Y90 Microprocessor

Technical Manual

Disclaimer

Systemyde International Corporation reserves the right to make changes at any time, without notice, to improve design or performance and provide the best product possible. Systemyde International Corporation makes no warrant for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make any commitment to update the information contained herein.

Systemyde International Corporation products are not authorized for use in life support devices or systems unless a specific written agreement pertaining to such use is executed between the manufacturer and the President of Systemyde International Corporation. Nothing contained herein shall be construed as a recommendation to use any product in violation of existing patents, copyrights or other rights of third parties. No license is granted by implication or otherwise under any patent, patent rights or other rights, of Systemyde International Corporation. All trademarks are trademarks of their respective companies.

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 Y90 design defines "proper operation".

Notice:

"Z80", "Z180" 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.

Table of Contents

Revison History

Date	Changes	Page(s)
03/04/2010	Added **ext_status_bus** and **ext_status_rd** to MPU version.	Ch 3 and Ch 11
	Added MPU version timing diagrams.	Ch 4
	Added Z80/Z180 compile option for CCF instruction.	Ch 5
	Changed timing for MLT to increase overall clock speed	Ch 5
	Changed opcode for LDMS instruction.	Ch 6
	Changed timing for some new instructions.	Ch 6
	Added **dma_req** Status compile option.	Ch 11
	Added **wait_req** Status compile option.	Ch 11
	Added RTI Status.	Ch 11
	Added RTI Generator discussion, diagrams.	Ch 12
03/27/2010	Updated **dma_req** timing diagram.	Ch 4
	Added HDL Compile Option descriptions.	Appendix 1
04/08/2010	Signal values during **dma_ack**, **halt_tran** and **sleep_tran**	Ch 3 & Ch 4
04/12/2010	Signal values during HALT and SLP instructions.	Ch 5
06/03/2010	Added Top Level Verilog chapter	Ch 13
10/11/2012	Updates to System Status Block	Ch 11
10/16/2012	Changed all MMU and System Status instructions so that they do not sample interrupts.	Ch 6 and Ch 11
10/22/2012	Fixed Typos in interrupt table	156
11/10/2012	Added **nmiack_tran** output	Ch 3
11/14/2012	Edits to pin descriptions.	Ch 3
12/17/2012	Clarify Trap, RTI and System Status operation	Ch 4, Ch 11, Ch 12
08/06/2013	Added **int_req_en** bus, **nmi_addr** bus to MPU version	Ch 3, Ch 4
01/16/2015	Fixed typo in MLT instruction encoding	104

Introduction

This book documents the operation of the Y90 microprocessor. The Y90 design is supplied in Verilog HDL and can be implemented in any technology supported by a logic synthesis tool that accepts Verilog HDL. Included in the design package is a test bench that exercises all instructions, flag settings, and representative data patterns. The test patterns should achieve at least 95% fault coverage.

The Y90 CPU was designed in a clean-room environment and is an upgrade of the Zilog Z80 and Z180 microprocessors. Only publicly available documentation was used to create this design so there may be minor differences where the public documentation is misleading or lacking. The instruction execution times are not identical between the two designs. The Y90 CPU operates with a consistent two-clock-cycle machine cycle, while the Z80 and Z180 microprocessors use machine cycles that vary from three to seven clock cycles.

The Y90 design, depending on the version, may not implement all of the instructions, features or operating modes of the Z80 architecture. The specific differences are covered in the various appendices.

This document should always be used as the final word on the operation of the Y90 CPU, but it is useful to refer to the Zilog documentation if the description given here is too cryptic. The Z80 architecture is over thirty years old, so it is assumed that it is already at least somewhat familiar to the reader.

The Y90 CPU is accompanied by full design documentation, in the form of a large spreadsheet, which describes nearly every facet of the internal operation of the processor. This provides knowledgeable users the opportunity to customize the design for unique application requirements.

The Y90 design is available in three versions: the Y90 CPU consists of the processor only, while the Y90 MPU surrounds the processor with a number of modules to support operation with an RTOS. This includes a memory management unit, an interrupt controller, a DMA request controller, a watch-dog timer, real-time interrupt for time slicing, and various fault monitors. The third version, called the Y90-180, contains the peripheral functions present in the original Zilog Z180. This version of the design is described in a separate document.

Features

* Fully functional synthesizable Verilog HDL version of the Z80 CPU or Z180 CPU

* Vendor and technology independent

* Software compatible with several industry-standard processors

* 189 instructions (more with optional modules)

* Eight addressing modes

* 64K byte (up to 256M byte with optional paged MMU) memory addressing capability

* Separate 64K byte I/O address space

* 16 bit ALU with bit, byte and BCD operations

* Powerful vectored interrupt capability with separate interrupt vector input bus

* Static, fully synchronous design uses no 3-state buses

* Uniform 2 clock-cycle machine cycle

* Memory interface matches common FPGA and ASIC memory timing

* Separate I/O bus, compatible with AMBA Peripheral Bus

* Illegal instruction detection, optional write-protect and execute-protect

* State machines include optional illegal state detection

* Full design documentation included

* Verilog simulation and test suite included

* Optional modules (interrupt, bus control, system status, real-time interrupt)

Shown below are the registers visible to the programmer. The main registers have both a primary and an alternate version. The primary register set consists of A, F, B, C, D, E, H, and L, while the alternate register set consists of A', F', B', C', D', E', H', and L'. At any given time only one bank is active, and care must be used when switching between banks, as there is no way for the programmer to check which bank is active. The accumulator, A, is the destination for all 8-bit arithmetic and logic operations, while the Flag register F contains the flag results of arithmetic and logic operations. The other general-purpose registers can be paired, BC or DE or HL, to form 16-bit registers. There are two index registers, IX and IY, used for indexed addressing mode. The I register holds the upper eight bits of the interrupt vector table address for use in Interrupt Mode 2. The R register is left over from the original Z80 architecture, where it was used to hold a refresh address for DRAMs. In the Y90 it is just another general purpose register. The Stack pointer, SP, holds the address of the stack, and the Program Counter, PC, holds the address of the currently executing instruction.

A	F
B	C
D	E
H	L
IX	
IY	

Main Register Bank

A'	F'
B'	C'
D'	E'
H'	L'

Alternate Register Bank

I	R

Special Function Registers

SP
PC

Pin Descriptions

The Y90 design does not attempt to match the signals or timing present on either the Z80 microprocessor or the Z180 microprocessor. Rather, the interfaces and signals are optimized for use in either an ASIC or an FPGA.

Memory and I/O use separate address and data buses in addition to the separate control signals. The memory bus is designed to match typical ASIC and FPGA memory timing, although it can be used with stand-alone memory devices just as easily. A separate interrupt vector bus is provided for use with an interrupt controller. If desired, this interrupt vector bus can be tied to either the memory or I/O input bus for operation more closely resembling that of the original Z80 and Z180.

The interface signals for the Y90 CPU are detailed below. Note that all inputs except the two resets are sampled by the rising edge of the clock and all outputs change in response to the rising edge of the clock.

clearb (input, active-Low) The Master (test) Reset signal is used to initialize all of the flip-flops that are not initialized by the user reset signal. Most user-visible registers are not affected by the user reset, so this signal allows full initialization for testing and simulation. This is an asynchronous signal that should be used for Power-On Reset.

clkc (input, active-High) The CPU Clock connects to all flip-flops in the design.

dma_ack (output, active-High) The DMA Acknowledge signal is activated to indicate that the processor has halted to allow another bus master to use the bus. The **iack_tran**, **io_addr_out**, **io_data_out**, **io_tran**, **mem_addr_out**, **mem_data_out**, **mem_tran**, **nmiack_tran**, **reti_tran** and **t1** signals are all inactive (Low) during this time. The processor will signal **dma_ack** while in the Halt or Sleep state without de-asserting the **halt_tran** or **sleep_tran** signals. Interrupts are not sampled while the **dma_ack** signal is active, so the exit from a coincident Halt or Sleep state will be deferred until the **dma_ack** signal is no longer active.

dma_req (input, active-High) The DMA Request signal requests that the processor halt to allow another bus master to transfer data on the bus. The processor only

releases the bus between instructions, rather than between individual bus transactions.

en_prftch (input, active-High). The Enable Prefetch signal enables the prefetch operation. Although the prefetch mode can be changed dynamically, it is recommended that this signal be tied either High or Low. The prefetch mechanism increases performance by prefetching an opcode byte during any address calculation time. Only the prefix byte (0xCB, 0xDD, 0xED or 0xFD) of a multi-byte instruction can actually be used after being prefetched.

fault_detect (output, active-High) The Fault Detect output is activated when an illegal state is detected in the main state machine. The main state machine uses a modified one-hot encoding, so the parity of a valid state is always even. Whenever the main state has odd parity an upset has occurred and this output will be activated.

halt_tran (output, active-High) The Halt Transaction signal is activated by the Halt instruction. While in the Halt state the CPU freezes and waits for an interrupt. The **iack_tran**, **io_addr_out**, **io_data_out**, **io_tran**, **mem_addr_out**, **mem_data_out**, **mem_tran**, **nmiack_tran**, **reti_tran** and **t1** signals are all inactive (Low) during this time.

iack_tran (output, active-High) The Interrupt Acknowledge Transaction signal is activated to identify an interrupt acknowledge bus transaction for an enabled Maskable Interrupt request. During an interrupt acknowledge the interrupt vector data bus is sampled, although the sampled value is only used in Interrupt Mode 0 or 2 with a maskable interrupt request.

inst2_trap (output, active-High) The Instruction Byte 2 Trap signal is activated during an interrupt acknowledge bus transaction if the interrupt acknowledge is due to the fetch of an illegal instruction during in the second byte of a two-byte opcode. This information can be used, along with the PC value written to the stack during the interrupt acknowledge sequence, to pinpoint the location of the illegal instruction.

inst3_trap (output, active-High) The Instruction Byte 3 Trap signal is activated during an interrupt acknowledge bus transaction if the interrupt acknowledge is due to the fetch of an illegal instruction during in the second byte of a three-byte opcode. This information can be used, along with the PC value written to the stack during the interrupt acknowledge sequence, to pinpoint the location of the illegal instruction.

int_req (input, active-High) The Interrupt Request signal is the maskable interrupt request. Maskable interrupts can be enabled and disabled under program con-

trol. This interrupt request is not latched, so it should remain active until an interrupt acknowledge transaction occurs.

io_addr_out (output, 16-bit bus) The I/O Address Output bus carries the address of the I/O port during an I/O transaction. To save power, this bus holds the current value until the next I/O transaction or until the **dma_ack** signal is activated.

io_data_in (input, 8-bit bus) The I/O Data Input bus is sampled during the various I/O input instructions. A separate bus allows peripherals to be connected without loading the memory data bus.

io_data_out (output, 8-bit bus) The I/O Data Output bus carries the output data for I/O output instructions. To save power, this bus holds the current value until the next I/O transaction or until the **dma_ack** signal is activated.

io_read (output, active-High) The I/O Read signal indicates the direction of data transfer during I/O transactions. High signals read and Low signals write. This signal is valid only during I/O transactions, and is held Low at all other times.

io_strobe (output, active-High) The I/O Strobe signal is one clock cycle wide (in the absence of Wait states) and identifies the data transfer clock cycle for I/O transactions.

io_tran (output, active-High) The I/O Transaction signal is activated for all I/O transactions.

ivec_data_in (input, 8-bit bus) The Interrupt Vector Data Input bus is sampled during interrupt acknowledge transactions. If the interrupt acknowledge was for a maskable interrupt and the CPU is in Interrupt Mode 2, this vector is used as a pointer in the interrupt vector table to find the starting address of the interrupt service routine. In Interrupt Mode 0 the vector is a one-byte RST instruction.

ivec_rd (output, active-High) The Interrupt Vector Read signal is one clock cycle wide (in the absence of Wait states) and identifies the data transfer clock cycle for interrupt acknowledge transactions.

mem_addr_out (output, 16-bit bus) The Memory Address Output bus carries the address during memory read and write transactions. This bus is driven with all zeros while the **dma_ack** signal is active.

mem_data_in (input, 8-bit bus) The Memory Data Input bus is sampled during memory read transactions. A separate bus allows peripherals to be connected without loading the memory data bus.

mem_data_out (output, 8-bit bus) The Memory Data Output bus carries the output data for memory write transactions. This bus holds the current value until the next I/O transaction or until the **dma_ack** signal is activated.

mem_rd (output, active-High) The Memory Read signal is one clock cycle wide (in the absence of Wait states) and identifies the data transfer clock cycle for memory read transactions.

mem_tran (output, active-High) The Memory Transaction signal is activated for memory read and write transactions. The **mem_tran** signal is active during the Halt state but is inactive during the Sleep state and while the **dma_ack** signal is active.

mem_wr (output, active-High) The Memory Write signal is one clock cycle wide (in the absence of Wait states) and identifies the data transfer clock cycle for memory write transactions.

nmiack_tran (output, active-High) The NMI Acknowledge Transaction signal is activated to identify an interrupt acknowledge bus transaction for a Non-Maskable Interrupt request.

nmi_req (input, active-High) The Non-Maskable Interrupt Request signal unconditionally interrupts the CPU. This request is internally latched, so that it can be as short as one clock cycle wide.

resetb (input, active-Low) The User Reset signal is used to initialize all state flip-flops and some user registers (the I, R, PC and SP registers). This is an asynchronous signal.

reti_tran (output, active-High) The Return From Interrupt transaction signal is activated immediately after the second stack read transaction during the Return From Interrupt (RETI) instruction. This signal may be used by an external interrupt controller to re-enable interrupts, for example.

sleep_tran (output, active-High) The Sleep Transaction signal is activated by the Sleep instruction. While in the Sleep state the CPU freezes and waits for an interrupt. The **iack_tran**, **io_addr_out**, **io_data_out**, **io_tran**, **mem_addr_out**, **mem_data_out**, **mem_tran**, **nmiack_tran**, **reti_tran** and **t1** signals are all inactive (Low) during this time.

t1 (output, active-High) The T1 signal is active during the first clock cycle of a bus transaction. This signal is inactive during the Halt and Sleep states.

wait_req (input, active-High) The Wait Request signal temporarily halts the CPU, usually to wait for memory access time to be met. The wait request is not honored while the **dma_ack**, **halt_tran** or **sleep_tran** signals are active.

The interface signals for the Y90 MPU are detailed below. Many signals are common to the two versions, but this version adds a significant number of system-level features.

clearb (input, active-Low) The Master (test) Reset signal is used to initialize all of the flip-flops that are not initialized by the user reset signal. Most user-visible registers are not affected by the user reset, so this signal allows full initialization for testing and simulation. This is an asynchronous signal that should be used for Power-On Reset.

clkc (input, active-High) The CPU Clock connects to all flip-flops in the design.

dma_ack_bus (output, 8-bit bus, active-High) This bus is composed of eight individual DMA Acknowledge signals, only one of which will be active at a time. A **dma_ack_bus** signal is activated to indicate that the processor has halted to allow another bus master to use the bus. The **iack_tran**, **io_addr_out**, **io_data_out**, **io_tran**, **mem_addr_out**, **mem_data_out**, **mem_tran**, **nmiack_tran**, **reti_tran** and **t1** signals are all inactive (Low) during this time. The processor can activate a **dma_ack_bus** signal while in the Halt or Sleep state without de-asserting the **halt_tran** or **sleep_tran** signals. Interrupts are not sampled while a **dma_ack_bus** signal is active, so the exit from a coincident Halt or Sleep state will be deferred until no **dma_ack_bus** signal is active.

dma_req_bus (input, 8-bit bus, active-High) This bus is composed of eight individual DMA Request signals. A DMA Request signal requests that the processor halt to allow another bus master to transfer data on the bus. The processor only releases the bus between instructions, rather than between individual bus transactions.

drq_act_lim (input, 5-bit value) The DMA Request Active Limit value sets the limit for the amount of time that a DMA may control the bus. The clock cycle limit is sixteen times this value, and a zero value disables the limit check. This value can be either static or controlled by an external I/O port.

drq_idl_lim (input, 5-bit value) The DMA Request Idle Limit value sets the minimum amount of time that the processor will execute between granting the bus for DMA. The clock cycle limit is sixteen times this value, and a zero value dis-

ables this function. This value can be either static or controlled by an external I/O port.

drq_timeout (output, active-High) The DMA Request Time-out signal is activated for one clock cycle whenever the clock cycle limit for DMA control of the bus is reached. External logic may use this information to accumulate performance information, or as a DMA fault indicator.

en_prftch (input, active-High). The Enable Prefetch signal enables the prefetch operation. Although the prefetch mode can be changed dynamically, it is recommended that this signal be tied either High or Low. The prefetch mechanism increases performance by prefetching an opcode byte during any address calculation time. Only the prefix byte (0xCB, 0xDD, 0xED or 0xFD) of a multi-byte instruction can actually be used after being prefetched.

