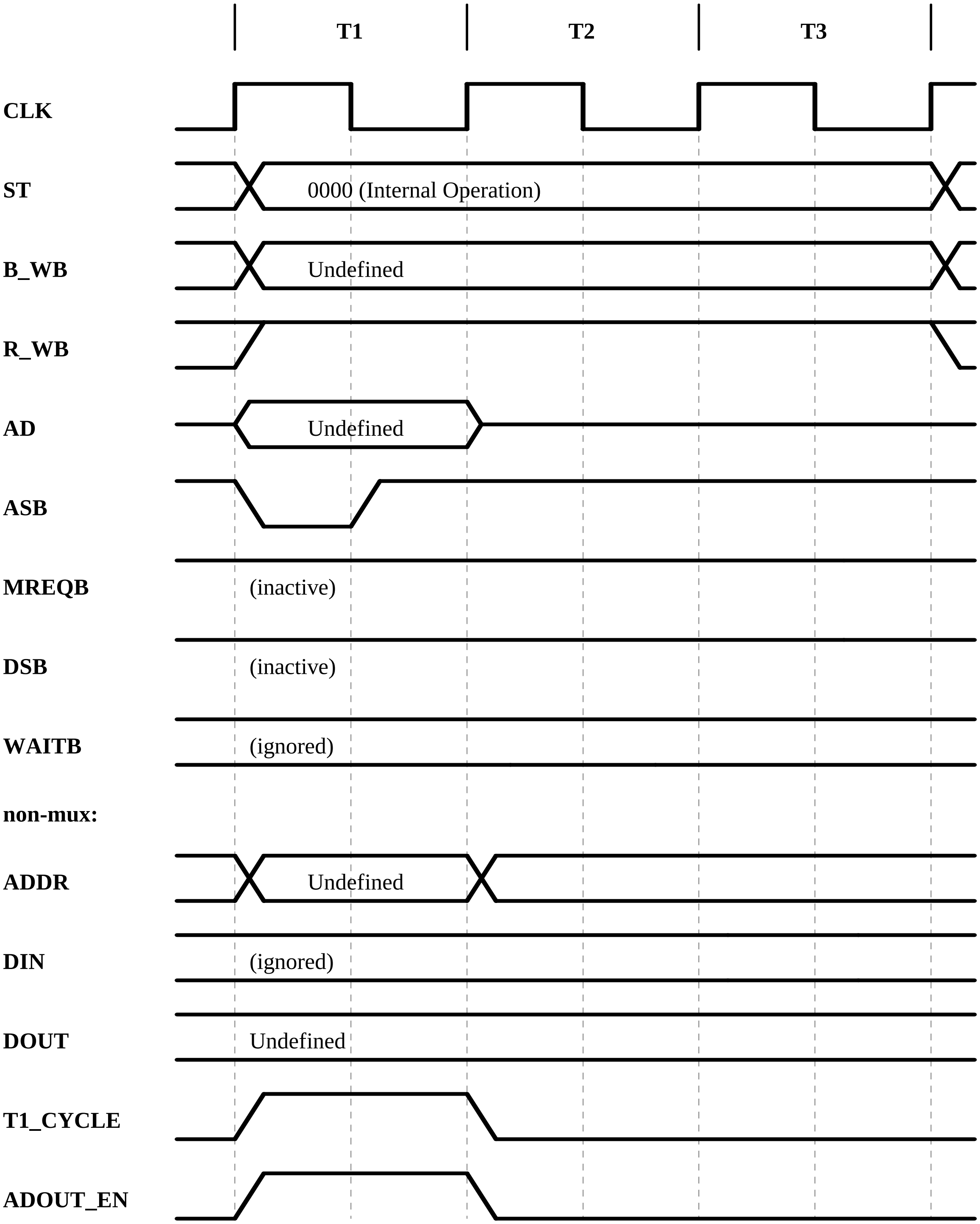
The figure below shows an I/O read bus transaction, with one additional wait state:



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

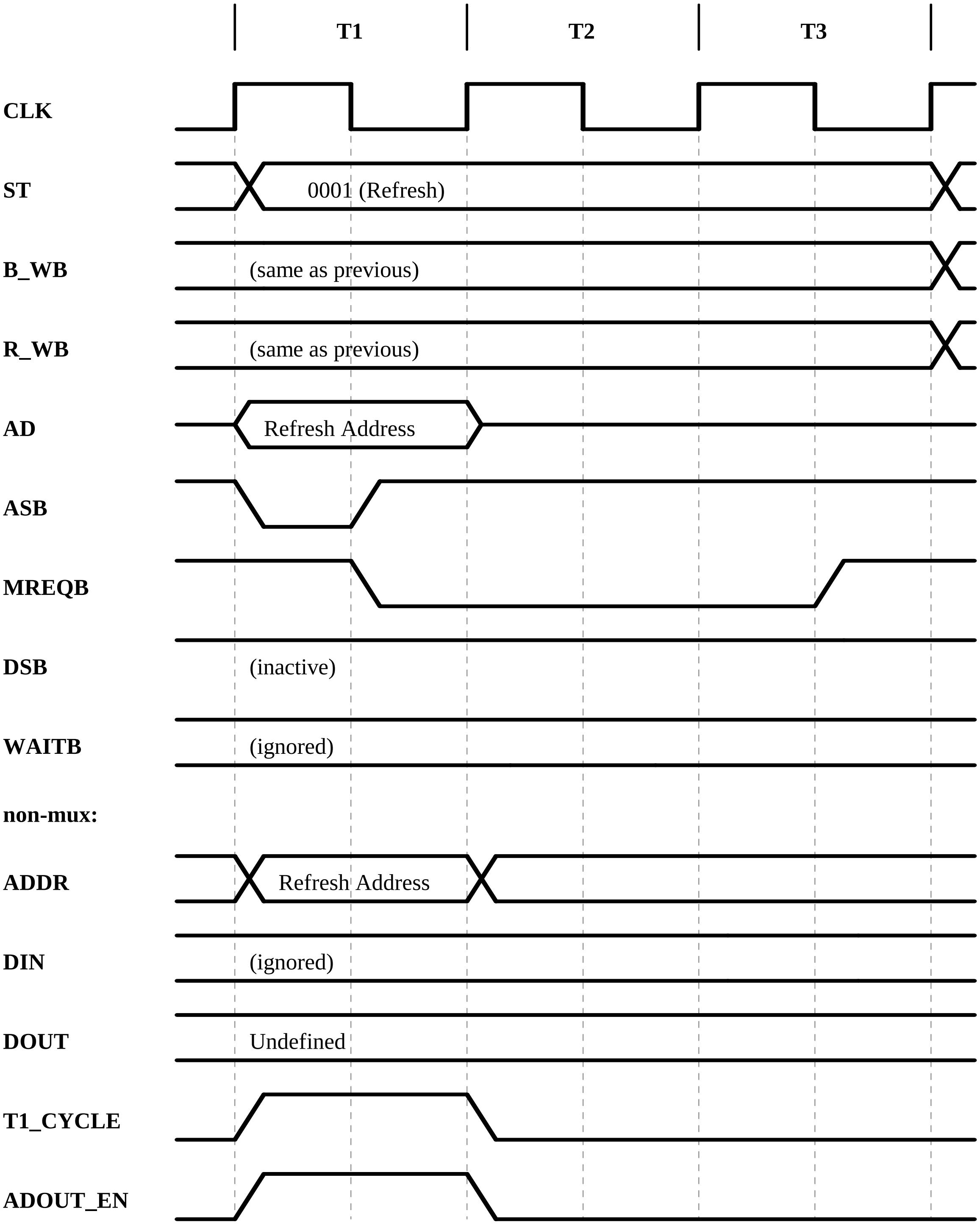


The figure below shows an internal operation bus transaction:



Note that the Wait Request input us not sampled for this type of bus transaction.

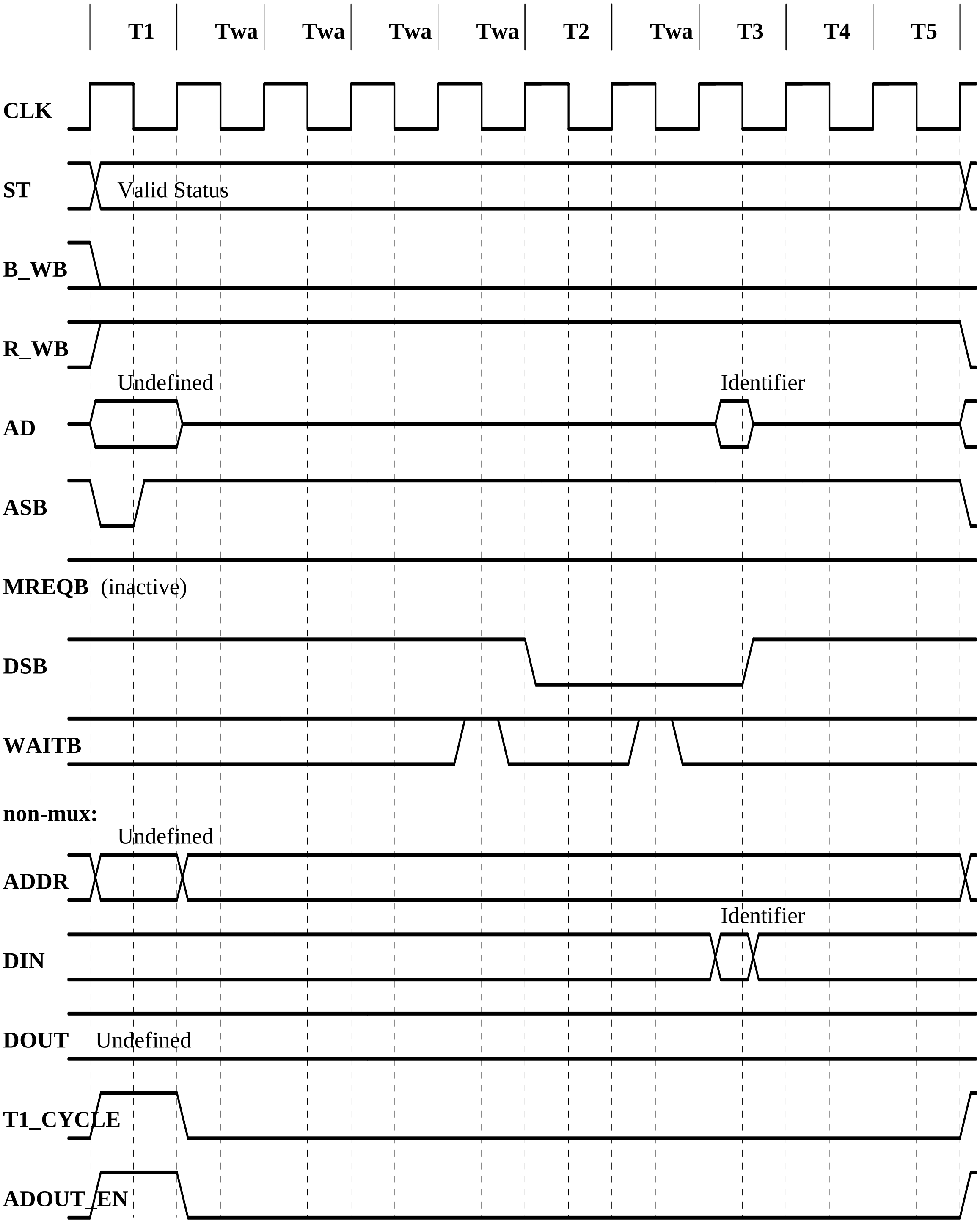
The figure below shows a refresh bus transaction:



Note that the Wait Request input us not sampled for this type of bus transaction.

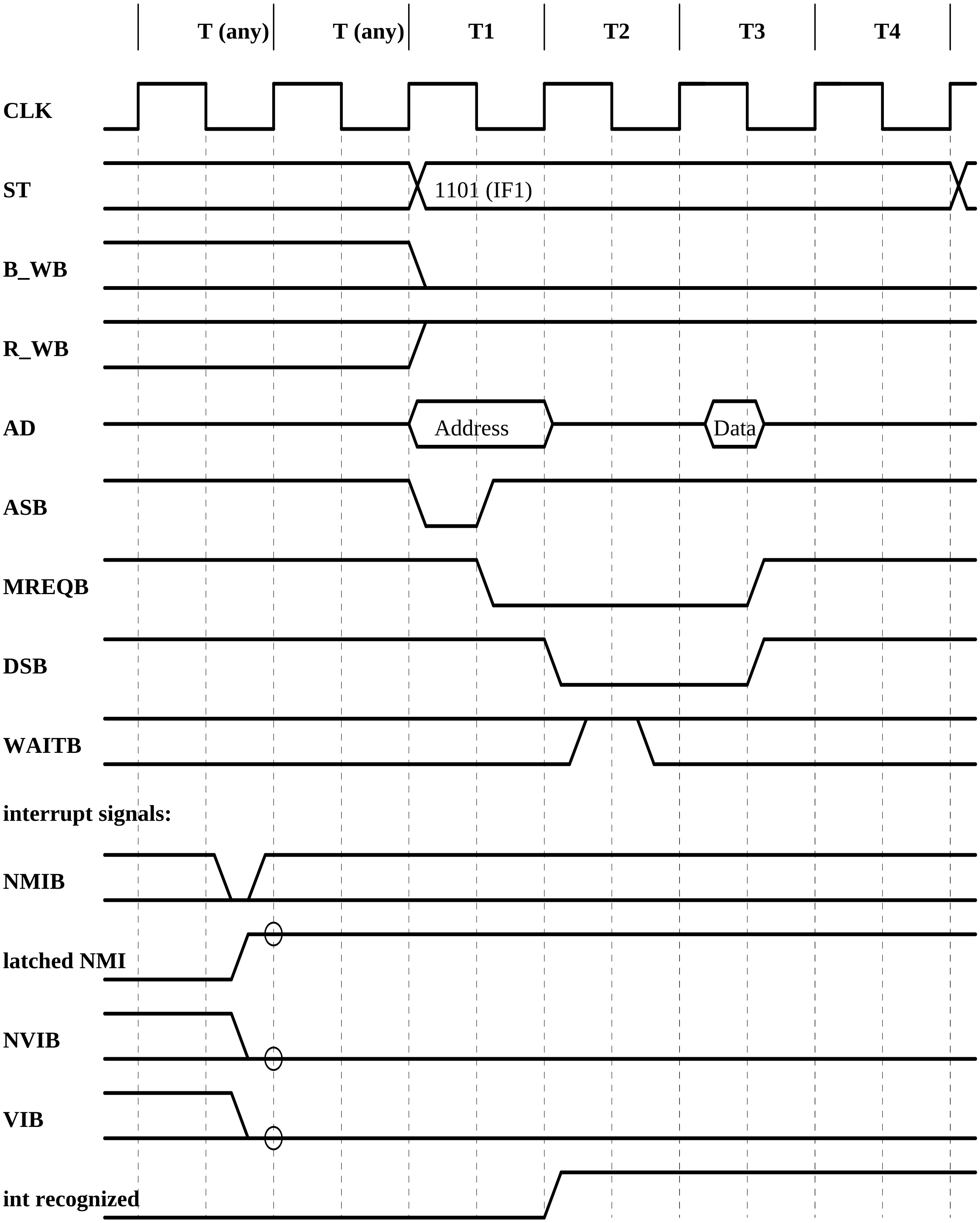
This type of bus transaction is also issued in response to a Stop request.