exec_inh (input, active-High) The Execute Inhibit signal is sampled during the **t1** time of fetch transactions for the first byte of an instruction. A trap is generated if this signal is sampled active at this time. This is accomplished by jamming an 0xC7 (RST 0) on the data bus into the CPU in response to the instruction fetch. Because only the fetch of the first byte of an instruction samples the **exec_inh** signal, a multi-byte instruction that starts in an execute-enabled region will execute properly, but if the next in-line instruction is in an execute-protected region it will cause a trap. The execute-inhibit function works identically with or without prefetch being enabled.

ext_stat_bus (input, 8-bit bus, active-High) This bus is composed of eight individual External Status signals. The state of this bus is sampled during the **t1** time of the instruction that reads the System Status Block. These signals will typically be used to report on the health of other parts of the system. Since the **ext_stat_bus** is only sampled for the read of the System Status Block, any transient conditions that contribute to this status must be latched externally.

ext_stat_rd (output, active-High) The External Status Read signal is activated when the System Status Block has been read. This signal can be used to clear any latches associated with signals contributing to the **ext_stat_bus** status.

fatal_detect (output, active-High) The Fatal Detect output is activated whenever an unrecoverable error is detected, and remains active until the next Reset. There are only two unrecoverable errors: First, when the **exec_inh** signal is activated for an instruction fetch in Page 0. Second, when the **wr_inh** signal is activated during a stack write. Both circumstances will lead to an infinite loop of traps. The **fatal_detect** signal forces a reset of the device to prevent this infinite loop.

fault_detect (output, active-High) The Fault Detect output is activated when an illegal state is detected in the main state machine. The main state machine uses a modified one-hot encoding, so the parity of a valid state is always even. Whenever the main state has odd parity an upset has occurred and this output will be activated. The **fault_detect** signal forces a reset of the device.

halt_tran (output, active-High) The Halt Transaction signal is activated by the Halt instruction. While in the Halt state the CPU freezes and waits for an interrupt. The **iack_tran**, **io_addr_out**, **io_data_out**, **io_tran**, **mem_addr_out**, **mem_data_out**, **mem_tran**, **nmiack_tran**, **reti_tran** and **t1** signals are all inactive (Low) during this time.

iack_tran (output, active-High) The Interrupt Acknowledge Transaction signal is activated to identify an interrupt acknowledge bus transaction. An interrupt acknowledge occurs in response to an enabled Maskable Interrupt request. During an interrupt acknowledge the interrupt controller supplies an interrupt vector for the maskable interrupt.

iack_wait_lim (input, 5-bit value) The Interrupt Acknowledge Wait Limit value sets the limit for the number of Wait states during an interrupt acknowledge transaction. This value is used directly, and a zero value disables the limit check. This value can be either static or controlled by an external I/O port.

int_ack_bus (output, 16-bit bus, active-High) This bus is composed of sixteen individual Interrupt Acknowledge signals. One **int_ack_bus** signal is activated during the interrupt acknowledge transaction to indicate exactly which interrupt request is being acknowledged. The active **int_ack_bus** signal can be used to clear the interrupt condition if desired.

int_req_bus (input, 16-bit bus, active-High) This bus is composed of sixteen individual Interrupt Request signals. The Interrupt Request signal is the maskable interrupt request, and is enabled using the corresponding Interrupt Enable signal. Maskable interrupts are globally enabled and disabled under program control. This interrupt request is latched, so it does not need to remain active until an interrupt acknowledge transaction occurs. Each interrupt request latch is cleared automatically by the corresponding Interrupt Acknowledge signal.

int_req_en (input, 16-bit bus, active-High) This bus is composed of sixteen individual Interrupt Enable signals. The Interrupt Enable signals provide a way to control the Maskable Interrupt requests via external hardware. These signals also provide a means to clear the interrupt request latches outside of the normal Interrupt Acknowledge clear, by temporarily disabling the Interrupt Request.

io_addr_out (output, 16-bit bus) The I/O Address Output bus carries the address of the I/O port during an I/O transaction. To save power, this bus holds the current value until the next I/O transaction or until the **dma_ack** signal is activated.

io_data_in (input, 8-bit bus) The I/O Data Input bus is sampled during the various I/O input instructions. A separate bus allows peripherals to be connected without loading the memory data bus.

io_data_out (output, 8-bit bus) The I/O Data Output bus carries the output data for I/O output instructions. To save power, this bus holds the current value until the next I/O transaction or until the **dma_ack** signal is activated.

io_read (output, active-High) The I/O Read signal indicates the direction of data transfer during I/O transactions. High signals read and Low signals write. This signal is valid only during I/O transactions, and is held Low at all other times.

io_strobe (output, active-High) The I/O Strobe signal is one clock cycle wide (in the absence of Wait states) and identifies the data transfer clock cycle for I/O transactions.

io_tran (output, active-High) The I/O Transaction signal is activated for all I/O transactions.

io_wait_lim (input, 5-bit value) The I/O Transaction Wait Limit value sets the limit for the number of Wait states during an I/O transaction. This value is used directly, and a zero value disables the limit check. This value can be either static or controlled by an external I/O port.

ivec_offset (input, 3-bit value) The Interrupt Vector Offset value is used to create the interrupt vector generated by the interrupt controller. In interrupt mode 2 this value is the three most-significant bits of the interrupt vector. In Interrupt Mode 0 this value selects which RST instruction to return during the interrupt acknowledge (0x0 is RST 0, 0x1 is RST 8, and so on). This value can be either static (the typical case), controlled by an external I/O port, or modified by the pending interrupt.

mem_addr_out (output, 16-bit bus) The Memory Address Output bus carries the address during memory read and write transactions. This bus is driven with all zeros while the **dma_ack** signal is active.

mem_data_in (input, 8-bit bus) The Memory Data Input bus is sampled during memory read transactions. A separate bus allows peripherals to be connected without loading the memory data bus.

mem_data_out (output, 8-bit bus) The Memory Data Output bus carries the output data for memory write transactions. This bus holds the current value until the next I/O transaction or until the **dma_ack** signal is activated.

mem_rd (output, active-High) The Memory Read signal is one clock cycle wide (in the absence of Wait states) and identifies the data transfer clock cycle for memory read transactions.

mem_tran (output, active-High) The Memory Transaction signal is activated for memory read and write transactions. The **mem_tran** signal is active during the Halt state but is inactive during the Sleep state and while the **dma_ack** signal is active.

mem_wait_lim (input, 5-bit value) The Memory Transaction Wait Limit value sets the limit for the number of Wait states during a memory transaction. This value is used directly, and a zero value disables the limit check. This value can be either static or controlled by an external I/O port.

mem_wr (output, active-High) The Memory Write signal is one clock cycle wide (in the absence of Wait states) and identifies the data transfer clock cycle for memory write transactions.

nmiack_tran (output, active-High) The NMI Acknowledge Transaction signal is activated to identify an interrupt acknowledge bus transaction for a Non-Maskable Interrupt request.

nmi_addr (input, 16-bit bus) The Non-Maskable Interrupt Address bus provides the logical address of the NMI service routine. The original Z80/Z180 vectored an NMI request to address 0x0066, so this address should be supplied on this bus if compatibility is required.

nmi_req (input, active-High) The Non-Maskable Interrupt Request signal unconditionally interrupts the CPU. This request is internally latched, so that it can be as short as one clock cycle wide.

ref_clock (input, asynchronous) The Reference Clock signal provides an external time reference for an operating system. The Real Time Interval is automatically synchronized to this signal. If this signal is not present the **rti_default** value is used to set the period for the **rti_pulse** signal. If the **ref_clock** signal is lost, the last automatically-generated value continues.

reset_bus (input, 2-bit bus, asynchronous edge-triggered) This bus consists of two independent Reset Request signals. Each of these signals is digitally filtered to detect a transition. A valid transition on either signal (where the level change persists for at least eleven clock cycles) causes a User Reset.

resetb (output, active-Low) The User Reset signal is generated in the System Status block from the various possible reset sources (**reset_bus**, **fatal_detect**, **fault_detect** and **wdt_timeout**). This signal is synchronous with the falling edge of **clkc** and is guaranteed to be at least three clock cycles wide.

rti_default (input, 24-bit value) The Real Time Interval Default value sets the period for the **rti_pulse** signal in the case there is no **ref_clock** or when the automatic **rti_pulse** generation is disabled. This value will be used until two valid **ref_clock** transitions have been recognized, at which time the automatically-generated period will be used.

rti_divisor (input, 7-bit value) The Real Time Interval Divisor value sets the number of **rti_pulse** signals during one period of the **ref_clock** signal. This value is used directly, and a zero value disables the automatic generation of the **rti_pulse** signal.

rti_pulse (output, active-High) The Real Time Interval Pulse signal is active for one clock cycle at the Real Time Interval rate.

rti_synced (output, active-High) The Real Time Interval Synced signal is active while the automatic **rti_pulse** generation is active.

sleep_tran (output, active-High) The Sleep Transaction signal is activated by the Sleep instruction. While in the Sleep state the CPU freezes and waits for an interrupt. The **iack_tran**, **io_addr_out**, **io_data_out**, **io_tran**, **mem_addr_out**, **mem_data_out**, **mem_tran**, **nmiack_tran**, **reti_tran** and **t1** signals are all inactive (Low) during this time.

t1 (output, active-High) The T1 signal is active during the first clock cycle of a bus transaction. This signal is inactive during the Halt and Sleep states. It is also inactive while **dma_ack** is active.

wait_iack (input, active-High) The Interrupt Acknowledge Wait Request signal temporarily halts the processor during interrupt acknowledge transactions. This signal is sampled only during interrupt acknowledge transactions.

wait_io (input, active-High) The I/O Wait Request signal temporarily halts the processor during I/O transactions. This signal is sampled only during I/O transactions.

wait_mem (input, active-High) The Memory Wait Request signal temporarily halts the processor during memory transactions. This signal is sampled only during memory transactions.

wait_timeout (output, active-High) The Wait Request Time-out signal is activated whenever the clock cycle limit for Wait states is met. External logic may use this information to accumulate performance information, or as a fault indicator.

wdt_lim (input, 8-bit value) The Watch-Dog Timer Limit value sets the time-out period, in multiples of 2^20 clock cycles, for the Watch-Dog Timer. This value can be either static or controlled by an external I/O port. A value of "n" selects "n+1" multiples of 2^20 clock cycles for the time-out period. This value is sampled by the WDRES instruction.

wr_inh (input, active-High) The Write Inhibit signal is sampled during the **t1** time of all memory write transactions, and if active the **mem_wr** signal is suppressed during the transaction. A trap is also generated. This is accomplished by jamming an 0xC7 (RST 0) on the data bus into the CPU in response to the next instruction fetch.

External Timing

The Y90 CPU uses a uniform two-clock-cycle machine cycle. This consistent timing simplifies the design of logic external to the CPU makes it easier to track the state of the CPU.

The memory interface timing and signals are designed to make it easy to interface to standard ASIC and FPGA memories. It uses separate read and write strobes.

The I/O interface is very close to the AMBA Peripheral Bus (APB) to allow connection to APB peripherals with a minimum of logic. It uses a single strobe with a separate direction control. The only difference relative to the APB is the setup time for the write data. In the APB the write data is setup one clock before the strobe; in this interface the write data changes coincident with the leading edge of the strobe. In most cases this will not be a problem.

The separate interrupt vector bus provides an easy way to connect to the optional interrupt controller. The interrupt vector bus is used for Mode 0 and Mode 2 maskable interrupts, so if these modes are not used the vector input bus can be tied to ground and the vector strobe output ignored.

In the diagrams below only the relevant signals are shown for each transaction. All other signals are either inactive or hold the previous value. Note that only one of the transaction identifiers (**mem_tran, io_tran, iack_tran, nmiack_tran, reti_tran, halt_tran** and **sleep_tran**) can be active at a time. If all are inactive, an idle bus transaction (usually for address calculation) is in progress. If prefetch is enabled most address calculation idle transactions are replaced by memory transactions. The **dma_ack** signal also indicates that the bus is idle, in response to the **dma_req** signal. The **dma_ack** signal can be active while either **halt_tran** or **sleep_tran** is active.

The **wait_req** input is only sampled for memory, I/O and interrupt acknowledge transactions and is ignored in all other cases. Wait states will disrupt the two-clock-cycle machine cycle rule. If this feature is important but wait states must be used, two wait states per transaction is recommended. If memory access time is an issue it might be better to stretch the first clock cycle of a transaction rather than add Wait states. The uniform two-clock machine cycle makes it relatively straightforward to do this.

Memory Read

The figure below shows the memory read transaction, without Wait states and with one Wait state. Memory read transactions are used for both instruction and data fetch. There is no separate instruction/data status indicator, although this status exists internally if it is needed.
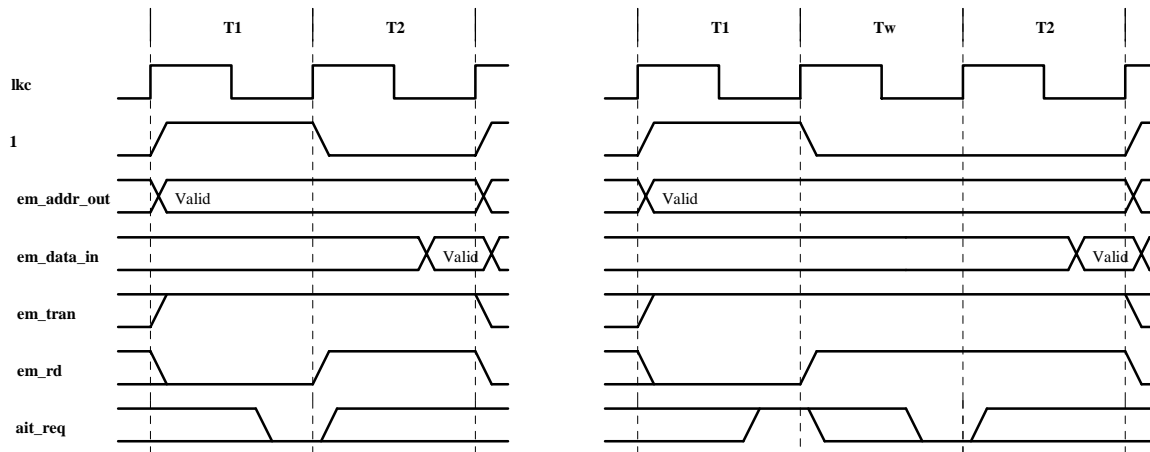
Memory Write

The figure below shows the memory write transaction, without Wait states and with one Wait state.
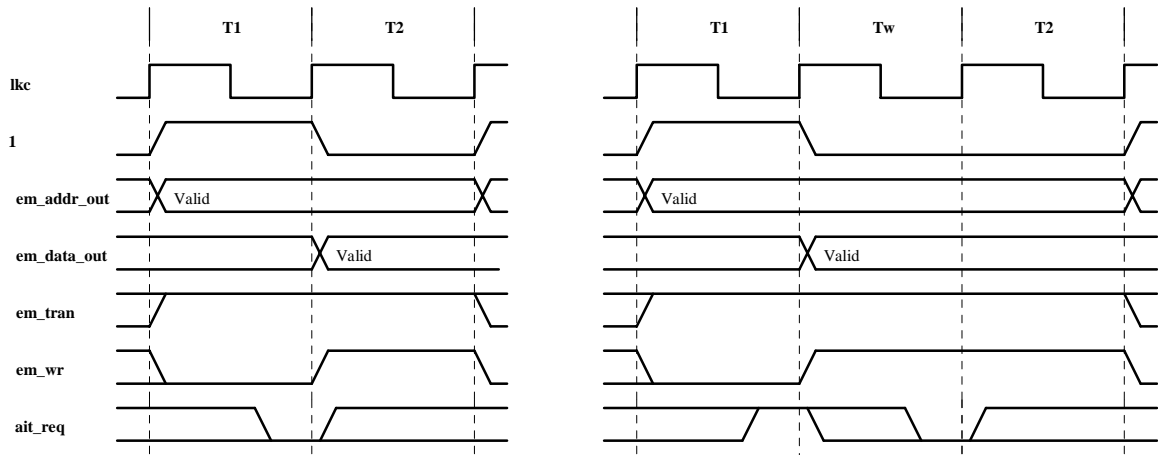
I/O Read

The figure below shows an I/O read transaction, without Wait states and with one Wait state.
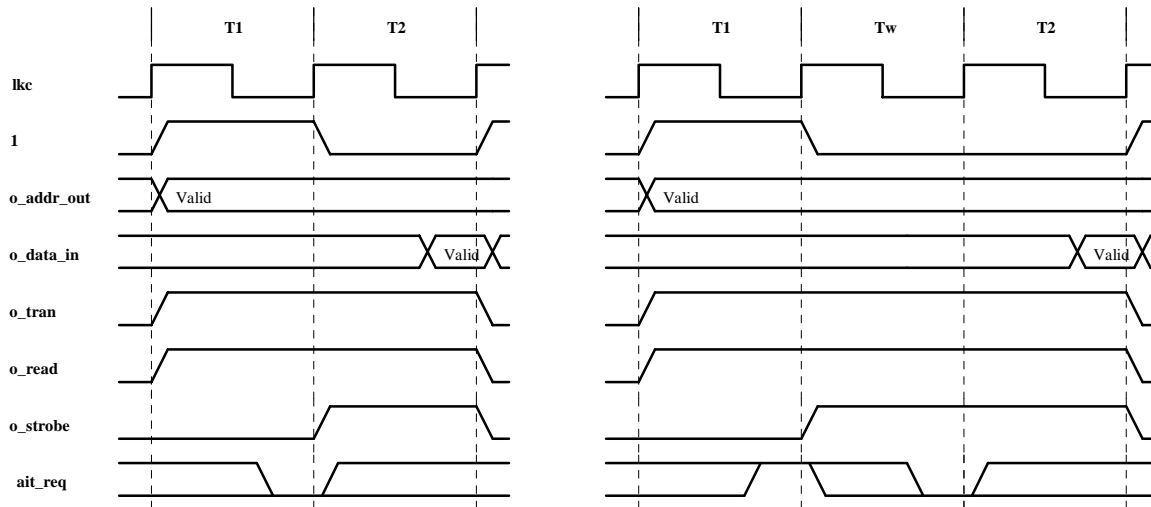
I/O Write

The figure below shows an I/O write transaction, without Wait states and with one Wait state.