The figure below shows an interrupt acknowledge bus transaction:



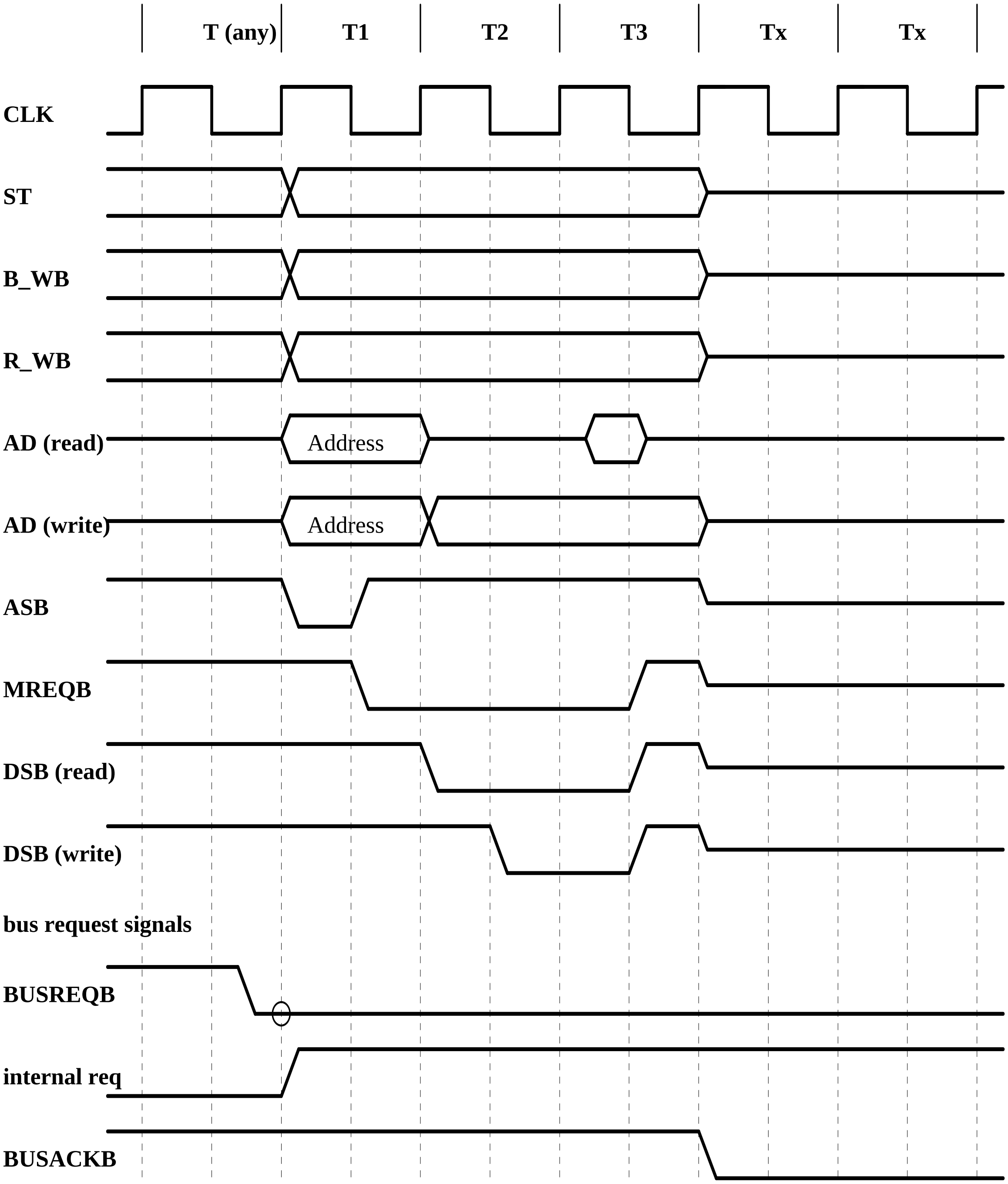
Note that the Wait Request input is sampled twice during this bus transaction, once before the Data Strobe is activated and once while Data Strobe is active. Memory write bus transactions follow immediately, to save the program status.

The figure below shows how interrupts are sampled:



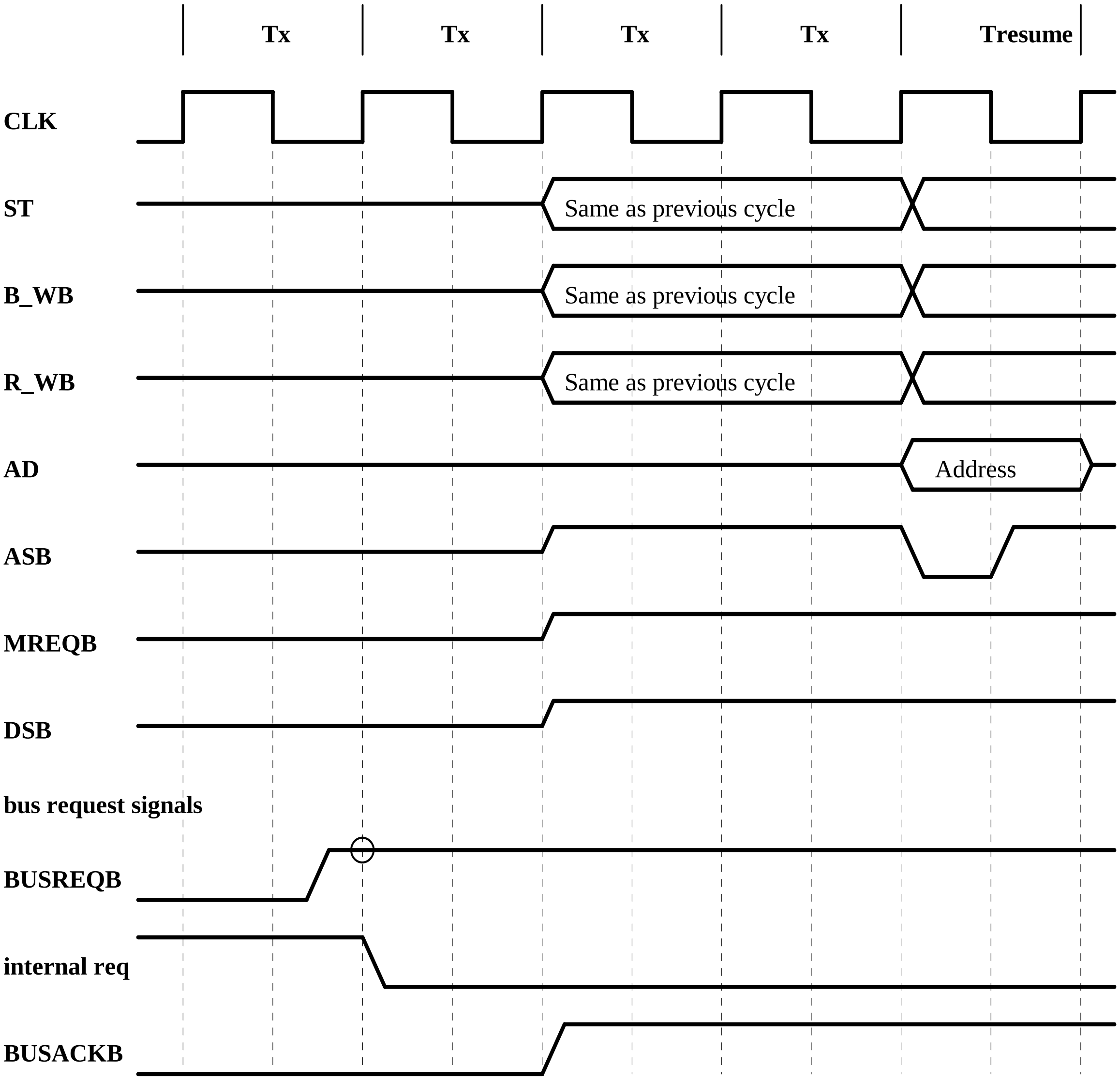
Note that the aborted IF1 transaction is always 4 clock cycles (plus Wait states) long in this design. The Z8000 documentation indicates that this aborted IF1 transaction may be from 3 to 7 cycles (plus Wait states) long. An interrupt acknowledge transaction follows immediately after this aborted IF1 transaction.

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.

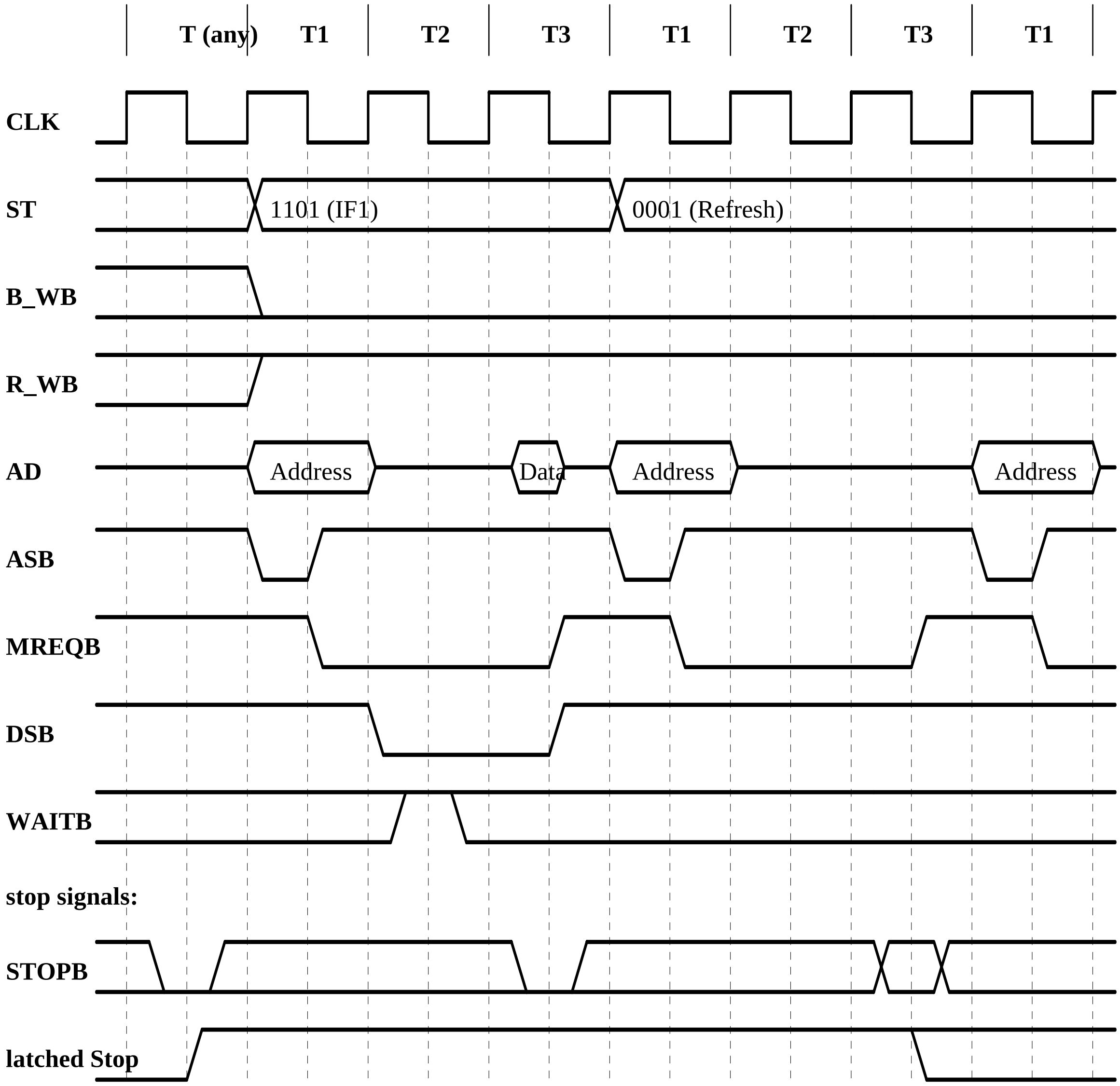
The figure below shows the sampling of Bus Request and the exit from the Bus Acknowledge state:



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.

Note that 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.