	T1	T2		T1	Tw	T2

lkc

1

o_addr_out — Valid ... Valid

o_data_out — Valid ... Valid

o_tran

o_tread

o_strobe

ait_req
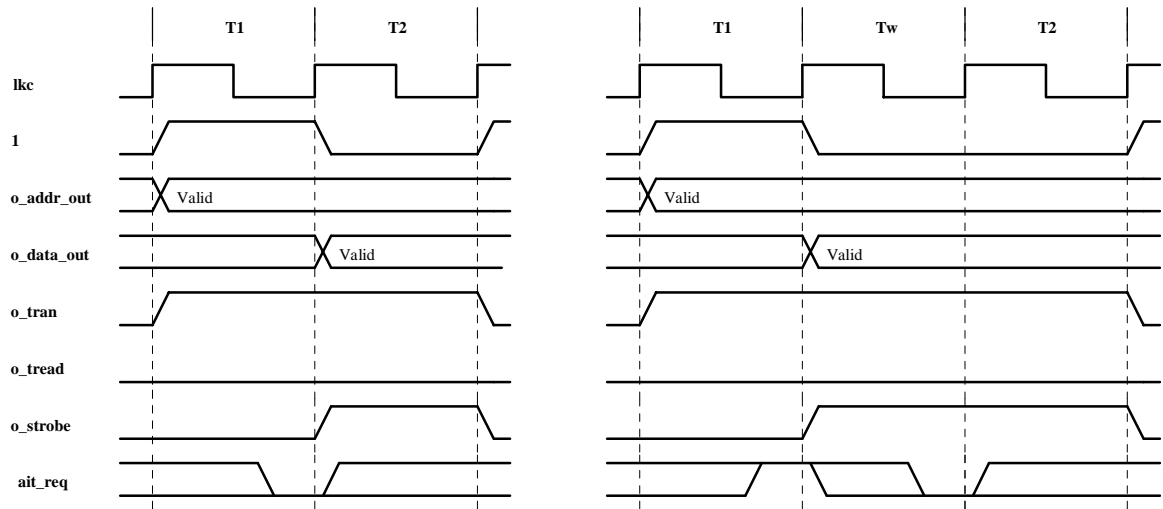
Interrupt Acknowledge

The figure below shows the interrupt acknowledge transaction, without Wait states and with one Wait state.
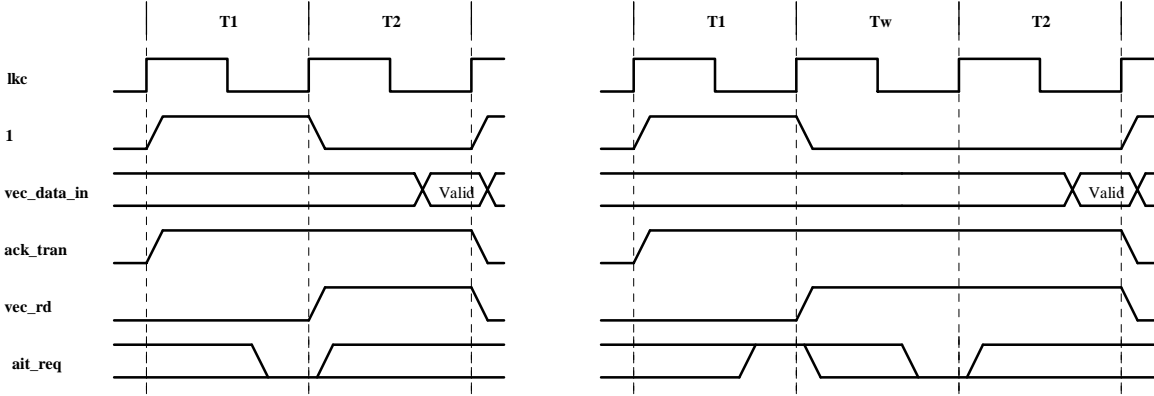
Prefetch

The figure below shows a typical instruction (a memory write) without the prefetch enabled and with the prefetch enabled. The prefetch logic uses address calculation machine cycles to look at the next opcode byte. If this opcode byte is one of the "prefix" bytes (0xCB, 0xDD, 0xED or 0xFD) the logic buffers this byte and will not re-fetch it when the current instruction completes. Only these four prefix bytes will be buffered, even though there are other multi-byte opcodes. Attempting to prefetch for every multi-byyte opcode would be significantly more complicated, with marginal performance improvement.

In practice, enabling the prefetch can improve execution time by about 5%, although this obviously depends on the exact code being executed. If Wait states are being used prefetch may not provide any performance gain, because of the Wait states added when prefetching bytes that may later be discarded.

Note that even though an instruction may execute faster when the prefetch is enabled, this instruction will still complete at the same time. However, the next instruction (the one with the first byte prefetched) will complete earlier. This is shown in the diagrams below.

The prefetch can be enabled and disabled on the fly, because the **en_prftch** signal is sampled during the **t1** time of the fetch of the first byte of an instruction.

If **exec_inh** is sampled active during the prefetch the byte is ignored. This causes the byte to be fetched normally at the end of the current instruction and the normal execute inhibit operation (substitution of RST 0) to commence.

Top diagram:

T2 | T1 | T2 | T1 | T2 | T1 | T2 | T1 | T2 | T1 | T2 | T1 | T2 | T1

clkc

mem_addr_out: PC | PC+1 | PC+2 | mem addr | PC+2 | PC+3 | PC+4

mem_data_in: execute | execute | Ignored | execute | execute

mem_data_out: write data

t1

mem_tran

mem_rd

mem_wr

inst at PC

inst at PC+2

inst at PC+4

Bottom diagram:

T2 | T1 | T2 | T1 | T2 | T1 | T2 | T1 | T2 | T1 | T2 | T1 | T2 | T1

clkc

mem_addr_out: PC | PC+1 | PC+2 | mem addr | PC+3 | PC+4 | PC+5

mem_data_in: execute | execute | buffer | execute | execute

mem_data_out: write data

t1

mem_tran

mem_rd

mem_wr

inst at PC

inst at PC+2

inst at PC+4

26

Illegal Instruction (2nd byte) Trap

The timing of an undefined second byte opcode trap is shown below. The fetch of the undefined opcode is followed by three machine cycles that flush the pipeline and rewind the Program Counter, an interrupt acknowledge with the **inst2_trap** signal active, and two writes to push the PC of the undefined opcode to the stack. The processor then jumps to location 0x0000 and starts fetching instructions.
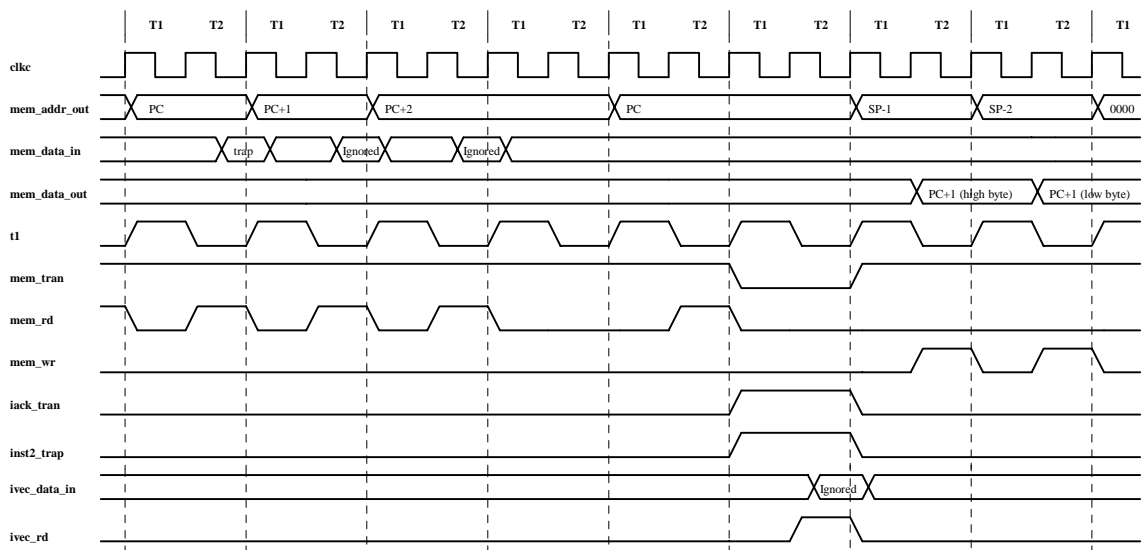
In the case of the Y90 CPU the **inst2_trap** information should be latched outside the CPU to distinguish this case from both the **inst3_trap** case and the normal reset case. The start of the illegal instruction in this case is the stacked PC value minus one.

The Y90 MPU latches the **inst2_trap** information as well as both the logical and physical address of the offending byte. This information is available in the System Status Block.

Illegal Instruction (3rd byte) Trap

The timing of an undefined third byte opcode trap is shown below. The fetch of the undefined opcode is followed by the normal Read cycle (all three-byte instructions use indexed addressing), two machine cycles that flush the pipeline and rewind the Program Counter, an interrupt acknowledge with the **inst3_trap** signal active, and two writes to push the PC of the undefined opcode to the stack. The processor then jumps to location 0x0000 and starts fetching instructions.

	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1
clkc																	
mem_addr_out	PC+2		PC+3		IX+d or IY+d		PC+4		PC+2				SP-1		SP-2		0000
mem_data_in		disp		trap		Ignored											
mem_data_out													PC+2 (high byte)		PC+2 (low byte)		
t1																	
mem_tran																	
mem_rd																	
mem_wr																	
iack_tran																	
inst3_trap																	
ivec_data_in											Ignored						
ivec_rd																	

In the case of the Y90 CPU the **inst3_trap** information should be latched outside the CPU to distinguish this case from both the **inst2_trap** case and the normal reset case. The start of the illegal instruction in this case is the stacked PC value minus two.

The Y90 MPU latches the **inst3_trap** information as well as both the logical and physical address of the offending byte. This information is available in the System Status Block.

Execute Inhibit Trap

The timing of an Execute Inhibit trap is shown below. The fetch of the first byte of an opcode in an execute-protected page causes the CPU to execute an RST 0 instruction in place of the offending instruction. This means there is no interrupt acknowledge transaction associated with the Execute Inhibit trap. Software should use the address latched in the System Status Block to determine the source of the trap, because the stack contents point to the byte after the one that caused the trap.
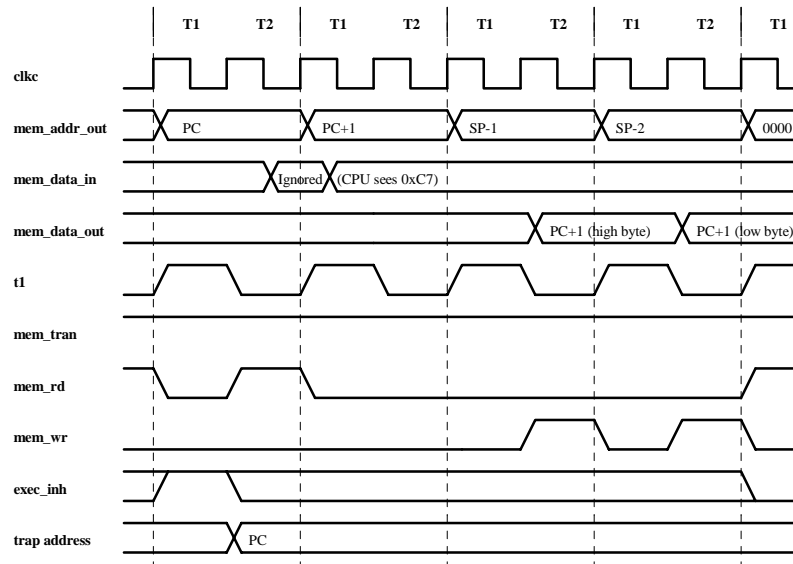
Execute Inhibit Trap & Fatal Error

The **exec_inh** signal should never be active for an address is Page 0. This will lead to a fatal error condition because the response to the trap is to jump to address 0x0000, leading to an infinite loop of traps. The Y90-MPU activates the **fatal_detect** signal in the case of an Execute Inhibit trap in Page 0, updates the System Status Block and generates the **resetb** signal. The timing of the **fatal_detect** signal is shown in the diagram below.
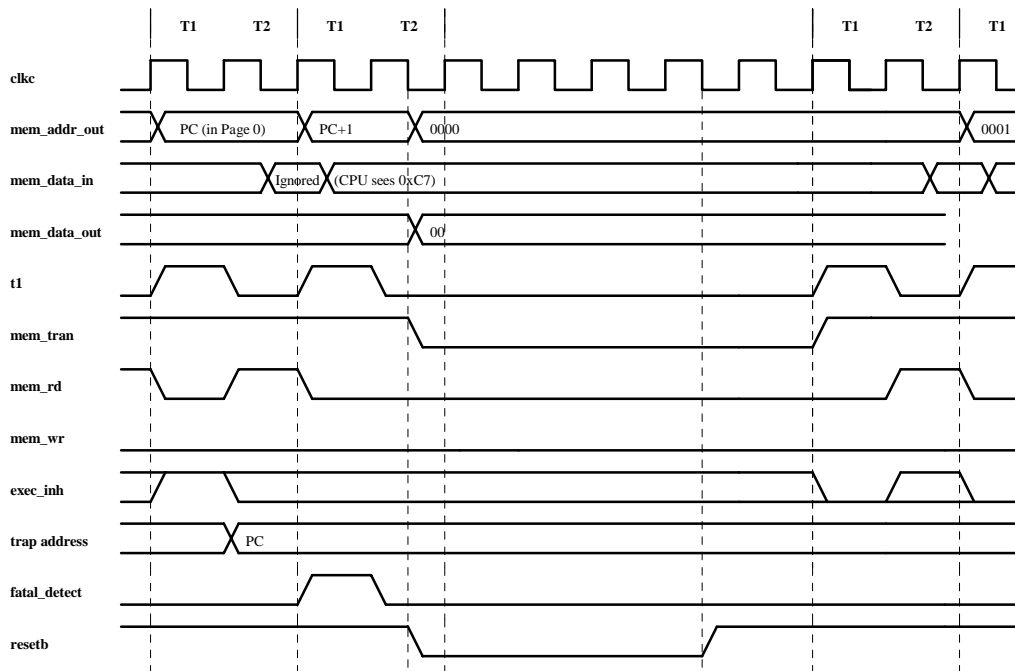
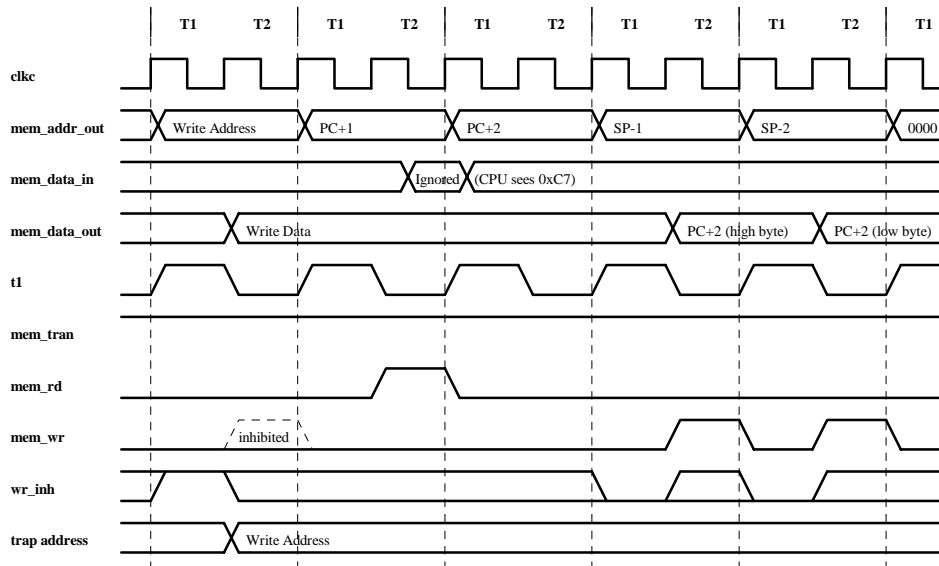
It is assumed here that the reset will clear whatever condition lead to signalling **exec_inh** during a Page 0 access. If this is not the case an infinte loop of traps can still result.