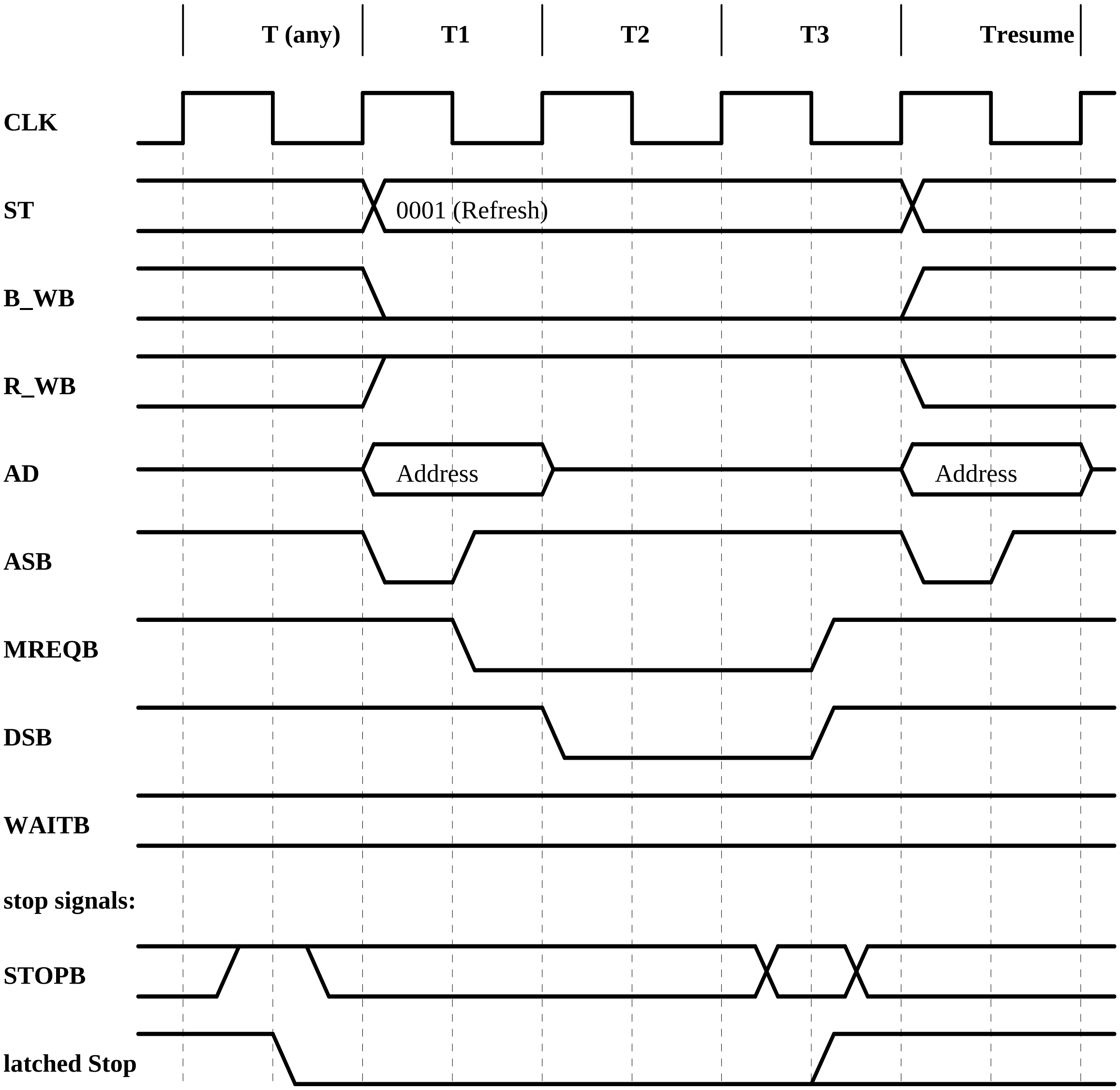
The figure below shows how the Stop Request is sampled:



Note that the Stop Request input is sampled prior to an IF1 bus transaction or a Stop bus transaction only. The IF1 bus transaction that follows the sampling of Stop Request active is always three clock cycles long, even if the IF1 bus transaction for the fetched instruction would normally be four or more clock cycles long. In this case the extra IF1 clock cycles follow the inserted Stop bus transactions.

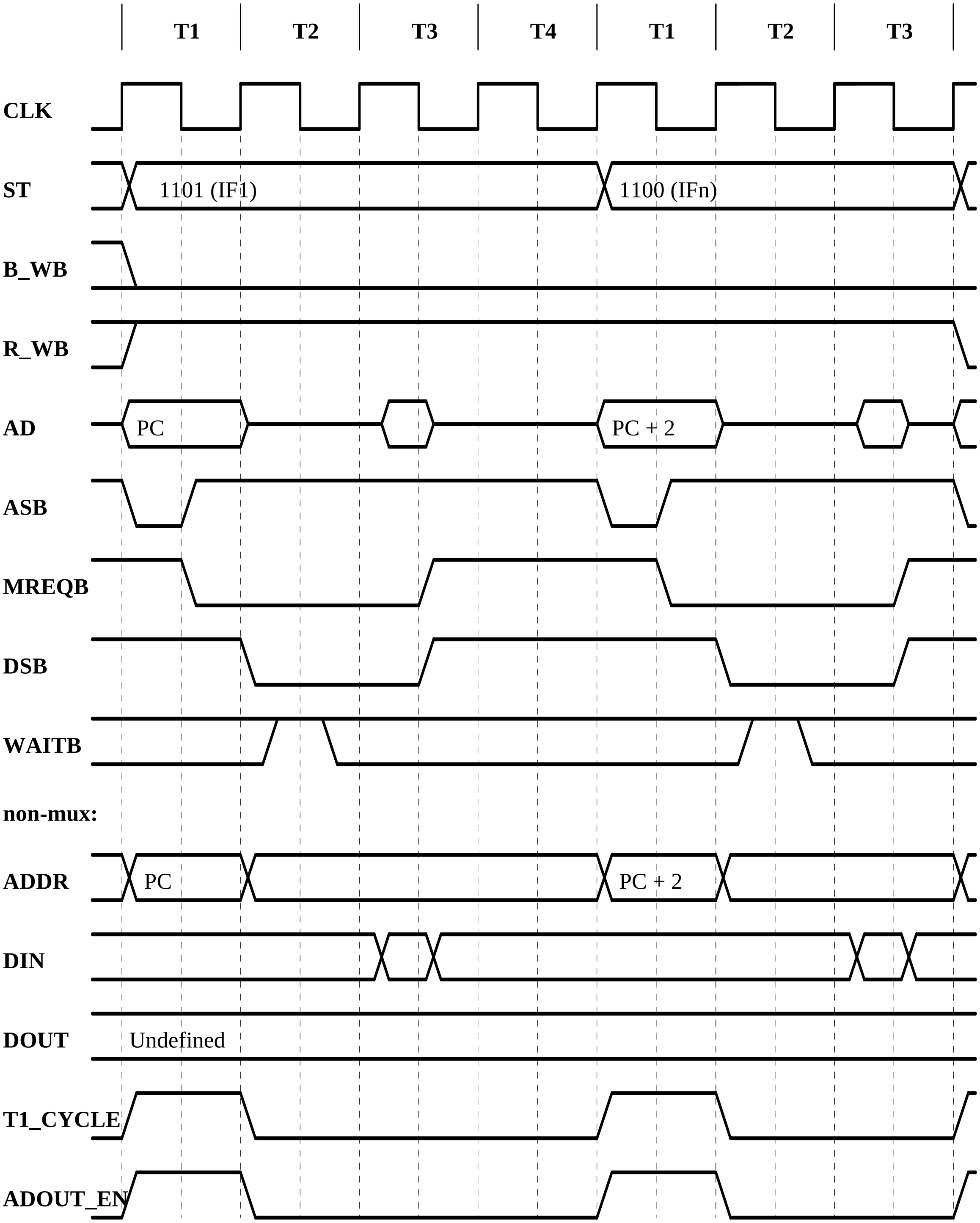
The diagram above shows two inserted Stop transactions.

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



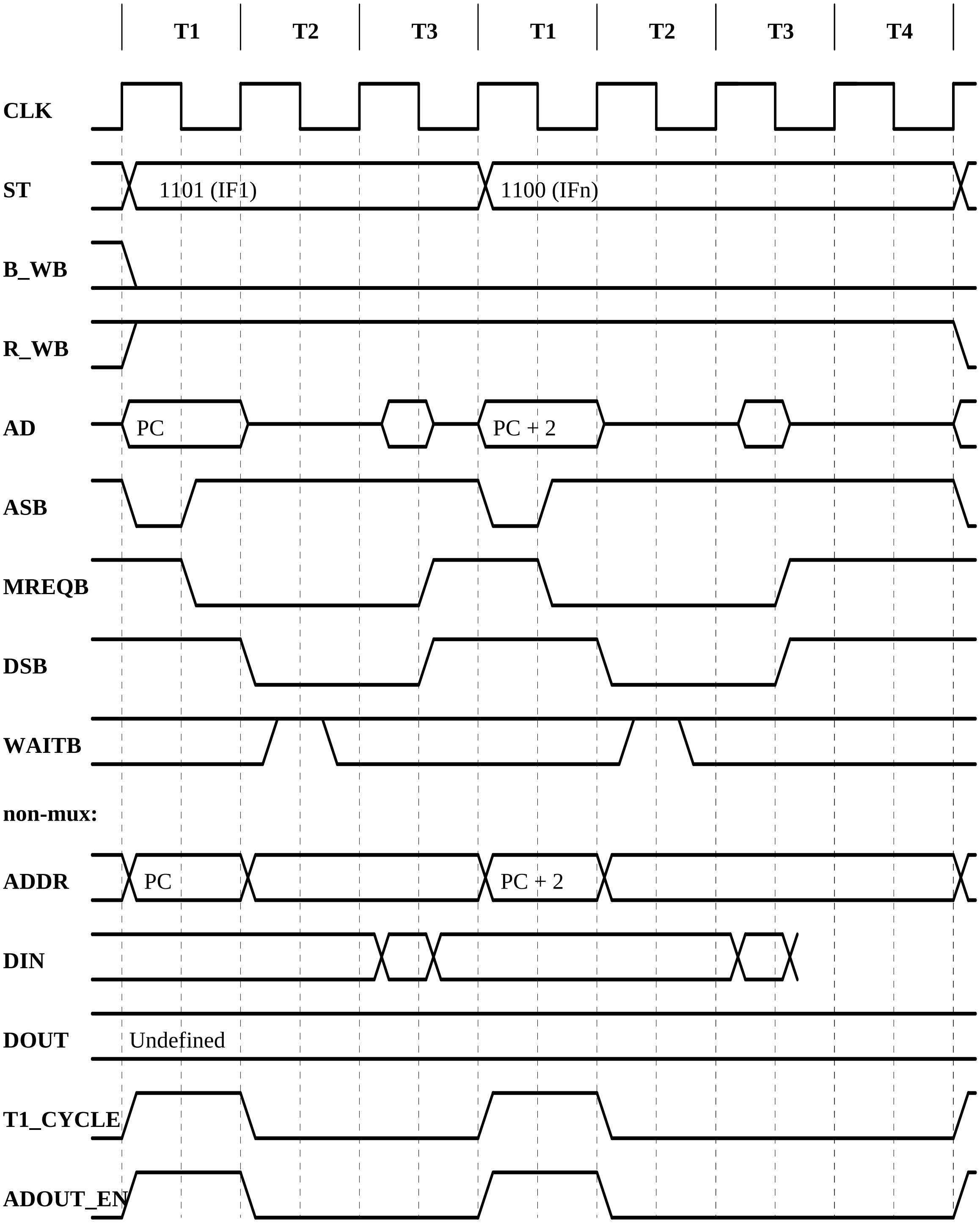
The behavior in response to the Stop Request input is identical to the behavior in the case of a Refresh request, except for the address placed on the Address/Data bus. Stop bus transactions use the PC for the address, while the refresh address is used for actual memory refresh bus transactions.

The figure below shows the first of three cases of the Trap timing:



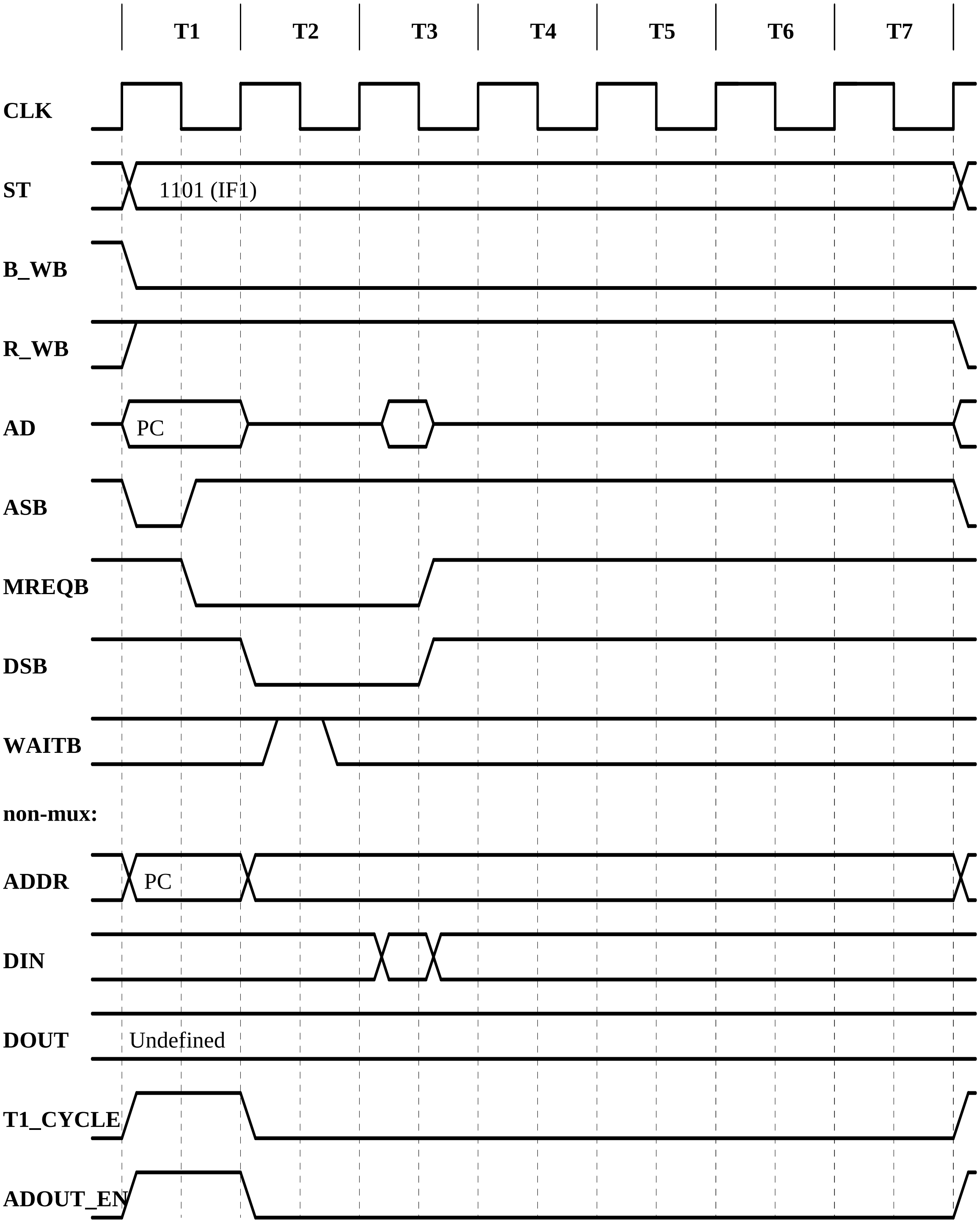
The instruction fetched during the IF1 bus transaction is pushed as the identifier for this trap. The word fetched during the IFn bus transaction is ignored. The Program Counter does not increment beyond PC + 2, and this is the value pushed on the stack.