Write Protect Trap

The timing of a Write Protect trap is shown below. The attempt to write to a write-protected page causes the CPU to execute an RST 0 instruction in place of the next instruction. This means there is no interrupt acknowledge transaction associated with the Write Protect trap. Software should use the address latched in the System Status Block to determine the source of the trap, because the stack contents point to an address in the instruction after the one that caused the trap.

|	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1
clkc											
mem_addr_out	Write Address		PC+1		PC+2		SP-1		SP-2		0000
mem_data_in					Ignored / (CPU sees 0xC7)						
mem_data_out		Write Data						PC+2 (high byte)		PC+2 (low byte)	
t1											
mem_tran											
mem_rd											
mem_wr		inhibited									
wr_inh											
trap address		Write Address									

Write Protect Trap & Fatal Error

The **wr_inh** signal should never be active for a stack write. This will lead to a fatal error condition because the response to the trap is to push an address to the stack and jump to address 0x0000, leading to an infinite loop of traps. The Y90-MPU activates the **fatal_detect** signal in the case of a Write Protect trap during a stack write, updates the System Status Block and generates the **resetb** signal. The timing of the **fatal_detect** signal is shown in the diagram below.
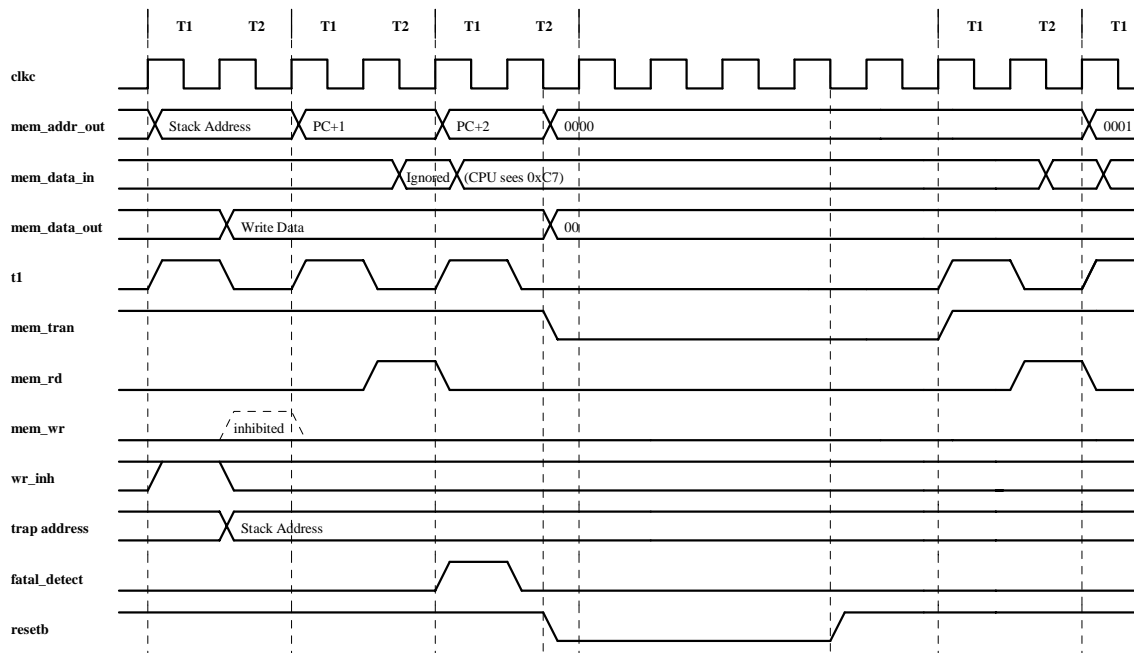
It is assumed here that the reset will clear whatever condition lead to signalling **wr_inh** during a stack write. If this is not the case an infinite loop of traps can still result.

Non-maskable Interrupt

The timing of a non-maskable interrupt acknowledge transaction is shown below. The **nmi_req** input cannot be masked by software. This input must be sampled active by a rising edge of **clkc** to be recognized by the processor, but does not need to remain active until the interrupt acknowledge transaction. In fact, to prevent an endless loop of acknowledges, the **nmi_req** input must be de-asserted before the start of the fetch of the first instruction of the service routine. The acknowledge sequence consists of an aborted instruction fetch, the interrupt acknowledge, and two writes to push the contents of the program counter onto the stack. Execution then begins at the address provided on the **nmi_addr** bus (shown as 0x0066 in the figure) with an instruction fetch. The non-maskable interrupt service routine must end with the RETN instruction to properly restore the state of the interrupt enable flag prior to the non-maskable interrupt.
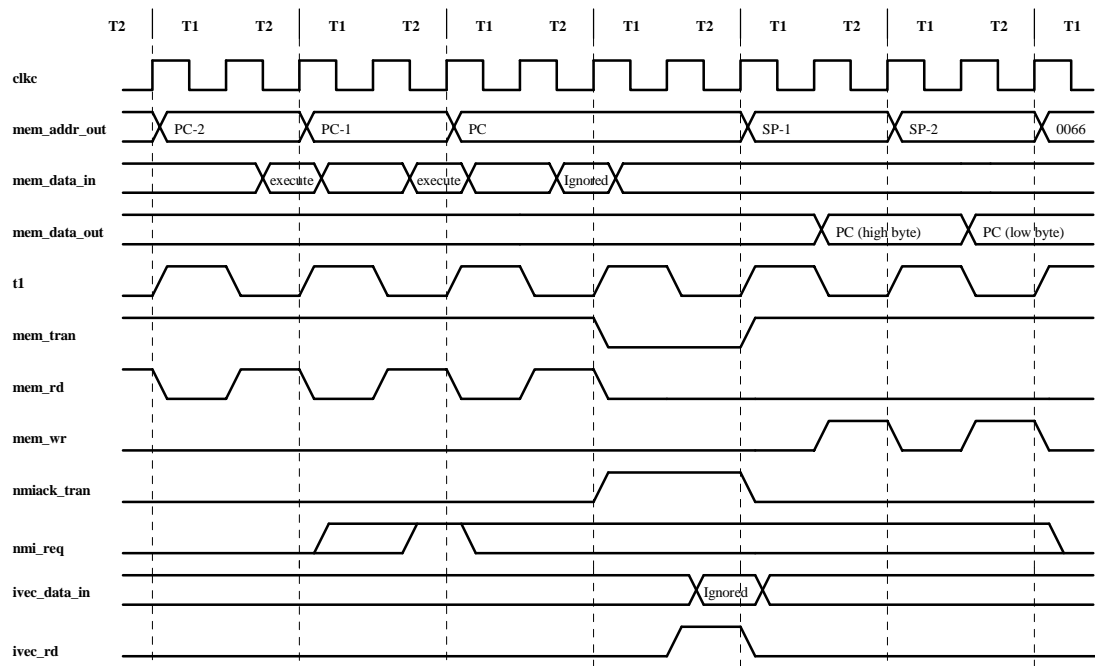
Interrupt Mode 0

The timing of a Mode 0 maskable interrupt acknowledge is shown below. The **int_req** signal needs to remain active until the interrupt acknowledge transaction. The acknowledge sequence consists of an aborted instruction fetch, the interrupt acknowledge, and two writes to push the contents of the program counter onto the stack. Execution then begins at the restart address specified by the RST instruction fetched during the interrupt acknowledge with an instruction fetch.
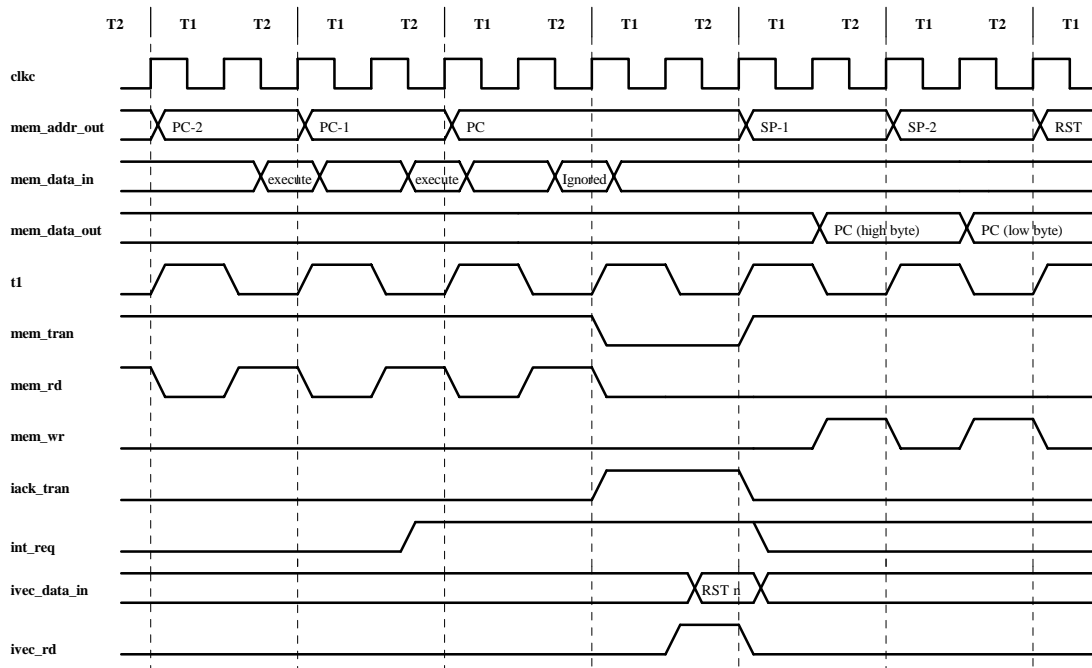
The use of an RST instruction is enforced by the hardware, which only uses bits 5-3 of the **ivec_data_in** bus to create one of the eight possible RST instructions.

Interrupt Mode 1

The timing of a Mode 1 interrupt acknowledge cycle is shown below. The **int_req** input needs to remain active until the interrupt acknowledge transaction. The acknowledge sequence consists of an aborted instruction fetch, the interrupt acknowledge, and two writes to push the contents of the program counter onto the stack. Execution then begins at address 0x0038 with an instruction fetch.
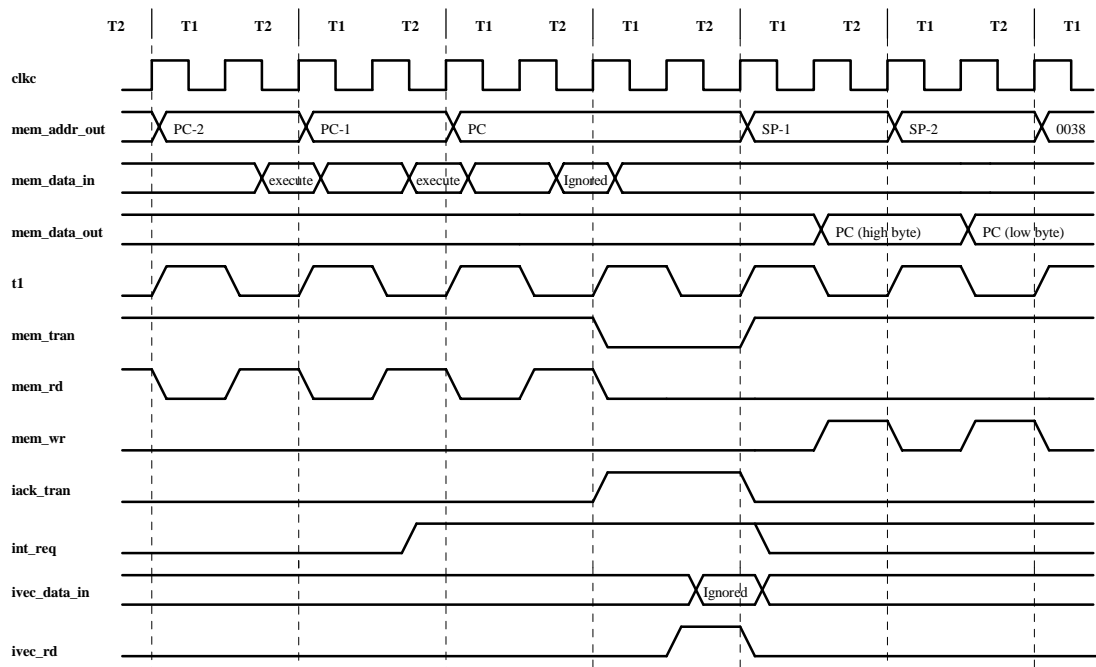
Interrupt Mode 2

The timing of a Mode 2 maskable interrupt acknowledge is shown below. The **int_req** input needs to remain active until the interrupt acknowledge transaction. The acknowledge sequence consists of an aborted instruction fetch, the interrupt acknowledge, an address calculation cycle, two reads of the interrupt vector table and two writes to push the contents of the program counter onto the stack. The processor automatically jumps to the address fetched from the interrupt vector table for the service routine. The upper eight bits of the interrupt vector table starting address are held in the I register in the processor. Note that the vector must be an even number. That is, the least significant bit of the vector must be a zero.
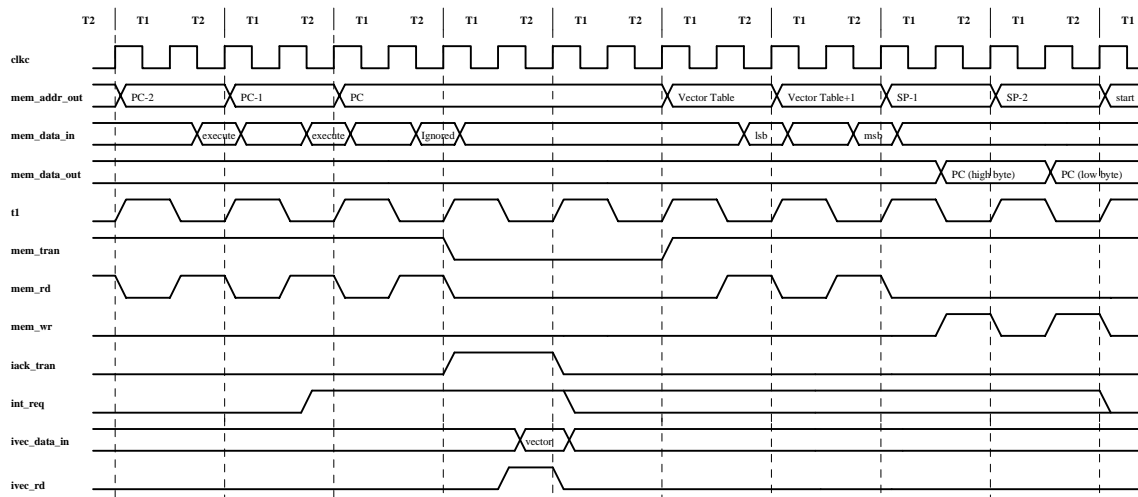
The interrupt controller in the Y90 MPU necessarily samples the **int_req_bus** inputs, and then goes into the edge-detect latch, which changes the timing slightly. The diagram below illustrates this change for Interrupt Mode 2. Also shown is the timing of the **int_ack_bus** and the **int_prio_out** signals.
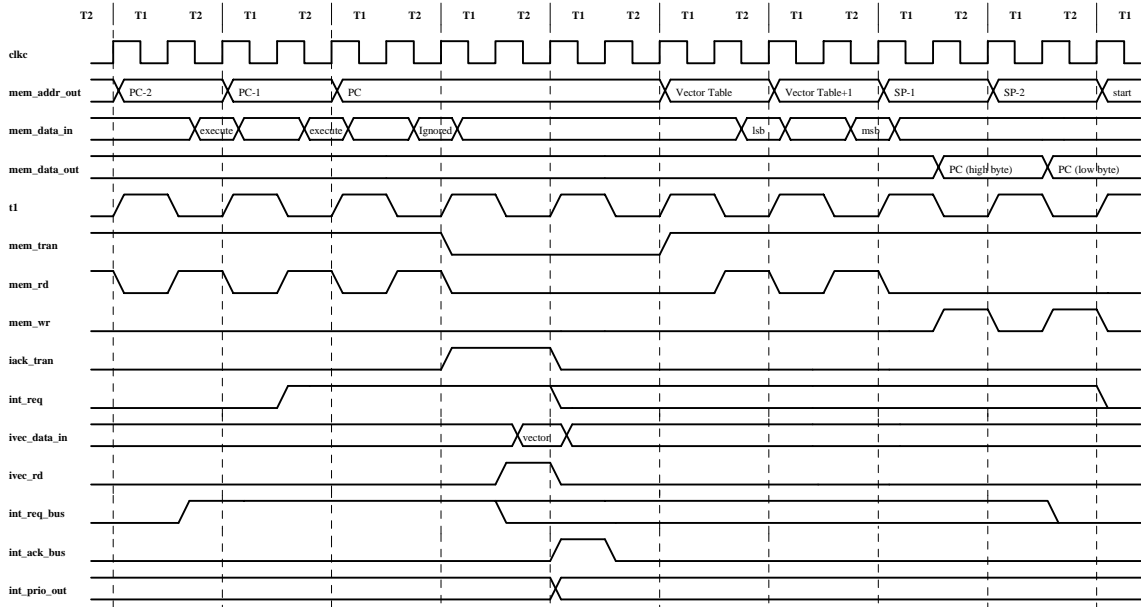
Although Interrupt Mode 2 is the preferred mode for use with the interrupt controller, the design allows the use of any interrupt mode. In Interrupt Mode 0 or 1 it will be necessary to use the **int_ack_bus** signals to externally latch the information about which interrupt is being acknowledged. This is because Interrupt Modes 0 and 1 will branch to a common interrupt service routine, rather than the individual routines possible in Interrupt Mode 2.