The figure below shows the second of three cases of the Trap timing:



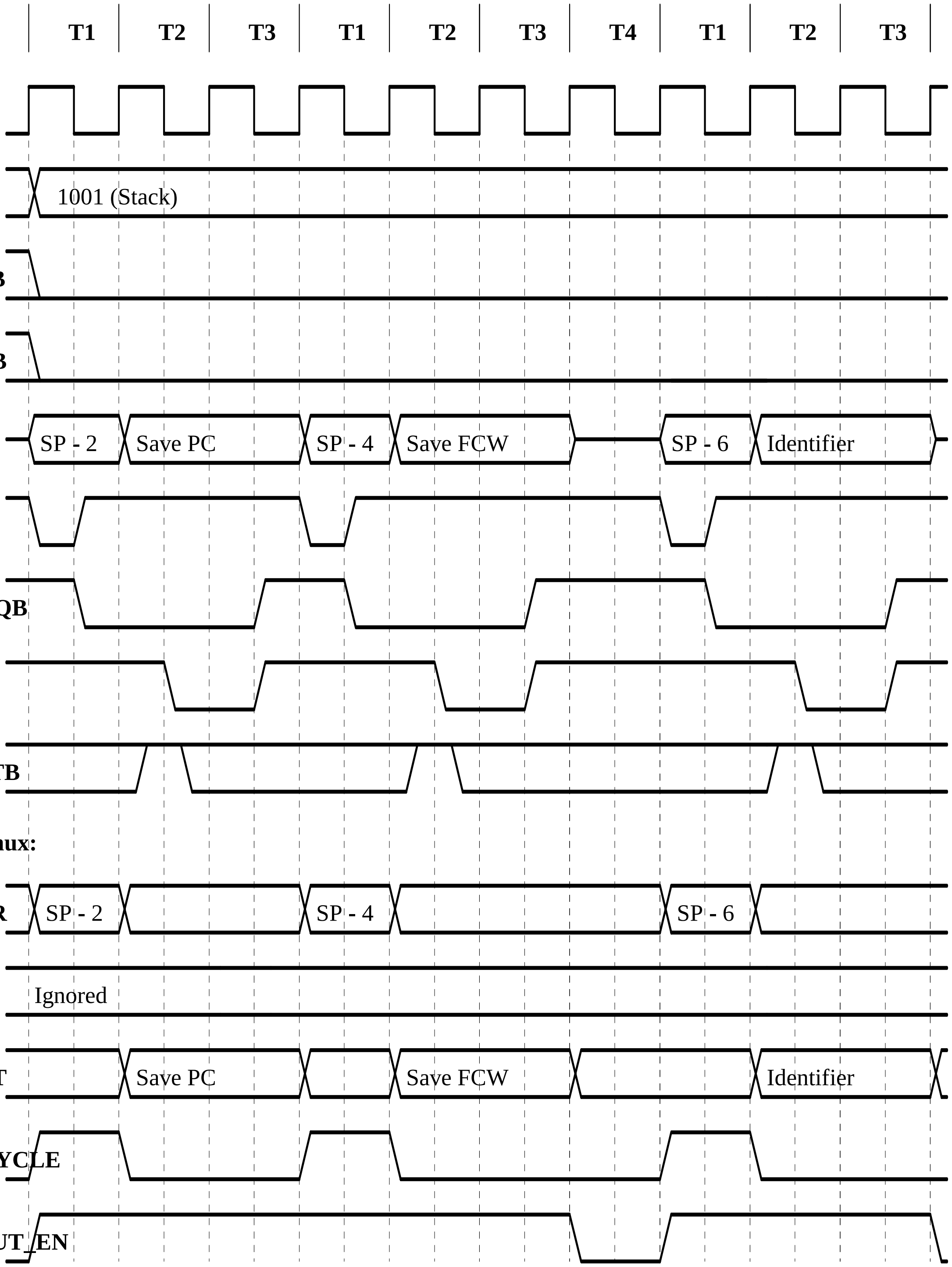
This case occurs when bits 15:14 of the fetched instruction are 01. The instruction fetched during the IF1 bus transaction is pushed as the identifier for this trap and the word fetched during the IFn bus transaction is ignored. The Program Counter does not increment beyond PC + 2, and this is the value pushed on the stack.

The figure below shows the third of three cases of the Trap timing:



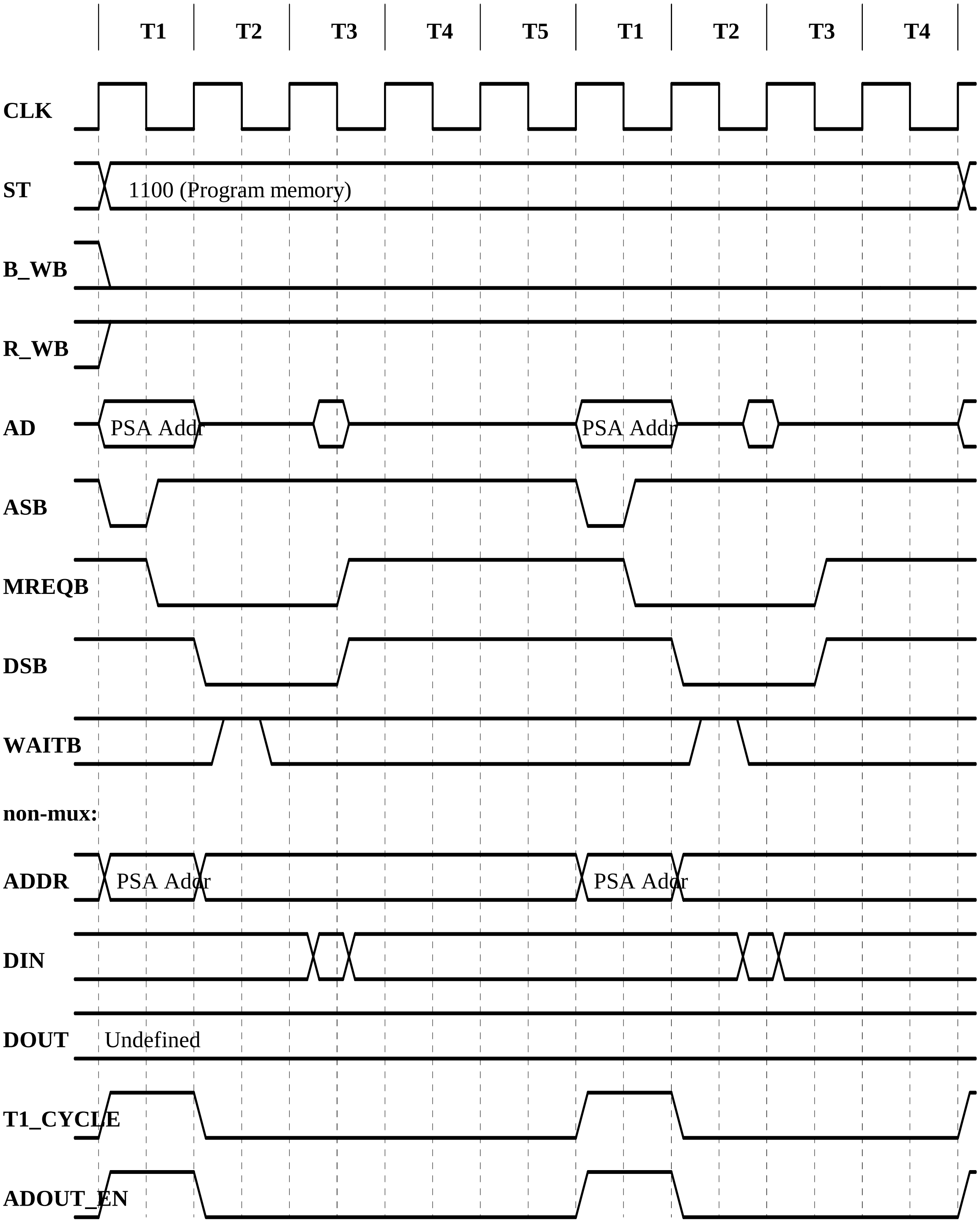
This is the case used by the System Call instruction, plus a number of illegal instructions. The timing for all of the different cases of a trap are identical, and only the signals during cycles 4-7 of the sequences are different.

The figure below shows the status saving sequence common to both interrupts and traps:



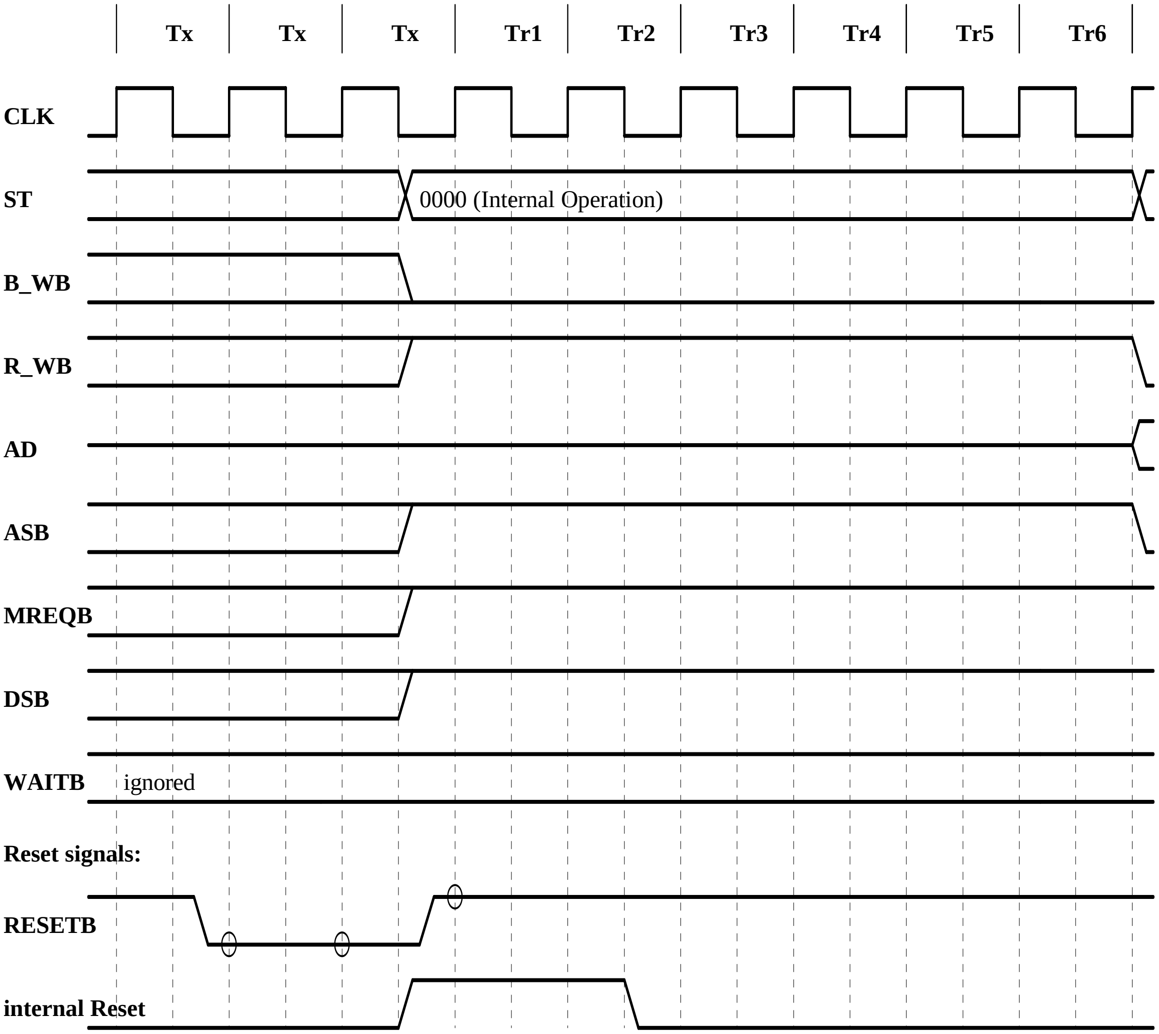
This status-saving sequence immediately follows the interrupt acknowledge or trap sequence. The stack pointer used is the System SP.

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



This sequence follows the reset sequence directly. This sequence follows the trap or interrupt sequence after seven clock cycles of no external operation.

The figure below shows the Reset timing and reset sequence:



Note that the Y8002 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.

Chapter 7

Interrupts and Traps

The Z8000 architecture supports three different types of interrupts and four different types of trap. One of the traps defined in the architecture (segmentation) is not present in either the Z8002 processor or in the Y8002 design and will not be discussed. In addition, the Y8002 design extends the operation of one of the traps. Each interrupt and trap type will be discussed below. The detailed timing for interrupts and traps was shown in the previous chapter.

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 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 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 PC value pushed in various cases is shown below:

Exception	**Pushed PC value**
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/Unimplemented Instruction Trap	Address of word following offending instruction word (next instruction for one-word instructions, second word of instruction for two-word instructions)
Any Interrupt	Address of next instruction (current instruction if block instruction that has not completed is interrupted)

The start of the Program Status Area is pointed to by the PSAP register in the processor. 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 table below:

The Program Status Area should be relocated from its reset location because the Reset FCW and PC are fetched from locations 0x0002 and 0x0004 respectively, which interferes with the Program Status Area entry for the Extended Instruction Trap when using the first 256-byte page as the Program Status Area. The two reserved locations at the start of the Program Status Area hint that the Z8000 architecture intended them to be used for Reset, but that there was an error in the original Z8000 implementation. The Y8002 matches the behavior of the Z8002 design.

Offset from PSAP	Contents	Interrupt/Trap type
0x000		Reserved
0x002		Reserved
0x004	FCW	Extended Instruction Trap
0x006	PC	
0x008	FCW	Privileged Instruction Trap
0x00A	PC	
0x00C	FCW	SC and Unimplemented Instruction Trap
0x00E	PC	
0x010		Not used
0x012		
0x014	FCW	Non-maskable Interrupt
0x016	PC	
0x018	FCW	Non-vectored Interrupt
0x01A	PC	
0x01C	FCW	Vectored Interrupts
0x01E	PC (vector = 0x00)	
0x020	PC (vector = 0x01)	
0x022	PC (vector = 0x02)	
.	.	
.	.	
.	.	
0x21A	PC (vector = 0xFE)	
0x21C	PC (vector = 0xFF)	

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 the Extended Instruction in software. In those versions of the Y8002 design where the Extended Instructions are not implemented this trap will never occur, as the Unimplemented Instruction Trap takes precedence.

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 affecting system resources such as I/O. In those versions of the Y8002 design where the System/Normal mode operation is not implemented this trap will never occur.

The SC and Unimplemented Instruction Trap operation is an extension of the original Z8002 processor's SC Instruction Trap. In addition to being triggered by the SC opcode, any opcode that is not implemented in the Y8002 design will also trigger this trap. Because the first word of the offending opcode is pushed onto the system stack as the trap identifier, these cases can always be distinguished in software. This enhanced operation makes the Y8002 design more robust by defining the operation of the processor for any opcode combination. Note that an illegal field encoding in a valid instruction, for example when selecting double or quadruple registers, does not trigger any kind of trap.

The Non-maskable Interrupt is triggered by a falling edge on the Non-maskable Interrupt Request input. This interrupt request really 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. Note that 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 PC values stored in the Program Status Area. Only the lower eight bits of the vector are used for this purpose, selecting from among 256 different PC 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 PC applies and the remaining pending interrupt requests are accepted as appropriate.

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

Chapter 8

Reset

The Device Reset input forces the Y8002 processor 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 state 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	Previous value (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	Previous value (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).
ST[3:0]	Status	All zero (signalling internal operation).
T1_CYCLE	T1 Cycle Identifier	Low (inactive).

Once the Device Reset input is sampled inactive the processor fetches an FCW and PC value from memory locations 0x0002 and 0x004 respectively. Both of these locations are in 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 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. As mentioned previously, the PSAP should be reprogrammed to point elsewhere in memory to avoid a conflict between the reset locations of 0x0002 and 0x0004 and the Program Status Area entry for the Extended Instruction trap.

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.

Chapter 9

Verilog HDL Source

This chapter presents an overview of the Verilog HDL source code for the Y8002 design sufficient to understand the basic organization and operation 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 the user may actually find it useful to modify the test bench.

The design can be viewed from two different standpoints. First is the organization of the design files themselves. Second is the logical organization of the Verilog HDL modules contained in the files. Both of these views are shown in the table below.

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 th*e* *‘define* statements for the design in one central location, and *version.v,* which selects the options for the design.

file organization

- y8002.v
  - version.v
  - core8002.v
    - defines.v
    - ff_byte.v
    - ff_word.v
    - reg_word.v
    - ext_int.v
    - machine.v
    - control.v
    - datapath.v

logical organization

- y8002.v
  - core8002.v
    - ext_int.v
    - machine.v
    - control.v
    - datapath.v
      - reg_word.v
      - reg_file.v
        - ff_byte.v
        - ff_word.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 often 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 Z8002 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 *y8002.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 ACTEL-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 *core8002.v* module contains essentially all of the design, but no logic is actually 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* out and *resetb* in), the instruction register (*inst_reg* out and *buf_inst_reg* in), and the signal which stops all of the state machines for Bus Request, Wait Request and Stop Request (*hold_mach* out and *buf_hold_mach* in). The clock input to this module is called *buf_CLK*, having been buffered outside of the module. As mentioned previously, 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 Z8002 device.

The file *defines.v* contains all of the encoding for multi-bit control fields and state machines. Most of these control field encoding 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.v* and *ff_word.v* 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.v* 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.v* module contains the Y8002 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.v* module contains the external inteface for the Y8002 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.v* 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.v* 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.v* 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 and to generate the register address. The third is used to count the iterations for the multiply and divide operations.

For more detailed information about the design refer to the comments embedded in the Verilog HDL source code.

Chapter 10

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. Also tested are all of the opcodes that cause an illegal instruction trap. 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 a tester. 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 tester uses the NRZ format for all signals and that there are two tester 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

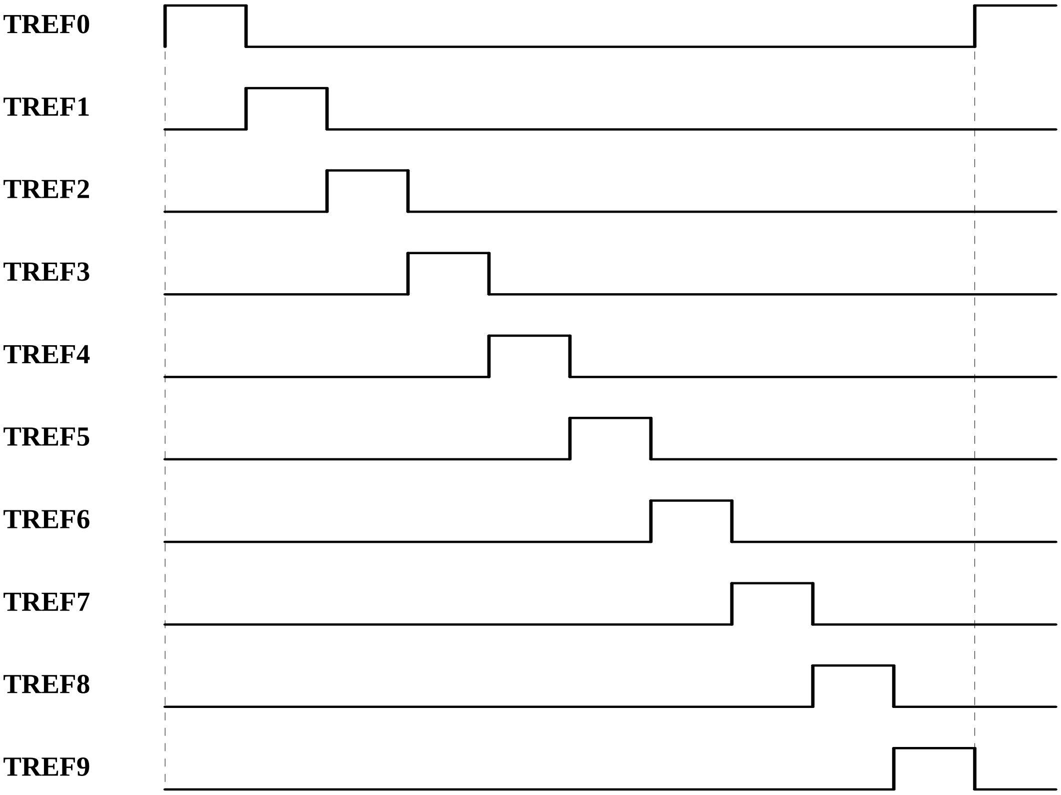
This section contains the program memory and, if necessary, the memory that holds the compare data. Both memories support both byte and word accesses. This is actually done in the memory interface section below.

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 tester 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 FFFEh 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
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 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, 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 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 110,000 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
trp_ops	6	all opcodes which are supposed to trap
int_ops	7	interrupts

Appendix 1

Execution details

This Appendix lists the details of the instruction timing. The external sequence shows the length of each bus cycle performed as part of the instruction, while the internal sequence shows the actual internal sequence. Refer to the design documentation for more details about the internal sequenceMultiply and divide are shown separately at the end.

Instruction	Cycles	External Sequence	Internal Sequence
ADC Rd, Rs	5	5	3, 2
ADCB Rbd, Rbs	5	5	3, 2
ADD Rd, Rs	4	4	3, 1
ADDB Rbd, Rbs	4	4	3, 1
ADDL RRd, RRs	8	8	3, 5
ADD Rd, #data	7	4, 3	3, 1, 3
ADDB Rbd, #data	7	4, 3	3, 1, 3
ADDL RRd, #data	14	4, 3, 7	3, 1, 3, 3, 4
ADD Rd, @Rs	7	4, 3	3, 1, 3
ADDB Rbd, @Rs	7	4, 3	3, 1, 3
ADDL RRd, @Rs	14	4, 3, 7	3, 1, 3, 3, 4
ADD Rd, address	9	3, 3, 3	3, 3, 3
ADDB Rbd, address	9	3, 3, 3	3, 3, 3
ADDL RRd, address	15	3, 3, 3, 6	3, 3, 3, 3, 3
ADD Rd, addr(Rs)	10	3, 4, 3	3, 3, 1, 3
ADDB Rbd, addr(Rs)	10	3, 4, 3	3, 3, 1, 3
ADDL RRd, addr(Rs)	16	3, 4, 3, 6	3, 3, 1, 3, 3, 3
AND Rd, Rs	4	4	3, 1
ANDB Rbd, Rbs	4	4	3, 1
AND Rd, #data	7	4, 3	3, 1, 3
ANDB Rbd, #data	7	4, 3	3, 1, 3
AND Rd, @Rs	7	4, 3	3, 1, 3
ANDB Rbd, @Rs	7	4, 3	3, 1, 3
AND Rd, address	9	3, 3, 3	3, 3, 3
ANDB Rbd, address	9	3, 3, 3	3, 3, 3
AND Rd, addr(Rs)	10	3, 4, 3	3, 3, 1, 3
ANDB Rbd, addr(Rs)	10	3, 4, 3	3, 3, 1, 3

Instruction	Cycles	External Sequence	Internal Sequence
BIT Rd, #b	4	4	3, 1
BITB Rbd, #b	4	4	3, 1
BIT @Rd, #b	8	4, 4	3, 1, 3, 1
BITB @Rd, #b	8	4, 4	3, 1, 3, 1
BIT address, #b	10	3, 3, 4	3, 3, 3, 1
BITB address, #b	10	3, 3, 4	3, 3, 3, 1
BIT addr(Rd), #b	11	3, 4, 4	3, 3, 1, 3, 1
BITB addr(Rd), #b	11	3, 4, 4	3, 3, 1, 3, 1
BIT Rd, Rs	10	4, 6	3, 1, 3, 3
BITB Rbd, Rs	10	4, 6	3, 1, 3, 3
CALL @Rd	10	4, 3, 3	3, 1, 3, 3
CALL address	12	3, 5, 4	3, 3, 2, 3, 1
CALL addr(Rd)	13	3, 6, 4	3, 3, 3, 3, 1
CALR address	10	6, 4	3, 3, 3, 1
CLR Rd	7	7	3, 4
CLRB Rbd	7	7	3, 4
CLR @Rd	8	5, 3	3, 2, 3
CLRB @Rd	8	5, 3	3, 2, 3
CLR address	11	3, 5, 3	3, 3, 2, 3
CLRB address	11	3, 5, 3	3, 3, 2, 3
CLR addr(Rd)	12	3, 6, 3	3, 3, 3, 3
CLRB addr(Rd)	12	3, 6, 3	3, 3, 3, 3
COM Rd	7	7	3, 4
COMB Rbd	7	7	3, 4
COM @Rd	12	5, 4, 3	3, 2, 3, 1, 3
COMB @Rd	12	5, 4, 3	3, 2, 3, 1, 3
COM address	15	3, 5, 4, 3	3, 3, 2, 3, 1, 3
COMB address	15	3, 5, 4, 3	3, 3, 2, 3, 1, 3
COM addr(Rd)	16	3, 6, 4, 3	3, 3, 3, 3, 1, 3
COMB addr(Rd)	16	3, 6, 4, 3	3, 3, 3, 3, 1, 3
COMFLG flag	7	7	3, 4
CP Rd, Rs	4	4	3, 1
CPB Rbd, Rbs	4	4	3, 1
CPL RRd, RRs	8	8	3, 5
CP Rd, #data	7	4, 3	3, 1, 3
CPB Rbd, #data	7	4, 3	3, 1, 3
CPL RRd, #data	14	4, 3, 7	3, 1, 3, 3, 4
CP Rd, @Rs	7	4, 3	3, 1, 3
CPB Rbd, @Rs	7	4, 3	3, 1, 3
CPL RRd, @Rs	14	4, 3, 7	3, 1, 3, 3, 4