DMA Request/Acknowledge

The timing of a DMA request and acknowledge is shown below. Note that like an interrupt, the **dma_req** signal is only sampled at the end of instructions. This guarantees that all instructions are atomic.

The delay from the **dma_req** signal to the **dma_ack** signal is always at least one bus cycle, irrespective of whether the processor is running, in the Halt state or in the Sleep state. This implies that it is more efficient to transfer multiple bytes each time that the **dma_req** signal is activated.

The **dma_req** signal can be asserted during the Halt or Sleep states. In this case the active **dma_req** signal will take precedence over **int_req** or **nmi_req** and inhibit either of these signals from causing an exit from the Halt or Sleep state. Once the **dma_req** signal is deasserted any pending or future interrupt request will cause the exit from the Halt or Sleep state.

The DMA request controller in the Y90 MPU necessarily samples the **dma_req_bus** inputs, which changes the timing slightly. The diagram below illustrates this change. Also shown is the tim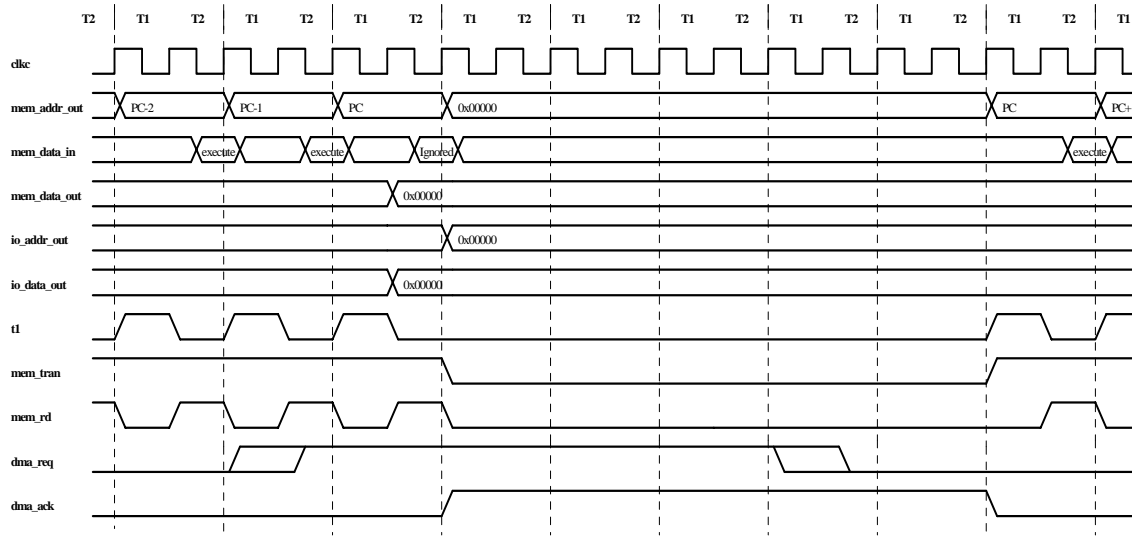ing of the **dma_ack_bus** and the **dma_prio_out** signals. In particular, note that the leading edge of the **dma_ack_bus** signals are delayed by one clock cycle from the normal **dma_ack** timing. The timing of the trailing edge of these signals is not affected.
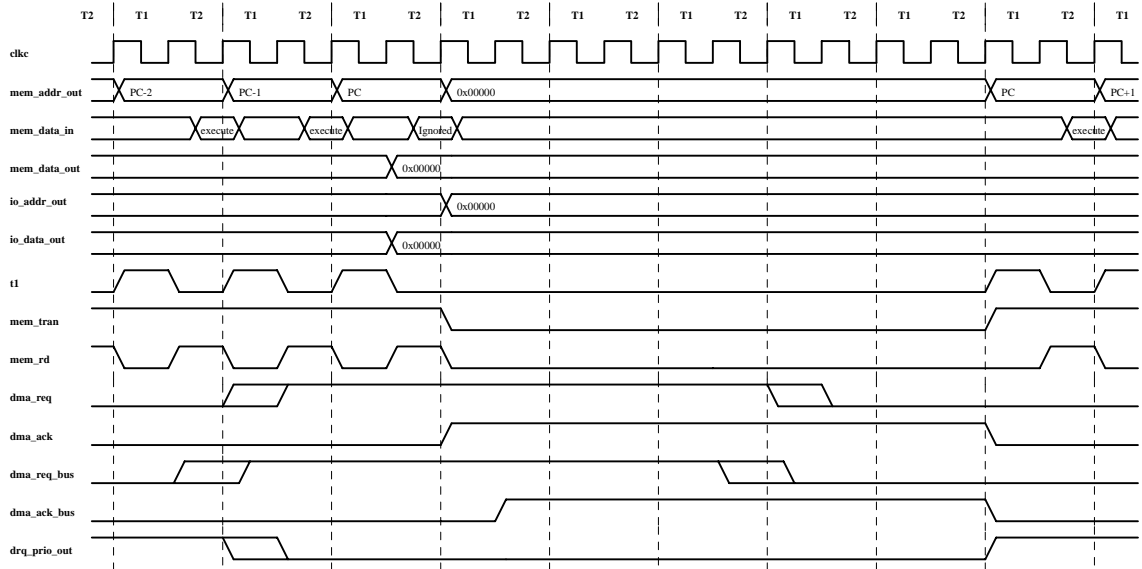
Halt state

The Halt state is entered when the HALT instruction is executed, as shown below. In the Halt state the processor freezes, for an unlimited number of two clock cycle machine cycles, with the **halt_tran** output active. The only way to exit the Halt state is with either an interrupt (either **nmi_req** or **int_req**) or via reset. Note that **int_req** can only be used to exit the Halt mode if interrupts are enabled when the HALT instruction is executed. The timing for exiting the Halt state with an interrupt is also shown below.

If the Halt state is exited by an interrupt, the processor will resume instruction execution (after the interrupt service routine) at the address of the instruction following the HALT instruction. The minimum width of the **halt_tran** signal is two clock cycles.
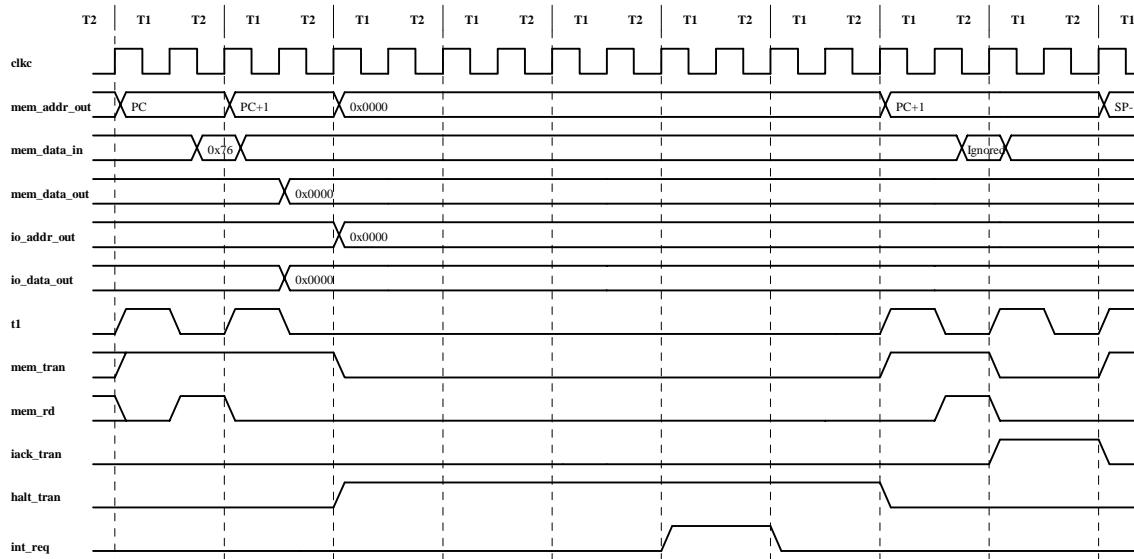
The Halt state in this design is slightly different from that in the Z80 or Z180 microprocessors. In those designs the processor continues to fetch the Halt instruction during the Halt state, leading to continued power dissipation. Since this operation requires the special step of "rewinding" the PC, no attempt was made to match this operation. Rather, the Halt state and the Sleep state are essentially identical, reducing the power consumption to a minimum by minimizing the number of signals that are transitioning during these states.

The Halt state differs from the Sleep state only for the case where interrupts are disabled. In the Halt state, if interrupts are disabled only the **nmi_req** or a reset (from any of the various sources) will cause an exit from this state. In the Sleep state if interrupts are dis-

abled a rising edge on the **int_req** will force an exit from the Sleep state, with execution continuing with the instruction following the SLP instruction.

Sleep state

The Sleep state is entered when the SLP instruction is executed, as shown below. In the Sleep state the processor freezes, for an unlimited number of two clock cycle machine cycles, with the **sleep_tran** output active. The only way to exit the Sleep state is with either an interrupt (either **nmi_req** or **int_req**) or via reset. The **int_req** signal can be used to exit the Sleep mode irrespective of whether or not interrupts are enabled when the SLP instruction is executed.

The timing for exiting the Sleep state with an enabled interrupt or non-maskable interrupt is shown below. In this case the processor will resume instruction execution (after the interrupt service routine) at the address of the instruction following the SLP instruction.
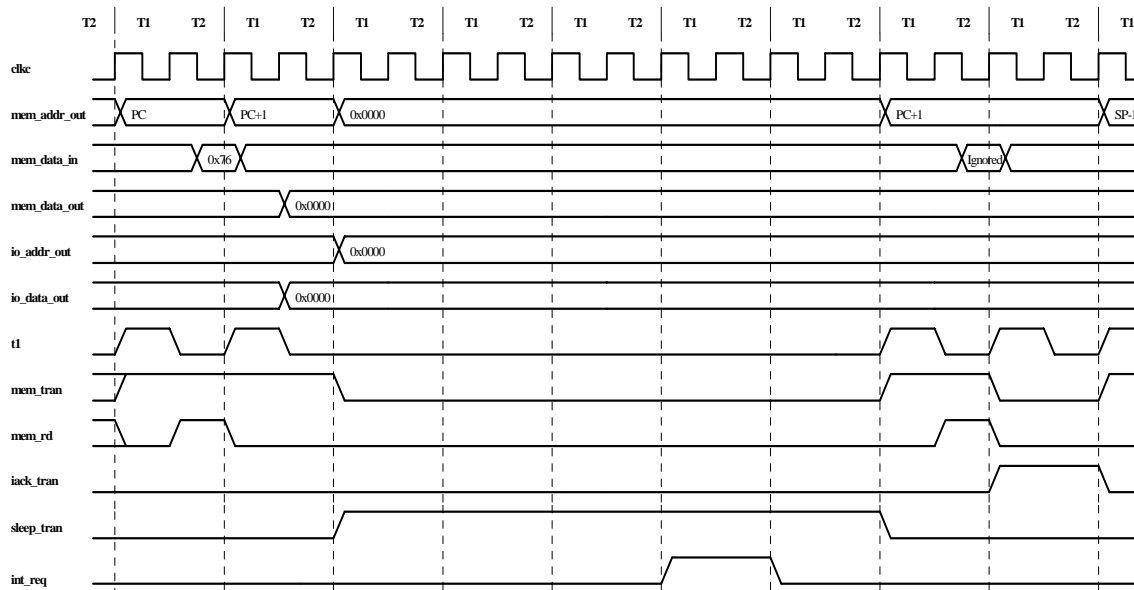
In the case where the Sleep state exit is caused by a maskable interrupt while interrupts are disabled the processor merely resumes execution at the address of the instruction following the SLP instruction, without going through an interrupt service routine. Note that the minimum width of the **sleep_tran** signal is two clock cycles.
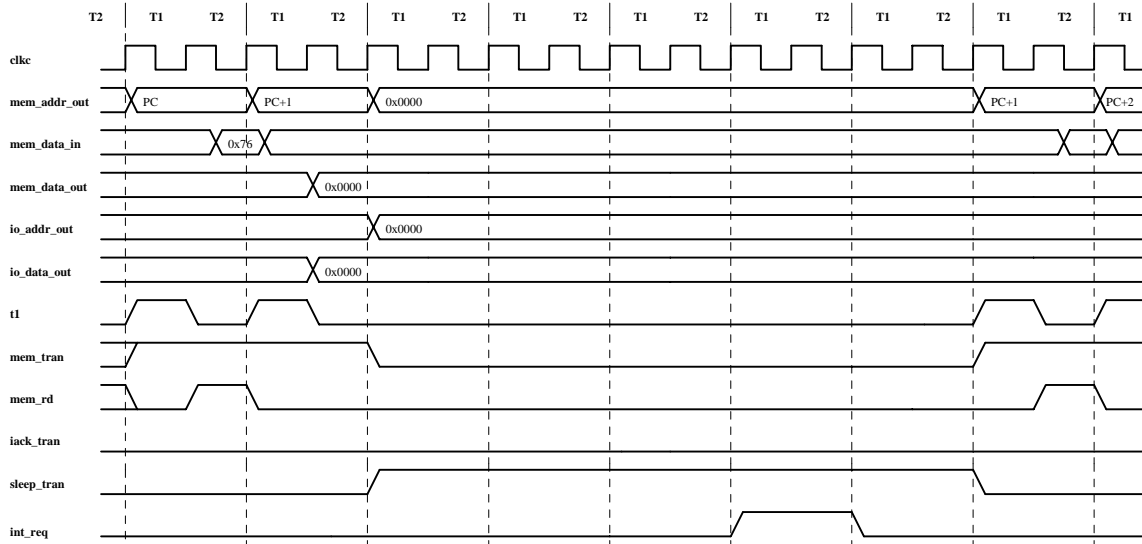
Reset

The Reset state is entered immediately when the **resetb** signal goes Low, independent of the current state, and this state continues until the first rising edge of **clkc** after the **resetb** signal is de-asserted. At this rising edge there is a one clock cycle transient state to set up the internal pipeline controls, and on the next clock the processor begins fetching the first instruction from address 0x0000.

Software starting at location 0x0000 must be able to distinguish between reset, execution of an RST 0 instruction, a trap, or watch-dog time-out. All of these cases cause the Program Counter to be reset to 0x0000. In the case of the Y90 MPU this information is available in the System Status Block.

The minimum width of the **resetb** signal is set by the flip-flops used in the design. The setup time for the **resetb** signal to the rising edge of the **clkc** signal is likewise determined by the flip-flops used in the design.

The **clearb** signal has the same timing requirements as the **resetb** signal. The **clearb** signal should only be used in the power-on case, and only affects those flip-flops not affected by the **resetb** signal.
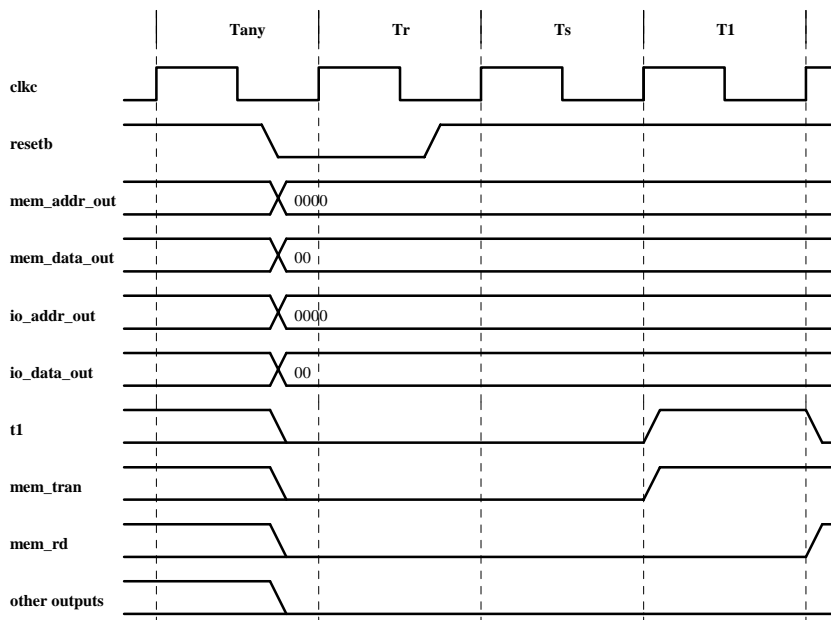
In the case of the Y90 MPU the **resetb** signal is an output, generated from the **reset_bus** input signals or internal error conditions. The **reset_bus** input signals are synchronized and filtered to generate the **resetb** signal synchronous with the falling edge of the **clkc** signal.