Instruction	**Cycles**	**External Sequence**	**Internal Sequence**
CP Rd, address	9	3, 3, 3	3, 3, 3
CPB Rbd, address	9	3, 3, 3	3, 3, 3
CPL RRd, address	15	3, 3, 3, 6	3, 3, 3, 3, 3
CP Rd, addr(Rs)	10	3, 4, 3	3, 3, 1, 3
CPB Rbd, addr(Rs)	10	3, 4, 3	3, 3, 1, 3
CPL RRd, addr(Rs)	16	3, 4, 3, 6	3, 3, 1, 3, 3, 3
CP @Rd, #data	11	4, 4, 3	3, 1, 3, 1, 3
CPB @Rd, #data	11	4, 4, 3	3, 1, 3, 1, 3
CP address, #data	14	3, 4, 4, 3	3, 3, 1, 3, 1, 3
CPB address, #data	14	3, 4, 4, 3	3, 3, 1, 3, 1, 3
CP addr(Rd), #data	15	3, 4, 5, 3	3, 3, 1, 3, 2, 3
CPB addr(Rd), #data	15	3, 4, 5, 3	3, 3, 1, 3, 2, 3
CPD Rd, @Rs, r, cc	20	4, 5, 11	3, 1, 3, 2, 3, 6, 2
CPDB Rbd, @Rs, r, cc	20	4, 5, 11	3, 1, 3, 2, 3, 6, 2
CPDR Rd, @Rs, r, cc	11+9n	4, 5, 9n, 2	3, 1, 3, 2, (3, 6)n, 2
CPDRB Rbd, @Rs, r, cc	11+9n	4, 5, 9n, 2	3, 1, 3, 2, (3, 6)n, 2
CPI Rd, @Rs, r, cc	20	4, 5, 11	3, 1, 3, 2, 3, 6, 2
CPIB Rbd, @Rs, r, cc	20	4, 5, 11	3, 1, 3, 2, 3, 6, 2
CPIR Rd, @Rs, r, cc	11+9n	4, 5, 9n, 2	3, 1, 3, 2, (3, 6)n, 2
CPIRB Rbd, @Rs, r, cc	11+9n	4, 5, 9n, 2	3, 1, 3, 2, (3, 6)n, 2
CPSD @Rd, @Rs, r, cc	25	4, 5, 5, 11	3, 1, 3, 2, 3, 2, 3, 6, 2
CPSDB @Rd, @Rs, r, cc	25	4, 5, 5, 11	3, 1, 3, 2, 3, 2, 3, 6, 2
CPSDR @Rd, @Rs, r, cc	11+14n	4, 5, (5, 9)n, 2	3, 1, 3, 2, (3, 2, 3, 6)n, 2
CPSDRB @Rd, @Rs, r, cc	11+14n	4, 5, (5, 9)n, 2	3, 1, 3, 2, (3, 2, 3, 6)n, 2
CPSI @Rd, @Rs, r, cc	25	4, 5, 5, 11	3, 1, 3, 2, 3, 2, 3, 6, 2
CPSIB @Rd, @Rs, r, cc	25	4, 5, 5, 11	3, 1, 3, 2, 3, 2, 3, 6, 2
CPSIR @Rd, @Rs, r, cc	11+14n	4, 5, (5, 9)n, 2	3, 1, 3, 2, (3, 2, 3, 6)n, 2
CPSIRB @Rd, @Rs, r, cc	11+14n	4, 5, (5, 9)n, 2	3, 1, 3, 2, (3, 2, 3, 6)n, 2
DAB Rbd	5	5	3, 2
DEC Rd, #n	4	4	3, 1
DECB Rbd, #n	4	4	3, 1
DEC @Rd, #n	11	4, 4, 3	3, 1, 3, 1, 3
DECB @Rd, #n	11	4, 4, 3	3, 1, 3, 1, 3
DEC address, #n	13	3, 3, 4, 3	3, 3, 3, 1, 3
DECB address, #n	13	3, 3, 4, 3	3, 3, 3, 1, 3
DEC addr(Rd), #n	14	3, 4, 4, 3	3, 3, 1, 3, 1, 3
DECB addr(Rd), #n	14	3, 4, 4, 3	3, 3, 1, 3, 1, 3
DI int	7	7	3, 4
DIV RRd, Rs	11/22/92	5, div, 1	3, 2, div, 1
DIVL RQd, RRs	14/36/500	5, div, 1	3, 2, div, 1

Instruction	Cycles	External Sequence	Internal Sequence
DIV RRd, #data	13/24/94	4, 3, div, 1	3, 1, 3, div, 1
DIVL RQd, #data	19/41/505	4, 3, 3, div, 1	3, 1, 3, 3, div, 1
DIV RRd, @Rs	13/24/94	4, 3, div, 1	3, 1, 3, div, 1
DIVL RQd, @Rs	19/41/505	4, 3, 3, div, 1	3, 1, 3, 3, div, 1
DIV RRd, address	15/26/96	3, 3, 3, div, 1	3, 3, 3, div, 1
DIVL RQd, address	21/43/507	3, 3, 3, 3, div, 1	3, 3, 3, 3, div, 1
DIV RRd, addr(Rs)	16/27/97	3, 4, 3, div, 1	3, 3, 1, 3, div, 1
DIVL RQd, addr(Rs)	22/44/508	3, 4, 3, 3, div, 1	3, 3, 1, 3, 3, div, 1
DJNZ R, address	11	11	3, 8
DBJNZ Rb, address	11	11	3, 8
EI int	7	7	3, 4
EX Rd, Rs	6	6	3, 3
EXB Rbd, Rbs	6	6	3, 3
EX Rd, @Rs	12	4, 5, 3	3, 1, 3, 2, 3
EXB Rd, @Rs	12	4, 5, 3	3, 1, 3, 2, 3
EX Rd, address	15	3, 4, 5, 3	3, 3, 1, 3, 2, 3
EXB Rbd, address	15	3, 4, 5, 3	3, 3, 1, 3, 2, 3
EX Rd, addr(Rs)	16	3, 5, 5, 3	3, 3, 2, 3, 2, 3
EXB Rbd, addr(Rs)	16	3, 5, 5, 3	3, 3, 2, 3, 2, 3
EXTSB Rd	11	11	3, 8
EXTS RRd	11	11	3, 8
EXTSL RQd	11	11	3, 8
HALT	8+3n	5, 3, 3n	3, 2, 3, 3n
IN Rd, @Rs	10	6, 4	3, 3, 4
INB Rbd, @Rs	10	6, 4	3, 3, 4
IN Rd, port	12	4, 4, 4	3, 1, 3, 1, 4
INB Rbd, port	12	4, 4, 4	3, 1, 3, 1, 4
SIN Rd, port	12	4, 4, 4	3, 1, 3, 1, 4
SINB Rbd, port	12	4, 4, 4	3, 1, 3, 1, 4
INC Rd, #n	4	4	3, 1
INCB Rbd, #n	4	4	3, 1
INC @Rd, #n	11	4, 4, 3	3, 1, 3, 1, 3
INCB @Rd, #n	11	4, 4, 3	3, 1, 3, 1, 3
INC address, #n	13	3, 3, 4, 3	3, 3, 3, 1, 3
INCB address, #n	13	3, 3, 4, 3	3, 3, 3, 1, 3
INC addr(Rd), #n	14	3, 4, 4, 3	3, 3, 1, 3, 1, 3
INCB addr(Rd), #n	14	3, 4, 4, 3	3, 3, 1, 3, 1, 3
IND @Rd, @Rs, r	21	4, 5, 5, 7	3, 1, 3, 2, 4, 1, 3, 2, 2
INDB @Rd, @Rs, r	21	4, 5, 5, 7	3, 1, 3, 2, 4, 1, 3, 2, 2
SIND @Rd, @Rs, r	21	4, 5, 5, 7	3, 1, 3, 2, 4, 1, 3, 2, 2
SINDB @Rd, @Rs, r	21	4, 5, 5, 7	3, 1, 3, 2, 4, 1, 3, 2, 2

Instruction	Cycles	External Sequence	Internal Sequence
INDR @Rd, @Rs, r	11+10n	4, 5, (5, 5)n, 2	3, 1, 3, 2, (4, 1, 3, 2)n, 2
INDRB @Rd, @Rs, r	11+10n	4, 5, (5, 5)n, 2	3, 1, 3, 2, (4, 1, 3, 2)n, 2
SINDR @Rd, @Rs, r	11+10n	4, 5, (5, 5)n, 2	3, 1, 3, 2, (4, 1, 3, 2)n, 2
SINDRB @Rd, @Rs, r	11+10n	4, 5, (5, 5)n, 2	3, 1, 3, 2, (4, 1, 3, 2)n, 2
INI @Rd, @Rs, r	21	4, 5, 5, 7	3, 1, 3, 2, 4, 1, 3, 2, 2
INIB @Rd, @Rs, r	21	4, 5, 5, 7	3, 1, 3, 2, 4, 1, 3, 2, 2
SINI @Rd, @Rs, r	21	4, 5, 5, 7	3, 1, 3, 2, 4, 1, 3, 2, 2
SINIB @Rd, @Rs, r	21	4, 5, 5, 7	3, 1, 3, 2, 4, 1, 3, 2, 2
INIR @Rd, @Rs, r	11+10n	4, 5, (5, 5)n, 2	3, 1, 3, 2, (4, 1, 3, 2)n, 2
INIRB @Rd, @Rs, r	11+10n	4, 5, (5, 5)n, 2	3, 1, 3, 2, (4, 1, 3, 2)n, 2
SINIR @Rd, @Rs, r	11+10n	4, 5, (5, 5)n, 2	3, 1, 3, 2, (4, 1, 3, 2)n, 2
SINIRB @Rd, @Rs, r	11+10n	4, 5, (5, 5)n, 2	3, 1, 3, 2, (4, 1, 3, 2)n, 2
IRET	13	6, 3, 4	3, 3, 3, 3, 1
JP cc, @Rd	7 or 10	4, 3, 3	3, 1, 3, 3
JP cc, address	7	3, 4	3, 3, 1
JP cc, addr(Rd)	8	3, 5	3, 3, 2
JR cc, address	6	3, 3	3, 3
LD Rd, Rs	3	3	3
LDB Rbd, Rbs	3	3	3
LDL RRd, RRs	5	5	3, 2
LD Rd, @Rs	7	4, 3	3, 1, 3
LDB Rbd, @Rs	7	4, 3	3, 1, 3
LDL RRd, @Rs	11	4, 3, 4	3, 1, 3, 3, 1
LD Rd, address	9	3, 3, 3	3, 3, 3
LDB Rbd, address	9	3, 3, 3	3, 3, 3
LDL RRd, address	12	3, 3, 3, 3	3, 3, 3, 3
LD Rd, addr(Rs)	10	3, 4, 3	3, 3, 1, 3
LDB Rbd, addr(Rs)	10	3, 4, 3	3, 3, 1, 3
LDL RRd, addr(Rs)	13	3, 4, 3, 3	3, 3, 1, 3, 3
LD Rd, Rs(#disp)	14	4, 7, 3	3, 1, 3, 4, 3
LDB Rbd, Rs(#disp)	14	4, 7, 3	3, 1, 3, 4, 3
LDL RRd, Rs(#disp)	17	4, 7, 3, 3	3, 1, 3, 4, 3, 3
LD Rd, Rs(Rx)	14	4, 7, 3	3, 1, 3, 4, 3
LDB Rbd, Rs(Rx)	14	4, 7, 3	3, 1, 3, 4, 3
LDL RRd, Rs(Rx)	17	4, 7, 3, 3	3, 1, 3, 4, 3, 3
LD @Rd, Rs	8	5, 3	3, 2, 3
LDB @Rd, Rbs	8	5, 3	3, 2, 3
LDL @Rd, RRs	11	5, 3, 3	3, 2, 3, 3
LD address, Rs	11	3, 5, 3	3, 3, 2, 3
LDB address, Rbs	11	3, 5, 3	3, 3, 2, 3
LDL address, RRs	14	3, 5, 3, 3	3, 3, 2, 3, 3

Instruction	Cycles	External Sequence	Internal Sequence
LD addr(Rd), Rs	12	3, 6, 3	3, 3, 3, 3
LDB addr(Rd), Rbs	12	3, 6, 3	3, 3, 3, 3
LDL addr(Rd), RRs	15	3, 6, 3, 3	3, 3, 3, 3, 3
LD Rd(#disp), Rs	14	4, 7, 3	3, 1, 3, 4, 3
LDB Rd(#disp), Rbs	14	4, 7, 3	3, 1, 3, 4, 3
LDL Rd(#disp), RRs	17	4, 7, 3, 3	3, 1, 3, 4, 3, 3
LD Rd(Rx), Rs	14	4, 7, 3	3, 1, 3, 4, 3
LDB Rd(Rx), Rbs	14	4, 7, 3	3, 1, 3, 4, 3
LDL Rd(Rx), RRs	17	4, 7, 3, 3	3, 1, 3, 4, 3, 3
LD Rd, #data	7	4, 3	3, 1, 3
LDB Rd, #data	7	4, 3	3, 1, 3
LDB Rbd, #data	5	5	3, 2
LDL RRd, #data	11	4, 3, 4	3, 1, 3, 3, 1
LD @Rd, #data	11	4, 4, 3	3, 1, 3, 1, 3
LDB @Rd, #data	11	4, 4, 3	3, 1, 3, 1, 3
LD address, #data	14	3, 4, 4, 3	3, 3, 1, 3, 1, 3
LDB address, #data	14	3, 4, 4, 3	3, 3, 1, 3, 1, 3
LD addr(Rd), #data	15	3, 4, 5, 3	3, 3, 1, 3, 2, 3
LDB addr(Rd), #data	15	3, 4, 5, 3	3, 3, 1, 3, 2, 3
LDA Rd, address	12	4, 4, 4	3, 1, 3, 1, 3, 1
LDA Rd, addr(Rs)	13	4, 5, 4	3, 1, 3, 2, 3, 1
LDA Rd, Rs(#disp)	15	4, 7, 4	3, 1, 3, 4, 3, 1
LDA Rd, Rs(Rx)	15	4, 7, 4	3, 1, 3, 4, 3, 1
LDAR Rd, address	15	4, 7, 4	3, 1, 3, 4, 3, 1
LDCTL FCW, Rs	7	7	3, 4
LDCTL REFRESH, Rs	7	7	3, 4
LDCTL PSAP, Rs	7	7	3, 4
LDCTL NSP, Rs	7	7	3, 4
LDCTL Rd, FCW	7	7	3, 4
LDCTL Rd, REFRESH	7	7	3, 4
LDCTL Rd, PSAP	7	7	3, 4
LDCTL Rd, NSP	7	7	3, 4
LDCTLB FLAGS, Rbs	7	7	3, 4
LDCTLB Rbd, FLAGS	7	7	3, 4
LDD @Rd, @Rs, r	20	4, 5, 4, 7	3, 1, 3, 2, 3, 1, 3, 2, 2
LDDB @Rd, @Rs, r	20	4, 5, 4, 7	3, 1, 3, 2, 3, 1, 3, 2, 2
LDDR @Rd, @Rs, r	11+9n	4, 5, (4, 5)n, 2	3, 1, 3, 2, (3, 1, 3, 2)n, 2
LDDRB @Rd, @Rs, r	11+9n	4, 5, (4, 5)n, 2	3, 1, 3, 2, (3, 1, 3, 2)n, 2
LDI @Rd, @Rs, r	20	4, 5, 4, 7	3, 1, 3, 2, 3, 1, 3, 2, 2
LDIB @Rd, @Rs, r	20	4, 5, 4, 7	3, 1, 3, 2, 3, 1, 3, 2, 2

Instruction	Cycles	External Sequence	Internal Sequence
LDIR @Rd, @Rs, r	11+9n	4, 5, (4, 5)n, 2	3, 1, 3, 2, (3, 1, 3, 2)n, 2
LDIRB @Rd, @Rs, r	11+9n	4, 5, (4, 5)n, 2	3, 1, 3, 2, (3, 1, 3, 2)n, 2
LDK Rd, #data	5	5	3, 2
LDM Rd, @Rs, #n	11+3n	4, 3, 3, 3n, 1	3, 1, 3, 3, 3n, 1
LDM Rd, address, #n	14+3n	3, 4, 3, 3, 3n, 1	3, 3, 1, 3, 3, 3n, 1
LDM Rd, addr(Rs), #n	15+3n	3, 4, 3, 4, 3n, 1	3, 3, 1, 3, 4, 3n, 1
LDM @Rd, Rs, #n	11+3n	4, 3, 3, 3n, 1	3, 1, 3, 3, 3n, 1
LDM address, Rs, #n	14+3n	3, 4, 3, 3, 3n, 1	3, 3, 1, 3, 3, 3n, 1
LDM addr(Rd), Rs, #n	15+3n	3, 4, 3, 4, 3n, 1	3, 3, 1, 3, 4, 3n, 1
LDPS @Rs	12	4, 3, 5	3, 1, 3, 3, 2
LDPS address	16	4, 4, 3, 5	3, 1, 3, 1, 3, 3, 2
LDPS addr(Rs)	17	4, 5, 3, 5	3, 1, 3, 2, 3, 3, 2
LDR Rd, address	14	4, 7, 3	3, 1, 3, 4, 3
LDRB Rbd, address	14	4, 7, 3	3, 1, 3, 4, 3
LDRL RRd, address	17	4, 7, 3, 3	3, 1, 3, 4, 3, 3
LDR address, Rs	14	4, 7, 3	3, 1, 3, 4, 3
LDRB address, Rbs	14	4, 7, 3	3, 1, 3, 4, 3
LDRL address, RRs	17	4, 7, 3, 3	3, 1, 3, 4, 3, 3
MBIT	7	7	3, 4
MREQ	12+7n	12, 7n	3, 2, 3, 4, (3, 4)n
MRES	5	5	3, 2
MSET	5	5	3, 2
MULT RRd, Rs	15/69	5, mul, 1	3, 2, mul, 1
MULTL RQd, RRs	21/258+4n	5, mul, 1	3, 2, mul, 1
MULT RRd, #data	17/71	4, 3, mul, 1	3, 1, 3, mul, 1
MULTL RQd, #data	26/263+4n	4, 3, 3, mul, 1	3, 1, 3, 3, mul, 1
MULT RRd, @Rs	17/71	4, 3, mul, 1	3, 1, 3, mul, 1
MULTL RQd, @Rs	26/263+4n	4, 3, 3, mul, 1	3, 1, 3, 3, mul, 1
MULT RRd, address	19/73	3, 3, 3, mul, 1	3, 3, 3, mul, 1
MULTL RQd, address	28/265+4n	3, 3, 3, 3, mul, 1	3, 3, 3, 3, mul, 1
MULT RRd, addr(Rs)	20/74	3, 4, 3, mul, 1	3, 3, 1, 3, mul, 1
MULTL RQd, addr(Rs)	29/266+4n	3, 4, 3, 3, mul, 1	3, 3, 1, 3, 3, mul, 1
NEG Rd	7	7	3, 4
NEGB Rbd	7	7	3, 4
NEG @Rd	12	5, 4, 3	3, 2, 3, 1, 3
NEGB @Rd	12	5, 4, 3	3, 2, 3, 1, 3
NEG address	15	3, 5, 4, 3	3, 3, 2, 3, 1, 3
NEGB address	15	3, 5, 4, 3	3, 3, 2, 3, 1, 3
NEG addr(Rd)	16	3, 6, 4, 3	3, 3, 3, 3, 1, 3
NEGB addr(Rd)	16	3, 6, 4, 3	3, 3, 3, 3, 1, 3
NOP	7	7	3, 4

Instruction	Cycles	External Sequence	Internal Sequence
OR Rd, Rs	4	4	3, 1
ORB Rbd, Rbs	4	4	3, 1
OR Rd, #data	7	4, 3	3, 1, 3
ORB Rbd, #data	7	4, 3	3, 1, 3
OR Rd, @Rs	7	4, 3	3, 1, 3
ORB Rbd, @Rs	7	4, 3	3, 1, 3
OR Rd, address	9	3, 3, 3	3, 3, 3
ORB Rbd, address	9	3, 3, 3	3, 3, 3
OR Rd, addr(Rs)	10	3, 4, 3	3, 3, 1, 3
ORB Rbd, addr(Rs)	10	3, 4, 3	3, 3, 1, 3
OTDR @Rd, @Rs, r	11+10n	4, 5, (4, 6)n, 2	3, 1, 3, 2, (3, 1, 4, 2)n, 2
OTDRB @Rd, @Rs, r	11+10n	4, 5, (4, 6)n, 2	3, 1, 3, 2, (3, 1, 4, 2)n, 2
SOTDR @Rd, @Rs, r	11+10n	4, 5, (4, 6)n, 2	3, 1, 3, 2, (3, 1, 4, 2)n, 2
SOTDRB @Rd, @Rs, r	11+10n	4, 5, (4, 6)n, 2	3, 1, 3, 2, (3, 1, 4, 2)n, 2
OTIR @Rd, @Rs, r	11+10n	4, 5, (4, 6)n, 2	3, 1, 3, 2, (3, 1, 4, 2)n, 2
OTIRB @Rd, @Rs, r	11+10n	4, 5, (4, 6)n, 2	3, 1, 3, 2, (3, 1, 4, 2)n, 2
SOTIR @Rd, @Rs, r	11+10n	4, 5, (4, 6)n, 2	3, 1, 3, 2, (3, 1, 4, 2)n, 2
SOTIRB @Rd, @Rs, r	11+10n	4, 5, (4, 6)n, 2	3, 1, 3, 2, (3, 1, 4, 2)n, 2
OUT @Rd, Rs	10	6, 4	3, 3, 4
OUTB @Rd, Rbs	10	6, 4	3, 3, 4
OUT port, Rs	12	4, 4, 4	3, 1, 3, 1, 4
OUTB port, Rbs	12	4, 4, 4	3, 1, 3, 1, 4
SOUT port, Rs	12	4, 4, 4	3, 1, 3, 1, 4
SOUTB port, Rbs	12	4, 4, 4	3, 1, 3, 1, 4
OUTD @Rd, @Rs, r	21	4, 5, 4, 8	3, 1, 3, 2, 3, 1, 4, 2, 2
OUTDB @Rd, @Rs, r	21	4, 5, 4, 8	3, 1, 3, 2, 3, 1, 4, 2, 2
SOUTD @Rd, @Rs, r	21	4, 5, 4, 8	3, 1, 3, 2, 3, 1, 4, 2, 2
SOUTDB @Rd, @Rs, r	21	4, 5, 4, 8	3, 1, 3, 2, 3, 1, 4, 2, 2
OUTI @Rd, @Rs, r	21	4, 5, 4, 8	3, 1, 3, 2, 3, 1, 4, 2, 2
OUTIB @Rd, @Rs, r	21	4, 5, 4, 8	3, 1, 3, 2, 3, 1, 4, 2, 2
SOUTI @Rd, @Rs, r	21	4, 5, 4, 8	3, 1, 3, 2, 3, 1, 4, 2, 2
SOUTIB @Rd, @Rs, r	21	4, 5, 4, 8	3, 1, 3, 2, 3, 1, 4, 2, 2
POP Rd, @Rs	8	5, 3	3, 2, 3
POPL RRd, @Rs	12	5, 3, 4	3, 2, 3, 3, 1
POP @Rd, @Rs	12	4, 5, 3	3, 1, 3, 2, 3
POPL @Rd, @Rs	19	4, 3, 5, 3, 4	3, 1, 3, 3, 2, 3, 3, 1
POP address, @Rs	16	3, 5, 5, 3	3, 3, 2, 3, 2, 3
POPL address, @Rs	23	3, 5, 3, 5, 3, 4	3, 3, 2, 3, 3, 2, 3, 3, 1
POP addr(Rd), @Rs	16	3, 5, 5, 3	3, 3, 2, 3, 2, 3
POPL addr(Rd), @Rs	23	3, 5, 3, 5, 3, 4	3, 3, 2, 3, 3, 2, 3, 3, 1

Instruction	Cycles	External Sequence	Internal Sequence
PUSH @Rd, Rs	9	6, 3	3, 3, 3
PUSHL @Rd, RRs	12	6, 3, 3	3, 3, 3, 3
PUSH @Rd, #data	12	4, 5, 3	3, 1, 3, 2, 3
PUSH @Rd, @Rs	13	5, 5, 3	3, 2, 3, 2, 3
PUSHL @Rd, @Rs	20	5, 3, 5, 3, 4	3, 2, 3, 3, 2, 3, 3, 1
PUSH @Rd, address	14	3, 4, 4, 3	3, 3, 1, 3, 1, 3
PUSHL @Rd, address	21	3, 4, 3, 4, 3, 4	3, 3, 1, 3, 3, 1, 3, 3, 1
PUSH @Rd, addr(Rs)	14	3, 4, 4, 3	3, 3, 1, 3, 1, 3
PUSHL @Rd, addr(Rs)	21	3, 4, 3, 4, 3, 4	3, 3, 1, 3, 3, 1, 3, 3, 1
RES Rd, #b	4	4	3, 1
RESB Rbd, #b	4	4	3, 1
RES @Rd, #b	11	4, 4, 3	3, 1, 3, 1, 3
RESB @Rd, #b	11	4, 4, 3	3, 1, 3, 1, 3
RES address, #b	13	3, 3, 4, 3	3, 3, 3, 1, 3
RESB address, #b	13	3, 3, 4, 3	3, 3, 3, 1, 3
RES addr(Rd), #b	14	3, 4, 4, 3	3, 3, 1, 3, 1, 3
RESB addr(Rd), #b	14	3, 4, 4, 3	3, 3, 1, 3, 1, 3
RES Rd, Rs	10	4, 6	3, 1, 3, 3
RESB Rbd, Rs	10	4, 6	3, 1, 3, 3
RESFLG flags	7	7	3, 4
RET cc	7 or 10	4, 3, 3	3, 1, 3, 3
RL Rd, #1	6	6	3, 3
RL Rd, #2	7	7	3, 4
RLB Rbd, #1	6	6	3, 3
RLB Rbd, #2	7	7	3, 4
RLC Rd, #1	6	6	3, 3
RLC Rd, #2	7	7	3, 4
RLCB Rbd, #1	6	6	3, 3
RLCB Rbd, #2	7	7	3, 4
RLDB Rbl, Rbs	9	9	3, 6
RR Rd, #1	6	6	3, 3
RR Rd, #2	7	7	3, 4
RRB Rbd, #1	6	6	3, 3
RRB Rbd, #2	7	7	3, 4
RRC Rd, #1	6	6	3, 3
RRC Rd, #2	7	7	3, 4
RRCB Rbd, #1	6	6	3, 3
RRCB Rbd, #2	7	7	3, 4
RRDB Rbl, Rbs	9	9	3, 6
SBC Rd, Rs	5	5	3, 2
SBCB Rbd, Rbs	5	5	3, 2