A transition, of either polarity, on either **reset_bus** signal, will generate the **resetb** signal. But this change-of-state on a **reset_bus** signal must persist for at least ten clock cycles for the **resetb** signal to be generated. Any pulse shorter than ten clock cycles will not pass the digital filter. The timing for the generation of the **resetb** signal is shown in the diagram below.
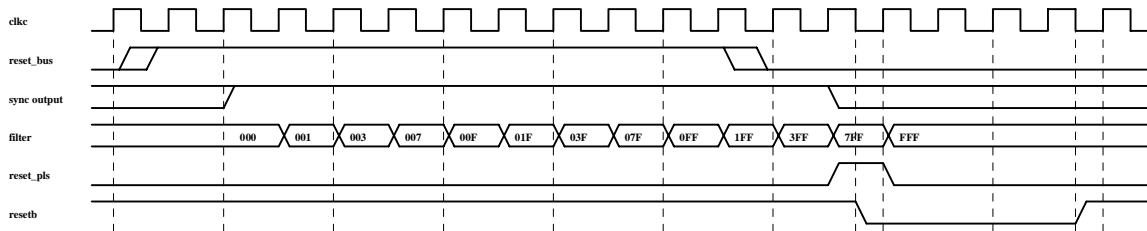
Base Instruction Set

This chapter presents the assembly language syntax, addressing modes, flag settings, binary encoding, and execution time for the base Y90 instruction set. The entire instruction set is presented in alphabetical order.

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

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

The effect of the instruction on each flag is listed, with a brief description. Normally the flags are updated by the main operation of the instruction, but for some complex instructions different flags may be affected by different parts of the instruction. This is specified in the description. The flags are organized as below in the F (Flag) register:

S	Z	U5	H	U3	P/V	N	C

These flags have the following meanings:

Flag	Meaning
S	Sign (a copy of the MSB of the result).
Z	Zero (indicating that the result was zero).
U5	Unused Bit 5 (an unused Flag register bit).
H	Half-Carry (carry out of the lower nibble, used for BCD math).
U3	Unused Bit 3 (an unused Flag register bit).
P/V	Parity/Overflow (parity of the result, or arithmetic overflow; depends on the instrcuction)
N	Negative (add/subtract flag, necessary for BCD math)
C	Carry (arithmetic carry, or shift linkage bit)

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 field in the instruction specifies a CPU register, employing the following encoding:

rrr	Register Selected
000	B
001	C
010	D
011	E
100	H
101	L
111	A (Accumulator)

Word registers are similarly encoded, although the exact encoding depends on the instruction:

dd, ss, tt, xx or yy	dd, ss Register	tt Register	xx Register	yy Register
00	BC	BC	BC	BC
01	DE	DE	DE	DE
10	HL	HL	IX	IY
11	SP	AF	SP	SP

The execution time for instructions is always a multiple of two clocks. Any number in parentheses is the execution time when the prefetch is enabled, via the **en_prftch** signal into the core. When enabled, the prefetch operation uses any address calculation time to look at the first byte of the next instruction. If this instruction byte can be pre-decoded the byte will be buffered for use when the current instruction finishes. This results in the execution time in parentheses. Only instructions that require more than one machine cycle to execute can be pre-decoded.

ADC

Add With Carry

ADC A, src   src: R, IM, IR, X

Operation: A <= A + src + CF

Flags: **S:** Set if result is negative; cleared otherwise.

Z: Set if result is zero; cleared otherwise.

H: Set if arithmetic carry out of bit 3; cleared otherwise.

P/V: Set if arithmetic overflow; cleared otherwise.

N: Cleared.

C: Set if arithmetic carry out of bit 7; cleared otherwise.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	ADC A, r	10001rrr	2
IM:	ADC A, n	11001110 ----n---	4
IR:	ADC A, (HL)	10001110	6 (4)
X:	ADC A, (IX+d) or ADC A, (IY+d)	11y11101 10001110 ----d---	10 (8)

Notes:

1. The **rrr** field uses the standard register select encoding

2. **y** = 0 selects IX and **y** = 1 selects IY

ADC

Add With Carry (Word)

ADC HL, src   src: RR

Operation: HL <= HL + src + CF

Flags: **S:** Set if result is negative; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
H: Set if arithmetic carry out of bit 11; cleared otherwise.
P/V: Set if arithmetic overflow; cleared otherwise.
N: Cleared.
C: Set if arithmetic carry out of bit 15; cleared otherwise.

Addressing Modes	Assembly Syntax	Encoding	Clocks
RR:	ADC HL, ss	11101101 01ss1010	4

Notes:

1.   The **ss** field uses the standard word register encoding.

ADD

Add

ADD A, src   src: R, IM, IR, X

Operation: A <= A + src

Flags: **S:** Set if result is negative; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
H: Set if arithmetic carry out of bit 3; cleared otherwise.
P/V: Set if arithmetic overflow; cleared otherwise.
N: Cleared.
C: Set if arithmetic carry out of bit 7; cleared otherwise.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	ADD A, r	10000rrr	2
IM:	ADD A, n	11000110 ----n---	4
IR:	ADD A, (HL)	10000110	6 (4)
X:	ADD A, (IX+d) or ADD A, (IY+d)	11y11101 10000110 ----d---	10 (8)

Notes:

1. The **rrr** field uses the standard register select encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY

ADD

Add (Word)

ADC dst, src   dst: HL, IX, IY
                                                               src: RR

Operation: dst <= dst + src

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Set if arithmetic carry out of bit 11; cleared otherwise.
               P/V: Unaffected.
               N: Cleared.
               C: Set if arithmetic carry out of bit 15; cleared otherwise.

Addressing Modes	Assembly Syntax	Encoding	Clocks
RR:	ADD HL, ss	00ss1001	2
	ADC IX, xx	11011101 01xx1001	4
	ADC IY, yy	11111101 01yy1001	4

Notes:

1. The **ss**, **xx** and **yy** fields use the standard word register select encodings.

AND

Logical AND

AND A, src   src: R, IM, IR, X

Operation: A <= A & src

Flags: **S:** Set if result is negative; cleared otherwise.
                   Z: Set if result is zero; cleared otherwise.
                   H: Set.
                   P/V: Set if parity of result even; cleared otherwise.
                   N: Cleared.
                   C: Cleared.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	AND A, r	10100rrr	2
IM:	AND A, n	11100110 ----n---	4
IR:	AND A, (HL)	10100110	6 (4)
X:	AND A, (IX+d) or AND A, (IY+d)	11y11101 10100110 ----d---	10 (8)

Notes:

1. The **rrr** field uses the standard register select encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY

BIT

Bit Test

BIT b, src   src: R, IR, X

Operation: $Z \Leftarrow \sim src[b]$

Flags: **S:** Unaffected.
Z: Set if tested bit is zero; cleared otherwise.
H: Set.
P/V: Unaffected.
N: Cleared.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	BIT b, r	11001011 01bbbrrr	4
IR:	BIT b, (HL)	10100110 01bbb1110	8 (6)
X:	BIT b, (IX+d) or BIT b, (IY+d)	11y11101 11001011 ----d--- 01bbb110	10

Notes:

1. The **rrr** field uses the standard register select encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY.

3. The **bbb** field uses normal binary encoding.

4. For the original Z80, the **S** and **C** flags are undefined.

CALL

Call Subroutine

CALL dst   dst: DA

Operation: SP <= SP - 2

(SP) <= PC

PC <= dst

Flags: **S:** Unaffected.

Z: Unaffected.

H: Unaffected.

P/V: Unaffected.

N: Unaffected.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
DA:	CALL mn	11001101	10
		----n---	
		----m---	

CALL

Conditional Call Subroutine

CALL cc, dst   dst: DA

Operation: if (cc = true) begin
       SP <= SP - 2
       (SP) <= PC
       PC <= dst
       end

Flags: **S:** Unaffected.
Z: Unaffected.
H: Unaffected.
P/V: Unaffected.
N: Unaffected.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
DA:	CALL cc, mn	11fff100 ----n--- ----m---	10/6 (taken/not taken)

Notes:

1.

Mnemonic	Encoding (fff)	Meaning	Flag case
NZ	000	Non-zero	Z = 0
Z	001	Zero	Z = 1
NC	010	Non-carry	C = 0
C	011	Carry	C = 1
PO	100	Parity Odd	P/V = 0
PE	101	Parity Even	P/V = 1
P	110	Plus	S = 0
M	111	Minus	S = 1

CCF

Complement Carry Flag

CCF

Operation:	CF <= ~CF

Flags:
S: Unaffected.
Z: Unaffected.
H: Copy of previous value of Carry flag.
P/V: Unaffected.
N: Cleared.
C: Set if previous Carry flag was zero; cleared otherwise.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IM:	CCF	00111111	2

Notes:

1. The default operation of the **H** flag for this instruction matches that of the original Z80 CPU. The original Z180 CPU behaves differently, clearing the **H** flag for this instruction. To enable Z180 compatibility, use the 'define Z180_CCF option in the Verilog source code file version.v.

CP

Compare

CP A, src   src: R, IM, IR, X

Operation: A - src

Flags: **S:** Set if result is negative; cleared otherwise.
                 Z: Set if result is zero; cleared otherwise.
                 H: Set if arithmetic borrow out of bit 3; cleared otherwise.
                 P/V: Set if arithmetic overflow; cleared otherwise.
                 N: Set.
                 C: Set if arithmetic borrow out of bit 7; cleared otherwise.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	CP A, r	10111rrr	2
IM:	CP A, n	11111110 ----n---	4
IR:	CP A, (HL)	10111110	6 (4)
X:	CP A, (IX+d) or CP (IY+d)	11y11101 10111110 ----d---	10 (8)

Notes:

1. The **rrr** field uses the standard register select encoding

2. **y** = 0 selects IX and **y** = 1 selects IY

CPD

<div align="right">Compare and Decrement</div>

CPD

Operation:	A - (HL)
	HL <= HL - 1
	BC <= BC - 1

Flags:
S: Set if result of compare is negative, cleared otherwise.
Z: Set if result of compare is zero; cleared otherwise.
H: Set if arithmetic borrow out of bit 3 during compare; cleared otherwise.
P/V: Set if result of BC decrement is non-zero; cleared otherwise.
N: Set.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	CPD	11101101 10101001	10 (8)

CPDR

Compare, Decrement and Repeat

CPDR

Operation: A - (HL)

HL <= HL - 1

BC <= BC - 1

repeat if BC != 0 and A - (HL) != 0

Flags: **S:** Set if result of compare is negative, cleared otherwise.

Z: Set if result of compare is zero; cleared otherwise.

H: Set if arithmetic borrow out of bit 3 during compare; cleared otherwise.

P/V: Set if result of BC decrement is non-zero; cleared otherwise.

N: Set.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	CPDR	11101101 10111001	8 + 4i

Notes:

1. This instruction can be interrupted after each iteration. The address saved on the stack in this case is the address of this instruction, allowing completion of the instruction after the interrupt service routine.

2. Interrupts are sampled during each memory read operation.

CPI

Compare and Increment

Operation: A - (HL)

HL <= HL + 1

BC <= BC - 1

Flags: **S:** Set if result of compare is negative, cleared otherwise.

Z: Set if result of compare is zero; cleared otherwise.

H: Set if arithmetic borrow out of bit 3 during compare; cleared otherwise.

P/V: Set if result of decrementing BC is non-zero; cleared otherwise.

N: Set.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	CPI	11101101 10100001	10 (8)

CPIR

Compare, Increment and Repeat

CPIR

Operation: A - (HL)

HL <= HL + 1

BC <= BC - 1

repeat if BC != 0 and A - (HL) != 0

Flags: **S:** Set if result of compare is negative, cleared otherwise.

Z: Set if result of compare is zero; cleared otherwise.

H: Set if arithmetic borrow out of bit 3 during compare; cleared otherwise.

P/V: Set if result of decrementing BC is non-zero; cleared otherwise.

N: Set.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	CPIR	11101101 10110001	8 + 4i

Notes:

1. This instruction can be interrupted after each iteration. The address saved on the stack in this case is the address of this instruction, allowing completion of the instruction after the interrupt service routine

2. Interrupts are sampled during each memory read operation.

CPL
Complement

CPL

Operation: A <= ~A

Flags: **S:** Unaffected.
Z: Unaffected.
H: Set.
P/V: Unaffected.
N: Set.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	CPL	00101111	2

DAA

Decimal Adjust Accumulator

DAA

Operation: A <= Decimal Adjust A

Flags: **S:** Set if result is negative; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
H: See table below.
P/V: Set if result has even parity; cleared otherwise.
N: Unaffected.
C: See table below.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	DAA	00100111	2

Notes:

Instruction	C before DAA	A[7:4] before DAA	H before DAA	A[3:0] before DAA	Number added to A	C after DAA
	0	0-9	0	0-9	00	0
	0	0-8	0	A-F	06	0
	0	0-9	1	0-3	06	0
ADC, ADD or INC	0	A-F	0	0-9	60	1
	0	9-F	0	A-F	66	1
	0	A-F	1	0-3	66	1
	1	0-2	0	0-9	60	1
	1	0-2	0	A-F	66	1
	1	0-3	1	0-3	66	1
	0	0-9	0	0-9	00	0
DEC, NEG, SUB or SBC	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   dst: R, IR, X

Operation: dst <= dst - 1

Flags: **S:** Set if result is negative; cleared otherwise.
               Z: Set if result is zero; cleared otherwise.
               H: Set if arithmetic borrow out of bit 3; cleared otherwise.
               P/V: Set if arithmetic overflow; cleared otherwise.
               N: Set.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	DEC r	00rrr101	2
IR:	DEC (HL)	00110101	8 (6)
X:	DEC (IX+d) or DEC (IY+d)	11y11101 00110101 ----d---	12 (10)

Notes:

1. The **rrr** field uses the standard register select encoding

2. **y** = 0 selects IX and **y** = 1 selects IY

DEC

Decrement (Word)

DEC dst   dst: RR, IX, IY

Operation: dst <= dst - 1

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Unaffected.
               P/V: Unaffected.
               N: Unaffected.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
RR:	DEC dd	00dd1011	2
IX, IY	DEC IX or DEC IY	11y11101 00101011	4

Notes:

1.   The **dd** field uses the standard word register encoding.

Operation:	IFF1 <= 0
	IFF2 <= 0

Flags:	**S:** Unaffected.
	Z: Unaffected.
	H: Unaffected.
	P/V: Unaffected.
	N: Unaffected.
	C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	DI	11110011	2

Notes:

1. Interrupts are last sampled during the machine cycle that fetches this instruction.

DJNZ

Decrement, Jump if Non-zero

DJNZ e

Operation: B <= B - 1
                   if (B != 0) PC <= PC + e (where PC is the PC of this instruction)

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Unaffected.
               P/V: Unaffected.
               N: Unaffected.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	DJNZ e	00010000	6
		--(e-2)-	

Notes:

1. Relative to the address of this instruction, the jump range is -126 to +129. Relative to the address of the next instruction, the jump range is -128 to +127.

EI

Enable Interrupt

EI

| **Operation:** | IFF1 <= 1 |
| | IFF2 <= 1 |

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Unaffected.
               P/V: Unaffected.
               N: Unaffected.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	EI	11111011	2

Notes:

1. Interrupts are first sampled during the fetch of the next instruction. If an interrupt is pending this instruction fetch will be ignored and an interrupt acknowledge cycle started.

EX

Exchange with Top-of-Stack

EX (SP), src   src: HL, IX, IY

Operation: (SP) <=> L or IXL or IYL
                   (SP+1) <=> H or IXH or IYH

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Unaffected.
               P/V: Unaffected.
               N: Unaffected.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	EX (SP), HL	11100011	12 (10)
	EX (SP), IX or EX (SP), IY	11y11101 11100011	14 (12)

Notes:

1. **y** = 0 selects IX and **y** = 1 selects IY

EX AF, AF'

EX AF, AF'

Operation: AF <=> AF'

Flags:

S: Replaced by alternate flag.
Z: Replaced by alternate flag.
H: Replaced by alternate flag.
P/V: Replaced by alternate flag.
N: Replaced by alternate flag.
C: Replaced by alternate flag.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	EX AF, AF'	00001000	2

Notes:

1. No data is actually moved. Instead the registers are renamed.

EX

Exchange (Word)

EX DE, HL

Operation: DE <=> HL

Flags: **S:** Unaffected.
Z: Unaffected.
H: Unaffected.
P/V: Unaffected.
N: Unaffected.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	EX DE, HL	11101011	2

EXX

EXX

Operation: BC <=> BC'

DE <=> DE'

HL <=> HL'

Flags: **S:** Unaffected.

Z: Unaffected.

H: Unaffected.

P/V: Unaffected.

N: Unaffected.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	EXX	11011001	2

Notes:

1. No data is actually moved. Instead the registers are renamed.

HALT

Halt

HALT

Operation: activate Halt signal and wait for interrupt

Flags: **S:** Unaffected.
                Z: Unaffected.
                H: Unaffected.
                P/V: Unaffected.
                N: Unaffected.
                C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	HALT	01110110	4 + 2n

Notes:

1. The CPU halts with an idle bus until an interrupt is requested. The address pushed to the stack during the interrupt acknowledge is the address of the next instruction. During Halt the **mem_addr_out** and **io_addr_out** are driven with 0x0000, and the **mem_data_out** and **io_data_out** are driven with 0x00.

IM i

Operation:	Set Interrupt Mode i

Flags: **S:** Unaffected.
Z: Unaffected.
H: Unaffected.
P/V: Unaffected.
N: Unaffected.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	IM 0	11101101 01000110	4
	IM 1	11101101 01010110	4
	IM 2	11101101 01011110	4

Notes:

1. Interrupt Mode 0 expects an RST instruction on the **ivec_bus** during the interrupt acknowledge cycle. Only an RST instruction is allowed.

2. Interrupt Mode 1 always jumps to location 0x0038 in response to a maskable interrupt request.

3. Interrupt Mode 2 uses the interrupt vector returned on the **ivec_bus** during an interrupt acknowledge cycle, along with the contents of the I register, to access an interrupt vector table in memory. The address stored at the selected location in the interrupt vector table is the starting addess of the interrupt service routine. Note that the least-significant bit of the interrupt vector must be zero to account for the two-byte entries in the interrupt vector table.

IN

Input

IN A, src   src: DA

Operation: A <= I/O(A:n)

Flags: **S:** Unaffected.
                 Z: Unaffected.
                 H: Unaffected.
                 P/V: Unaffected.
                 N: Unaffected.
                 C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	IN A, (n)	11011011	8 (6)
		----n---	

IN

Input

IN r, (C)   dst: R

Operation: r <= I/O(BC)

Flags: **S:** Set if the input data is negative; cleared otherwise.
              Z: Set if the input data is zero; cleared otherwise.
              H: Cleared.
              P/V: Set if the parity of the input data is even; cleared otherwise.
              N: Cleared.
              C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	IN r, (C)	11101101 01rrr000	8 (6)

Notes:

1. The **rrr** field uses the standard register select encoding

IN0

Input (page 0)

IN0 r, (n)   dst: R

Operation: r <= I/O(0:n)

Flags: **S:** Set if input byte is negative; cleared otherwise.
Z: Set if input byte is zero; cleared otherwise.
H: Cleared.
P/V: Set if parity of input byte is even; cleared otherwise.
N: Cleared.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	IN0 r, (n)	11101101	10 (8)
		00rrr000	
		----n---	

Notes:

1. The **rrr** field uses the standard register select encoding

2. This instruction is not present in the original Z80, but is a feature of the Z180.

INC dst   dst: R, IR, X

Operation: dst <= dst + 1

Flags: **S:** Set if result is negative; cleared otherwise.
                     Z: Set if result is zero; cleared otherwise.
                     H: Set if arithmetic carry out of bit 3; cleared otherwise.
                     P/V: Set if arithmetic overflow; cleared otherwise.
                     N: Cleared.
                     C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	INC r	00rrr100	2
IR:	INC (HL)	00110100	8 (6)
X:	INC (IX+d) or INC (IY+d)	11y11101 00110100 ----d---	12 (10)

Notes:

1. The **rrr** field uses the standard register select encoding

2. **y** = 0 selects IX and **y** = 1 selects IY

INC

Increment (Word)

INC dst   dst: RR, IX, IY

Operation: dst <= dst + 1

Flags: **S:** Unaffected.
                 Z: Unaffected.
                 H: Unaffected.
                 P/V: Unaffected.
                 N: Unaffected.
                 C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
RR:	INC ss	00dd0011	2
IX, IY	INC IX or INC IY	11y11101 00100011	4

Notes:

1.   The **dd** field uses the standard word register encoding.

IND

IND

Operation:	(HL) <= I/O(BC)
	B <= B - 1
	HL <= HL -1

Flags: **S:** Unaffected.
Z: Set if result of decrementing B is zero; cleared otherwise.
H: Unaffected.
P/V: Unaffected.
N: Set.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	IND	11101101 10101010	10 (8)

Notes:

1. For the original Z80, the **S**, **H** and **P/V** flags are undefined.

INDR

Input, Decrement and Repeat

INDR

Operation:	(HL) <= I/O(BC)
	B <= B - 1
	HL <= HL -1
	repeat if B != 0

Flags: **S:** Unaffected.
Z: Set if result of decrementing B is zero; cleared otherwise.
H: Unaffected.
P/V: Unaffected.
N: Set.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	INDR	11101101 10111010	8 + 4i

Notes:

1. This instruction can be interrupted after each iteration. The address saved on the stack in this case is the address of this instruction, allowing completion of the instruction after the interrupt service routine

2. Interrupts are sampled during each I/O read operation.

3. For the original Z80, the **S**, **H** and **P/V** flags are undefined.

INI

INI

Operation:	(HL) <= I/O(BC)
	B <= B - 1
	HL <= HL + 1

Flags:
S: Unaffected.
Z: Set if result of decrementing B is zero; cleared otherwise.
H: Unaffected.
P/V: Unaffected.
N: Set.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	INI	11101101 10100010	10 (8)

Notes:

1. For the original Z80, the **S**, **H** and **P/V** flags are undefined.

INIR

Input, Increment and Repeat

INIR

Operation: (HL) <= I/O(BC)
B <= B - 1
HL <= HL + 1
repeat if B != 0

Flags: **S:** Unaffected.
Z: Set if result of decrementing B is zero; cleared otherwise.
H: Unaffected.
P/V: Unaffected.
N: Set.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	INIR	11101101 10110010	8 + 6i

Notes:

1. This instruction can be interrupted after each iteration. The address saved on the stack in this case is the address of this instruction, allowing completion of the instruction after the interrupt service routine

2. Interrupts are sampled during each I/O read operation.

3. For the original Z80, the **S**, **H** and **P/V** flags are undefined.

JP

Jump

JP dst   dst: IM, IR

Operation: PC <= dst

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Unaffected.
               P/V: Unaffected.
               N: Unaffected.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	JP (HL)	11101001	4
	JP (IX) or JP (IY)	11y11101 11101001	6
IM:	JP mn	11000011 ----n--- ----m---	8

Notes:

1. The indirect jumps use the contents of the register directly for the jump address.

JP

Conditional Jump

JP cc, mn

Operation: if (cc = true) PC <= mn

Flags: **S:** Unaffected.
Z: Unaffected.
H: Unaffected.
P/V: Unaffected.
N: Unaffected.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IM:	JP cc, mn	11fff010	8 (taken)
		----n---	6 (not taken)
		----m---	

Notes:

1.

Mnemonic	Encoding (fff)	Meaning	Flag case
NZ	000	Non-zero	Z = 0
Z	001	Zero	Z = 1
NC	010	Non-carry	C = 0
C	011	Carry	C = 1
PO	100	Parity Odd	P/V = 0
PE	101	Parity Even	P/V = 1
P	110	Plus	S = 0
M	111	Minus	S = 1

JR

Jump Relative

JR e

Operation:	PC <= PC + e (where PC is the PC of this instruction)

Flags: **S:** Unaffected.
          Z: Unaffected.
          H: Unaffected.
          P/V: Unaffected.
          N: Unaffected.
          C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	JR e	00011000 --(e-2)-	6

Notes:

1. Relative to the address of this instruction, the jump range is -126 to +129. Relative to the address of the next instruction, the jump range is -128 to +127.

JR

Conditional Jump Relative

JR cc, e

Operation: if (cc = true) PC <= PC + e (where PC is the PC of this instruction)

Flags: **S:** Unaffected.
Z: Unaffected.
H: Unaffected.
P/V: Unaffected.
N: Unaffected.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	JR cc, e	001cc000	6 (taken)
		--(e-2)-	4 (not taken)

Notes:

1. Relative to the address of this instruction, the jump range is -126 to +129. Relative to the address of the next instruction, the jump range is -128 to +127.

1.	Mnemonic	Encoding (cc)	Meaning	Flag case
	NZ	00	Non-zero	Z = 0
	Z	01	Zero	Z = 1
	NC	10	Non-carry	C = 0
	C	11	Carry	C = 1

LD

Load Accumulator from Memory

LD A, src   src: DA, IR

Operation: A <= src

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Unaffected.
               P/V: Unaffected.
               N: Unaffected.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
DA:	LD A, (mn)	00111010 ----n--- ----m---	10 (8)
IR:	LD A, (BC)	00001010	6 (4)
	LD A, (DE)	00011010	6 (4)

LD

Load Accumulator from Special Register

LD A, src   src: special register

Operation: A <= src

Flags: **S:** Set if the contents of the Special Register is negative; cleared otherwise.
Z: Set if the contents of the Special Register is zero; cleared otherwise.
H: Cleared.
P/V: Loaded with the contents if the IFF2 interrupt enable flag.
N: Cleared.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	LD A, I	11101101 01010111	4
	LD A, R	11101101 01011111	4

LD

Load Memory from Accumulator

LD dst, A   dst: DA, IR

Operation: dst <= A

Flags: **S:** Unaffected.
              Z: Unaffected.
              H: Unaffected.
              P/V: Unaffected.
              N: Unaffected.
              C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
DA:	LD (mn), A	00110010 ----n--- ----m---	10 (8)
IR:	LD (BC), A	00000010	6 (4)
	LD (DE), A	00010010	6 (4)

LD

Load Memory with Immediate

LD dst, n   dst: IR, X

Operation: dst <= n

Flags: **S:** Unaffected.
                  Z: Unaffected.
                  H: Unaffected.
                  P/V: Unaffected.
                  N: Unaffected.
                  C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	LD (HL), n	00110110 ----n---	6
X:	LD (IX+d), n or LD (IY+d), n	11y11101 00110110 ----d--- ----n---	10

Notes:

1. **y** = 0 selects IX and **y** = 1 selects IY

LD

Load Memory from Register

LD dst, r   dst: IR, X

Operation: dst <= r

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Unaffected.
               P/V: Unaffected.
               N: Unaffected.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IR:	LD (HL), r	01110rrr	6 (4)
X:	LD (IX+d), r or LD (IY+d), r	11y11101 01110rrr	10 (8)

Notes:

1. The **rrr** field uses the standard register select encoding

2. **y** = 0 selects IX and **y** = 1 selects IY

LD

Load Memory from Register (Word)

LD (mn), src   src: HL, RR, IX, IY

Operation: (mn) <= src

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Unaffected.
               P/V: Unaffected.
               N: Unaffected.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
HL:	LD (mn), HL	00100010 ----n--- ----m---	12 (10)
RR:	LD (mn), ss	11101101 01ss0011 ----n--- ----m---	14 (12)
IX, IY:	LD (mn), IX or LD (mn), IY	11y11101 00100010 ----n--- ----m---	14 (12)

Notes:

1.   The **ss** field uses the standard word register encoding.

LD

Load Register

LD r, src   dst: R, IM, IR, X

Operation: r <= src

Flags: **S:** Unaffected.
              Z: Unaffected.
              H: Unaffected.
              P/V: Unaffected.
              N: Unaffected.
              C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	LD rd, rs	01rdrrsr	2
IM	LD r, n	00rrr110 ----n---	4
IR:	LD r, (HL)	01rrr110	6 (4)
X:	LD r, (IX+d) or LD r, (IY+d)	11y11101 01rrr110 ----d---	10 (8)

Notes:

1. The **rdr, rsr** and **rrr** fields use the standard register select encoding

2. **y** = 0 selects IX and **y** = 1 selects IY

LD

Load Register Immediate (Word)

LD dst, mn   dst: RR, IX, IY

Operation: dst <= mn

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Unaffected.
               P/V: Unaffected.
               N: Unaffected.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
IM:	LD dd, mn	00dd0001 ----n--- ----m---	6
	LD IX, mn or LD IY, mn	11y11101 00100001 ----n--- ----m---	8

Notes:

1. The **dd** field uses the standard word register encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY

LD

Load Register (Word)

LD dst, (mn)   dst: RR, IX, IY

Operation: dst <= (mn)

Flags: **S:** Unaffected.
             Z: Unaffected.
             H: Unaffected.
             P/V: Unaffected.
             N: Unaffected.
             C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
DA:	LD HL, (mn)	00101010 ----n--- ----m---	12 (10)
	LD dd, (mn)	11101101 01dd1011 ----n--- ----m---	14 (12)
	LD IX, (mn) or LD IY, (mn)	11y11101 00101010 ----n--- ----m---	14 (12)

Notes:

1. The **dd** field uses the standard word register encoding.

LD

Load Special Register from Accumulator

LD dst, A   dst: special register

Operation: dst <= A

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Unaffected.
               P/V: Unaffected.
               N: Unaffected.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	LD I, A	11101101 01000111	4
	LD R, A	11101101 01001111	4

LD

Load Stack pointer

LD SP, src   src: HL, IX, IY

Operation: SP <=src

Flags: **S:** Unaffected.
                Z: Unaffected.
                H: Unaffected.
                P/V: Unaffected.
                N: Unaffected.
                C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	LD SP, HL	11111001	2
	LD SP, IX or LD SP, IY	11y11101 11111001	4

Notes:

2. **y** = 0 selects IX and **y** = 1 selects IY

LDD

Load and Decrement

LDD

Operation: (DE) <= (HL)
               BC <= BC - 1
               DE <= DE - 1
               HL <= HL -1

Flags: **S:** Unaffected.
           Z: Unaffected.
           H: Cleared.
           P/V: Set if result of decrementing BC is non-zero; cleared otherwise.
           N: Cleared.
           C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	LDD	11101101 10101000	10 (8)

LDDR

Load, Decrement and Repeat

LDDR

Operation: (DE) <= (HL)

BC <= BC - 1

DE <= DE - 1

HL <= HL -1

repeat if BC != 0

Flags: **S:** Unaffected.

Z: Unaffected

H: Cleared.

P/V: Set if result of decrementing BC is non-zero; cleared otherwise.

N: Cleared.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	LDDR	11101101 10111010	8 + 4i

Notes:

1. This instruction can be interrupted after each iteration. The address saved on the stack in this case is the address of this instruction, allowing completion of the instruction after the interrupt service routine

2. Interrupts are sampled during each memory read operation.

LDI

Load and Increment

INI

Operation: (DE) <= (HL)

                  BC <= BC - 1

                  DE <= DE + 1

                  HL <= HL + 1

Flags: **S:** Unaffected.

              Z: Unaffected.

              H: Cleared.

              P/V: Set if result of decrementing BC is non-zero; cleared otherwise.

              N: Cleared.

              C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	LDI	11101101 10100000	10 (8)

LDIR

Input, Increment and Repeat

LDIR

Operation: (DE) <= (HL)

BC <= BC - 1

DE <= DE + 1

HL <= HL + 1

repeat if BC != 0

Flags: **S:** Unaffected.

Z: Unaffected.

H: Cleared.

P/V: Set if result of decrementing BC is non-zero; cleared otherwise.

N: Cleared.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	LDIR	11101101 10110000	8 + 4i

Notes:

1. This instruction can be interrupted after each iteration. The address saved on the stack in this case is the address of this instruction, allowing completion of the instruction after the interrupt service routine

2. Interrupts are sampled during each memory read operation.

MLT

Multiply

MLT src   src: R

Operation: src <= srch * srcl

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Unaffected.
               P/V: Unaffected.
               N: Unaffected.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	MLT ss	11101101 01ss1100	14 (note 3)

Notes:

1. The **ss** field uses the standard word register encoding.

2. This is an unsigned multiply.

3. A compile-time option exists to change the execution time to 4 clock cycles. This option should only be selected if the technology supports fast carry chains, as it uses a parallel 8x8 multiplier.

4. This instruction is not present in the original Z80, but is a feature of the Z180.

NEG

Negate

Operation:	A <= 0 - A

Flags:

S: Set if result is negative; cleared otherwise.

Z: Set if result is zero; cleared otherwise.

H: Set if arithmetic borrow out of bit 3; cleared otherwise.

P/V: Set if arithmetic overflow (A was 0x80 before inst); cleared otherwise.

N: Cleared.

C: Set if arithmetic borrow out of bit 7 (A was not 0x00 before inst); cleared otherwise.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	NEG	11101101 00100100	4

NOP

No Operation

NOP

Operation:	none	

Flags:	**S:** Unaffected.	
	Z: Unaffected.	
	H: Unaffected.	
	P/V: Unaffected.	
	N: Unaffected.	
	C: Unaffected.	