Instruction	Cycles	External Sequence	Internal Sequence
SC #src	33	6, 3, 4, 5, 6, 5, 4	3, 3, 3, 3, 1, 3, 2, 3, 3, 3, 2, 3, 1
SDA Rd, Rs	15+3n	4, 11, 3n	3, 1, 3, 8, 3n
SDAB Rbd, Rs	15+3n	4, 11, 3n	3, 1, 3, 8, 3n
SDAL RRd, Rs	15+3n	4, 11, 3n	3, 1, 3, 8, 3n
SDL Rd, Rs	15+3n	4, 11, 3n	3, 1, 3, 8, 3n
SDLB Rbd, Rs	15+3n	4, 11, 3n	3, 1, 3, 8, 3n
SDLL RRd, Rs	15+3n	4, 11, 3n	3, 1, 3, 8, 3n
SET Rd, #b	4	4	3, 1
SETB Rbd, #b	4	4	3, 1
SET @Rd, #b	11	4, 4, 3	3, 1, 3, 1, 3
SETB @Rd, #b	11	4, 4, 3	3, 1, 3, 1, 3
SET address, #b	13	3, 3, 4, 3	3, 3, 3, 1, 3
SETB address, #b	13	3, 3, 4, 3	3, 3, 3, 1, 3
SET addr(Rd), #b	14	3, 4, 4, 3	3, 3, 1, 3, 1, 3
SETB addr(Rd), #b	14	3, 4, 4, 3	3, 3, 1, 3, 1, 3
SET Rd, Rs	10	4, 6	3, 1, 3, 3
SETB Rbd, Rs	10	4, 6	3, 1, 3, 3
SETFLG flags	7	7	3, 4
SLA Rd, #n	13+3n	4, 9, 3n	3, 1, 3, 6, 3n
SLAB Rbd, #n	13+3n	4, 9, 3n	3, 1, 3, 6, 3n
SLAL RRd, #n	13+3n	4, 9, 3n	3, 1, 3, 6, 3n
SLL Rd, #n	13+3n	4, 9, 3n	3, 1, 3, 6, 3n
SLLB Rbd, #n	13+3n	4, 9, 3n	3, 1, 3, 6, 3n
SLLL RRd, #n	13+3n	4, 9, 3n	3, 1, 3, 6, 3n
SRA Rd, #n	13+3n	4, 9, 3n	3, 1, 3, 6, 3n
SRAB Rbd, #n	13+3n	4, 9, 3n	3, 1, 3, 6, 3n
SRAL RRd, #n	13+3n	4, 9, 3n	3, 1, 3, 6, 3n
SRL Rd, #n	13+3n	4, 9, 3n	3, 1, 3, 6, 3n
SRLB Rbd, #n	13+3n	4, 9, 3n	3, 1, 3, 6, 3n
SRLL RRd, #n	13+3n	4, 9, 3n	3, 1, 3, 6, 3n
SUB Rd, Rs	4	4	3, 1
SUBB Rbd, Rbs	4	4	3, 1
SUBL RRd, RRs	8	8	3, 5
SUB Rd, #data	7	4, 3	3, 1, 3
SUBB Rbd, #data	7	4, 3	3, 1, 3
SUBL RRd, #data	14	4, 3, 7	3, 1, 3, 3, 4
SUB Rd, @Rs	7	4, 3	3, 1, 3
SUBB Rbd, @Rs	7	4, 3	3, 1, 3
SUBL RRd, @Rs	14	4, 3, 7	3, 1, 3, 3, 4

Instruction	Cycles	External Sequence	Internal Sequence
SUB Rd, address	9	3, 3, 3	3, 3, 3
SUBB Rbd, address	9	3, 3, 3	3, 3, 3
SUBL RRd, address	15	3, 3, 3, 6	3, 3, 3, 3, 3
SUB Rd, addr(Rs)	10	3, 4, 3	3, 3, 1, 3
SUBB Rbd, addr(Rs)	10	3, 4, 3	3, 3, 1, 3
SUBL RRd, addr(Rs)	16	3, 4, 3, 6	3, 3, 1, 3, 3, 3
TCC cc, Rd	5	5	3, 2
TCCB cc, Rbd	5	5	3, 2
TEST Rd	7	7	3, 4
TESTB Rbd	7	7	3, 4
TESTL RRd	9	9	3, 6
TEST @Rd	8	5, 3	3, 2, 3
TESTB @Rd	8	5, 3	3, 2, 3
TESTL @Rd	13	5, 3, 5	3, 2, 3, 3, 2
TEST address	11	3, 5, 3	3, 3, 2, 3
TESTB address	11	3, 5, 3	3, 3, 2, 3
TESTL address	16	3, 5, 3, 5	3, 3, 2, 3, 3, 2
TEST addr(Rd)	12	3, 6, 3	3, 3, 3, 3
TESTB addr(Rd)	12	3, 6, 3	3, 3, 3, 3
TESTL addr(Rd)	17	3, 6, 3, 5	3, 3, 3, 3, 3, 2
TRDB @Rd, @Rs, r	25	4, 5, 5, 4, 7	3, 1, 3, 2, 3, 2, 3, 1, 3, 4
TRDRB @Rd, @Rs, r	11+14n	4, 5, (5, 4, 5)n, 2	3, 1, 3, 2, (3, 2, 3, 1, 3, 2)n, 2
TRIB @Rd, @Rs, r	25	4, 5, 5, 4, 7	3, 1, 3, 2, 3, 2, 3, 1, 3, 4
TRIRB @Rd, @Rs, r	11+14n	4, 5, (5, 4, 5)n, 2	3, 1, 3, 2, (3, 2, 3, 1, 3, 2)n, 2
TRTDB @Rs1, @Rs2, r	25	4, 5, 5, 4, 7	3, 1, 3, 2, 3, 2, 3, 1, 3, 4
TRTDRB @Rs1, @Rs2, r	11+14n	4, 5, (5, 4, 5)n, 2	3, 1, 3, 2, (3, 2, 3, 1, 3, 2)n, 2
TRTIB @Rs1, @Rs2, r	25	4, 5, 5, 4, 7	3, 1, 3, 2, 3, 2, 3, 1, 3, 4
TRTIRB @Rs1, @Rs2, r	11+14n	4, 5, (5, 4, 5)n, 2	3, 1, 3, 2, (3, 2, 3, 1, 3, 2)n, 2
TSET Rd	7	7	3, 4
TSETB Rbd	7	7	3, 4
TSET @Rd	11	5, 3, 3	3, 2, 3, 3
TSETB @Rd	11	5, 3, 3	3, 2, 3, 3
TSET address	14	3, 5, 3, 3	3, 3, 2, 3, 3
TSETB address	14	3, 5, 3, 3	3, 3, 2, 3, 3
TSET addr(Rd)	15	3, 6, 3, 3	3, 3, 3, 3, 3
TSETB addr(Rd)	15	3, 6, 3, 3	3, 3, 3, 3, 3
XOR Rd, Rs	4	4	3, 1
XORB Rbd, Rbs	4	4	3, 1
XOR Rd, #data	7	4, 3	3, 1, 3
XORB Rbd, #data	7	4, 3	3, 1, 3

Instruction	Cycles	External Sequence	Internal Sequence
XOR Rd, @Rs	7	4, 3	3, 1, 3
XORB Rbd, @Rs	7	4, 3	3, 1, 3
XOR Rd, address	9	3, 3, 3	3, 3, 3
XORB Rbd, address	9	3, 3, 3	3, 3, 3
XOR Rd, addr(Rs)	10	3, 4, 3	3, 3, 1, 3
XORB Rbd, addr(Rs)	10	3, 4, 3	3, 3, 1, 3
LD @Rd, EPU	11+3n	4, 6, 3n, 1	3, 1, 3, 3, 3n, 1
LD address, EPU	14+3n	3, 4, 6, 3n, 1	3, 3, 1, 3, 3, 3n, 1
LD addr(Rd), EPU	15+3n	3, 4, 7, 3n, 1	3, 3, 1, 3, 4, 3n, 1
LD EPU, @Rs	11+3n	4, 6, 3n, 1	3, 1, 3, 3, 3n, 1
LD EPU, address	14+3n	3, 4, 6, 3n, 1	3, 3, 1, 3, 3, 3n, 1
LD EPU, addr(Rs)	15+3n	3, 4, 7, 3n, 1	3, 3, 1, 3, 4, 3n, 1
LD Rd, EPU	11+4n	4, 6, 4n, 1	3, 1, 3, 3, 4n, 1
LD EPU, Rs	11+4n	4, 6, 4n, 1	3, 1, 3, 3, 4n, 1
LD FCW, EPU	15	4, 6, 5	3, 1, 3, 3, 4, 1
LD EPU, FCW	15	4, 6, 5	3, 1, 3, 3, 4, 1
EPUI	11+4n	4, 6, 4n, 1	3, 1, 3, 3, 4n, 1

The table below shows the detailed timing for Divide and Multiply: Idle cycles add to the previous bus transaction, as they do not correspond to a new bus transaction.

Operation	Case	Min Cycles	Max Cycles	External Sequence
div	word	86	86	17 idle, 4 (15 times), 9
div	word by zero	5	5	5 idle
div	word overflow	16	16	16 idle
div	long	494	494	37 idle, 14 (31 times), 23
div	long by zero	8	8	8 idle
div	long overflow	30	30	30 idle
mul	word	63	63	13 idle, 3 (15 times), 5
mul	word zero	9	9	9 idle
mul	long	256	380	24 idle, 7 or 11 (31 times), 15
mul	long zero	15	15	15 idle

Appendix 2

Unimplemented Features/Instructions

Revision 1 of the Y8002 design does not implement a number of features and instructions of the Z8000 architecture. Unimplemented features include the refresh mechanism (although the REFRESH register is implemented), System/User mode of operation (only System mode is supported and the SYS bit in the FCW is forced High), and the Extended Processing architecture (the EPA bit in the FCW is forced Low). Even though Normal mode is not supported, the Normal Stack Pointer is present.

Unimplemented instructions are listed in the table below:

Instruction	**Opcode (first word only)**
LD @Rd, EPU	00001111_Rdnz_11xx
LD address, EPU	01001111_0000_11xx
LD addr(Rd), EPU	01001111_Rdnz_11xx
LD EPU, @Rs	00001111_Rsnz_01xx
LD EPU, address	01001111_0000_01xx
LD EPU, addr(Rs)	01001111_Rsnz_01xx
LD Rd, EPU	10001111_0xxx_00xx
LD EPU, Rs	10001111_0xxx_10xx
LD FCW, EPU	10001110_xxxx_00xx
LD EPU, FCW	10001110_xxxx_10xx
EPUI	10001110_xxxx_01xx

Note that any of these instructions can be easily added to the design if required.

Appendix 3

Trapped Opcodes

This Appendix lists all of the instruction ecodings which will result in an Unimplemented Instruction trap, along with the valid instructions included with each encoding.

Opcode	Instruction(s)
0x00110x_xxxx_xx11	Reserved
0x00110x_xxxx_1x1x	Reserved
0x00110x_xxxx_11xx	Reserved
0x001100_xxxx_1xx1	Reserved
01001101_xxxx_1xx1	Reserved
x0001100_xxxx_xx11	Reserved
x0001100_xxxx_1x1x	Reserved
x0001100_xxxx_11xx	Reserved
10001101_xxxx_1xx1	Reserved
10001101_xxxx_1x1x	Reserved
10001101_xxxx_11xx	Reserved
0x011100_xxxx_0000	Reserved
0x011100_xxxx_xx1x	Reserved
0x011100_xxxx_x1xx	Reserved
0x00111x_xxxx_xxxx	EPU
x000111x_xxxx_xxxx	EPU
00110110_xxxx_xxxx	Reserved
00111000_xxxx_xxxx	Reserved
00111001_xxxx_xxx1	Reserved
00111001_xxxx_xx1x	Reserved
00111001_xxxx_x1xx	Reserved
00111001_xxxx_1xxx	Reserved
0011101x_xxxx_11xx	Reserved
01111011_xxxx_0xx1	Reserved
01111011_xxxx_0x10	Reserved
01111011_xxxx_x011	Reserved
01111011_xxxx_x100	Reserved
01111011_xxxx_111x	Reserved

Opcode	Instruction(s)
01111000_xxxx_xxxx	Reserved
01111001_xxxx_xxx1	Reserved
01111001_xxxx_xx1x	Reserved
01111001_xxxx_x1xx	Reserved
01111001_xxxx_1xxx	Reserved
01111101_xxxx_x00x	Reserved
01111101_xxxx_x1x0	Reserved
0111111x_xxxx_xxxx	Reserved, SC
1001110x_xxxx_xxx1	Reserved
1001110x_xxxx_xx1x	Reserved
1001110x_xxxx_x1xx	Reserved
1001110x_xxxx_0xxx	Reserved
100111x1_xxxx_xxxx	Reserved
10110010_xxxx_x1x1	Reserved
10111001_xxxx_xxxx	Reserved
10111000_xxxx_xxx1	Reserved
1011101x_xxxx_x011	Reserved
1011101x_xxxx_x1x1	Reserved
10111111_xxxx_xxxx	Reserved

Appendix 4

Known Timing Differences

Every effort has been made to match the documented operation of the original Z8002 microprocessor as far as timing is concerned. However, there are three cases where this was not possible, and these are described in this Appendix. The cases are Interrupt Acknowledge timing, the Divide instruction, and the Multiply instruction.

In addition to the known timing differences, there are a number of cases of potential timing differences. These cases arise where the Zilog documentation is either incomplete or undefined. The cases that are potentially different are Shift and Shift Dynamic with zero shift, Shift and Shift Dynamic with an out-of-range shift value, and Halt when responding to an interrupt. If the Z8002 behavior is logical, the Y8002 design probably matches it.

The interrupt acknowledge cycle timing includes an aborted IF1 bus transaction. In the Y8002 design this transaction is always four clock cycles in length (plus any Wait states). The Zilog documentation states that this bus transaction may be anywhere from three to seven clock cycles long (plus Wait states) but does not indicate why this is the case. Variable timing for this instance does not seem logical and the Y8002 design does not attempt to match this behavior.

The Divide instruction differences are shown in the table below. Note that the numbers in the Zilog documentation are somewhat suspicious because they do not appear to correctly account for the different addressing modes.

Divide	**Y8002 timing**			**published Z8002 timing**		
Instruction	**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

The Multiply instruction differences are shown in the table below. As in the case of Divide, the Zilog documentation does not appear to correctly account for the different addressing modes. In addition, the seven clock cycle difference, depending on whether an operand bit is one or zero in the case of MULTL, seems excessive.

Multiply		**Y8002 timing**			**published Z8002 timing**	
Instruction	**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