Addressing Modes	Assembly Syntax	Encoding	Clocks
	NOP	00000000	2

OR

Logical OR

OR A, src   src: R, IM, IR, X

Operation: A <= A | src

Flags: **S:** Set if result is negative; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
H: Cleared.
P/V: Set if parity of result is even; cleared otherwise.
N: Cleared.
C: Cleared.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	OR A, r	10110rrr	2
IM:	OR A, n	11110110 ----n---	4
IR:	OR A, (HL)	10110110	6 (4)
X:	OR A, (IX+d) or OR A, (IY+d)	11y11101 10110110 ----d---	10 (8)

Notes:

1. The **rrr** field uses the standard register select encoding

2. **y** = 0 selects IX and **y** = 1 selects IY

OTDM

Output and Decrement Multiple

OTDM

Operation: I/O(0,C) <= (HL)
B <= B - 1
C <= C - 1
HL <= HL -1

Flags: **S:** Set if the result of decrementing B is negative; cleared otherwise.
Z: Set if result of decrementing B is zero; cleared otherwise.
H: Set if there is a borrow out of bit 3 while decrementing B; cleared otherwise.
P/V: Set if the parity of the result of decrementing B is even; cleared otherwise.
N: Set if the byte transferred is negative; cleared otherwise.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	OTDM	11101101 10001011	10 (8)

Notes:

1. This instruction is not present in the original Z80, but is a feature of the Z180.

OTDMR

Output, Decrement Multiple and Repeat

OTDMR

Operation:	I/O(BC) <= (HL)
	B <= B - 1
	C <= C - 1
	HL <= HL -1
	repeat if B != 0

Flags: **S:** Set if the result of decrementing B is negative; cleared otherwise.
Z: Set if result of decrementing B is zero; cleared otherwise.
H: Set if there is a borrow out of bit 3 while decrementing B; cleared otherwise.
P/V: Set if the parity of the result of decrementing B is even; cleared otherwise.
N: Set if the byte transferred is negative; cleared otherwise.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	OTDMR	11101101 10011011	8 + 4i

Notes:

1. This instruction can be interrupted after each iteration. The address saved on the stack in this case is the address of this instruction, allowing completion of the instruction after the interrupt service routine

2. Interrupts are sampled during each memory read operation.

OTDR

Output, Decrement and Repeat

OTDR

Operation: I/O(BC) <= (HL)

B <= B - 1

HL <= HL -1

repeat if B != 0

Flags: **S:** Unaffected.

Z: Set if result of decrementing B is zero; cleared otherwise.

H: Unaffected.

P/V: Unaffected.

N: Set.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	OTDR	11101101 10111011	8 + 4i

Notes:

1. This instruction can be interrupted after each iteration. The address saved on the stack in this case is the address of this instruction, allowing completion of the instruction after the interrupt service routine

2. Interrupts are sampled during each memory read operation.

3. For the original Z80, the **S**, **H** and **P/V** flags are undefined.

OTIM

OTIM

Operation:	I/O(BC) <= (HL)
	B <= B - 1
	C <= C + 1
	HL <= HL + 1

Flags: **S:** Set if the result of decrementing B is negative; cleared otherwise.

Z: Set if result of decrementing B is zero; cleared otherwise.

H: Set if there is a borrow out of bit 3 while decrementing B; cleared otherwise.

P/V: Set if the parity of the result of decrementing B is even; cleared otherwise.

N: Set if the byte transferred is negative; cleared otherwise.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	OTIM	11101101	10 (8)
		10000011	

Notes:

1. This instruction is not present in the original Z80, but is a feature of the Z180.

OTIMR

Output, Increment Multiple and Repeat

OTIMR

Operation: I/O(BC) <= (HL)

B <= B - 1

C <= C + 1

HL <= HL + 1

repeat if B != 0

Flags: **S:** Set if the result of decrementing B is negative; cleared otherwise.

Z: Set if result of decrementing B is zero; cleared otherwise.

H: Set if there is a borrow out of bit 3 while decrementing B; cleared otherwise.

P/V: Set if the parity of the result of decrementing B is even; cleared otherwise.

N: Set if the byte transferred is negative; cleared otherwise.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	OTIMR	11101101 10010011	8 + 4i

Notes:

1. This instruction can be interrupted after each iteration. The address saved on the stack in this case is the address of this instruction, allowing completion of the instruction after the interrupt service routine

2. Interrupts are sampled during each memory read operation.

3. This instruction is not present in the original Z80, but is a feature of the Z180.

OTIR

OTIR

Operation:	I/O(BC) <= (HL)
	B <= B - 1
	HL <= HL + 1
	repeat if B != 0

Flags:
S: Unaffected.
Z: Unaffected.
H: Unaffected.
P/V: Set if result of decrementing B is zero; cleared otherwise.
N: Cleared.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	OTIR	11101101 10110011	8 + 4i

Notes:

1. This instruction can be interrupted after each iteration. The address saved on the stack in this case is the address of this instruction, allowing completion of the instruction after the interrupt service routine

2. Interrupts are sampled during each memory read operation.

3. For the original Z80, the **S**, **H** and **P/V** flags are undefined.

OUT

Output

OUT dst, A   dst: DA

Operation: I/O(A:n) <= A

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Unaffected.
               P/V: Unaffected.
               N: Unaffected.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	OUT (n), A	11010011 ----n---	8 (6)

OUT

Output

OUT (C), r   src: R

Operation: I/O(BC) <= r

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Unaffected.
               P/V: Unaffected.
               N: Unaffected.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	OUT (C), r	11101101 01rrr001	8 (6)

Notes:

1. The **rrr** field uses the standard register select encoding

OUT0

Output (page 0)

OUT0 (n), r   src: R

Operation: I/O(0,n) <= r

Flags: **S:** Unaffected.
                  Z: Unaffected.
                  H: Unaffected.
                  P/V: Unaffected.
                  N: Unaffected.
                  C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	OUT0 (n), r	11101101 00rrr001 ----n---	10 (8)

Notes:

1. The **rrr** field uses the standard register select encoding

2. This instruction is not present in the original Z80, but is a feature of the Z180.

OUTD

Output and Decrement

OUTD

Operation:	I/O(BC) <= (HL)
	B <= B - 1
	HL <= HL -1

Flags:

S: Unaffected.
Z: Set if result of decrementing B is zero; cleared otherwise.
H: Unaffected.
P/V: Unaffected.
N: Set.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	OUTD	11101101 10101011	10 (8)

Notes:

1. For the original Z80, the **S**, **H** and **P/V** flags are undefined.

OUTI

Output and Increment

OUTI

Operation: I/O(BC) <= (HL)

B <= B - 1

HL <= HL + 1

Flags: **S:** Unaffected.

Z: Unaffected.

H: Unaffected.

P/V: Set if result of decrementing B is zero; cleared otherwise.

N: Cleared.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	OUTI	11101101 10100011	10 (8)

Notes:

1. For the original Z80, the **S**, **H** and **P/V** flags are undefined.

POP

Pop from Stack

POP dst   dst: RR, IX, IY

Operation:	dst[lsb] <= (SP)
	dst[msb] <= (SP+1)
	SP <= SP + 2

Flags: **S:** Unaffected.
              Z: Unaffected.
              H: Unaffected.
              P/V: Unaffected.
              N: Unaffected.
              C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
RR:	POP tt	11tt0001	8 (6)
IX, IY	POP IX or POP IY	11y11101 11100001	10 (8)

Notes:

1. The **tt** field uses the standard word register encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY

PUSH

Push to Stack

PUSH src   src: RR, IX, IY

Operation: (SP-1) <= src[msb]
                 (SP-2) <= src[lsb]
                 SP <= SP - 2

Flags: **S:** Unaffected.
             Z: Unaffected.
             H: Unaffected.
             P/V: Unaffected.
             N: Unaffected.
             C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
RR:	PUSH tt	11tt0101	8 (6)
IX, IY	PUSH IX or PUSH IY	11y11101 11100101	10 (8)

Notes:

1. The **tt** field uses the standard word register encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY

RES b, dst   src: R, IR, X

Operation: dst[b] <= 0

Flags: **S:** Unaffected.
              Z: Unaffected.
              H: Unaffected.
              P/V: Unaffected.
              N: Unaffected.
              C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	RES b, r	11001011 10bbbrrr	4
IR:	RES b, (HL)	10100110 10bbb1110	10 (8)
X:	RES b, (IX+d) or RES (IY+d)	11y11101 11001011 ----d--- 10bbb110	12 (10)

Notes:

1. The **rrr** field uses the standard register select encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY.

3. The **bbb** field uses normal binary encoding.

RET

Return from Subroutine

Operation: PC[lsb] <= (SP)

PC[msb] <= (SP+1)

SP <= SP + 2

Flags: **S:** Unaffected.

Z: Unaffected.

H: Unaffected.

P/V: Unaffected.

N: Unaffected.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	RET	11001001	10

RET

Conditional Return from Subroutine

RET cc

Operation: if (cc = true) begin
      PC[lsb] <= (SP)
      PC[msb] <= (SP+1)
      SP <= SP + 2
      end

Flags: **S:** Unaffected.
Z: Unaffected.
H: Unaffected.
P/V: Unaffected.
N: Unaffected.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	RET cc	11fff000	10 (taken) 2 (not taken)

Notes:

1.

Mnemonic	Encoding (fff)	Meaning	Flag case
NZ	000	Non-zero	Z = 0
Z	001	Zero	Z = 1
NC	010	Non-carry	C = 0
C	011	Carry	C = 1
PO	100	Parity Odd	P/V = 0
PE	101	Parity Even	P/V = 1
P	110	Plus	S = 0
M	111	Minus	S = 1

RETI

Return from Interrupt

RETI

Operation: PC[lsb] <= (SP)

PC[msb] <= (SP+1)

SP <= SP + 2

Flags: **S:** Unaffected.

Z: Unaffected.

H: Unaffected.

P/V: Unaffected.

N: Unaffected.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	RETI	11101101 01001101	12

Notes:

1. This instruction activates the dedicated RETI signal out of the core.

RETN

Return from Non-Maskable Interrupt

RETN

Operation:	PC[lsb] <= (SP)
	PC[msb] <= (SP+1)
	SP <= SP + 2
	IFF2 <= IFF1

Flags: **S:** Unaffected.
Z: Unaffected.
H: Unaffected.
P/V: Unaffected.
N: Unaffected.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	RETN	11001001	12
		01000101	

RL

Rotate Left

RL src   src: R, IR, X

Operation: {CF, src} <= {src, CF}

Flags: **S:** Set if result is negative; cleared otherwise.
               Z: Set if result is zero; cleared otherwise.
               H: Cleared.
               P/V: Set if parity of result is even; cleared otherwise.
               N: Cleared.
               C: Data from bit 7.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	RL r	11001011 00010rrr	4
IR:	RL (HL)	10100110 00010110	10 (8)
X:	RL (IX+d) or RL (IY+d)	11y11101 11001011 ----d--- 00010110	12 (10)

Notes:

1. The **rrr** field uses the standard register select encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY.

RLA

Rotate Left Accumulator

RLA

Operation:	{CF, A} <= {A, CF}

Flags: **S:** Unaffected
               Z: Unaffected.
               H: Cleared.
               P/V: Unaffected.
               N: Cleared.
               C: Data from bit 7.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	RLA	00010111	2

RLC

Rotate Left Circular

RLC src   src: R, IR, X

Operation: {CF, src} <= {src, src[7]}

Flags: **S:** Set if result is negative; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
H: Cleared.
P/V: Set if parity of result is even; cleared otherwise.
N: Cleared.
C: Data from bit 7.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	RLC r	11001011 00000rrr	4
IR:	RLC (HL)	10100110 00000110	10 (8)
X:	RLC (IX+d) or RLC (IY+d)	11y11101 11001011 ----d--- 00000110	12 (10)

Notes:

1. The **rrr** field uses the standard register select encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY.

RLCA

Rotate Left Circular Accumulator

Operation: {CF, A} <= {A, A[7]}

Flags: **S:** Unaffected
Z: Unaffected.
H: Cleared.
P/V: Unaffected.
N: Cleared.
C: Data from bit 7.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	RLCA	00000111	2

RLD

Rotate Left Digit

RLD

Operation: {A, (HL)} <= {A[7:4], (HL), A[3:0]}

Flags: **S:** Set if A is negative after the operation; cleared otherwise.
Z: Set if A is zero after the operation; cleared otherwise.
H: Cleared.
P/V: Set if parity of A is even after the operation; cleared otherwise.
N: Cleared.
C: Unaffected

Addressing Modes	Assembly Syntax	Encoding	Clocks
	RLD	11101101 01101111	10 (8)

130

RR

Rotate Right

RR src   src: R, IR, X

Operation: {src, CF} <= {CF, src}

Flags: **S:** Set if result is negative; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
H: Cleared.
P/V: Set if parity of result is even; cleared otherwise.
N: Cleared.
C: Data from bit 0.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	RR r	11001011 00011rrr	4
IR:	RR (HL)	10100110 00011110	10 (8)
X:	RR (IX+d) or RR (IY+d)	11y11101 11001011 ----d--- 00011110	12 (10)

Notes:

1. The **rrr** field uses the standard register select encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY.

RRA

Rotate Right Accumulator

RRA

Operation: {A, CF} <= {CF, A}

Flags:

S: Unaffected
Z: Unaffected.
H: Cleared.
P/V: Unaffected.
N: Cleared.
C: Data from bit 0.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	RRA	00011111	2

RRC

Rotate Right Circular

RRC src   src: R, IR, X

Operation: {src, CF} <= {src[0], src}

Flags: **S:** Set if result is negative; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
H: Cleared.
P/V: Set if parity of result is even; cleared otherwise.
N: Cleared.
C: Data from bit 0.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	RRC r	11001011 00001rrr	4
IR:	RRC (HL)	10100110 00001110	10 (8)
X:	RRC (IX+d) or RRC (IY+d)	11y11101 11001011 ----d--- 00001110	12 (10)

Notes:

1. The **rrr** field uses the standard register select encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY.

RRCA

Rotate Right Circular Accumulator

Operation: {A, CF} <= {A[0], A}

Flags: **S:** Unaffected

Z: Unaffected.

H: Cleared.

P/V: Unaffected.

N: Cleared.

C: Data from bit 0.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	RRCA	00001111	2

RRD

RRD

Operation: {A, (HL)} <= {A[7:4], (HL)[3:0], A[3:0], (HL)[7:4]}

Flags: **S:** Set if A is negative after the operation; cleared otherwise.
Z: Set if A is zero after the operation; cleared otherwise.
H: Cleared.
P/V: Set if parity of A is even after the operation; cleared otherwise.
N: Cleared.
C: Unaffected

Addressing Modes	Assembly Syntax	Encoding	Clocks
	RRD	11101101 01100111	10 (8)

RST

Restart

RST v

Operation: SP <= SP - 2
                   (SP) <= PC
                   PC <= v

Flags: **S:** Unaffected.
               Z: Unaffected.
               H: Unaffected.
               P/V: Unaffected.
               N: Unaffected.
               C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	RST v	11vvv111	8

Notes:

1.

Mnemonic	Encoding (vvv)	Restart Address
0	000	0x0000
0x8	001	0x0008
0x10	010	0x0010
0x18	011	0x0018
0x20	100	0x0020
0x28	101	0x0028
0x30	110	0x0030
0x38	111	0x0038

SBC

Subtract With Carry

SBC A, src   src: R, IM, IR, X

Operation: A <= A - src - CF

Flags: **S:** Set if result is negative; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
H: Set if arithmetic borrow out of bit 3; cleared otherwise.
P/V: Set if arithmetic overflow; cleared otherwise.
N: Cleared.
C: Set if arithmetic borrow out of bit 7; cleared otherwise.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SBC A, r	10011rrr	2
IM:	SBC A, n	11011110 ----n---	4
IR:	SBC A, (HL)	10011110	6 (4)
X:	SBC A, (IX+d) or SBC A, (IY+d)	11y11101 10011110 ----d---	10 (8)

Notes:

1. The **rrr** field uses the standard register select encoding

2. **y** = 0 selects IX and **y** = 1 selects IY

SBC

Subtract With Carry (Word)

SBC HL, src   src: RR

Operation: HL <= HL - src - CF

Flags: **S:** Set if result is negative; cleared otherwise.
                  Z: Set if result is zero; cleared otherwise.
                  H: Set if arithmetic borrow out of bit 11; cleared otherwise.
                  P/V: Set if arithmetic overflow; cleared otherwise.
                  N: Cleared.
                  C: Set if arithmetic carry out of bit 15; cleared otherwise.

Addressing Modes	Assembly Syntax	Encoding	Clocks
RR:	SBC HL, ss	11101101 01ss0010	4

Notes:

1. The **ss** field uses the standard word register encoding.

138

CCF

Operation:	CF <= 1

Flags: **S:** Unaffected.
Z: Unaffected.
H: Cleared.
P/V: Unaffected.
N: Cleared.
C: Set.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	SCF	00110111	2

SET

Bit Set

SET b, dst   src: R, IR, X

Operation: dst[b] <= 1

Flags: **S:** Unaffected.
                 Z: Unaffected.
                 H: Unaffected.
                 P/V: Unaffected.
                 N: Unaffected.
                 C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SET b, r	11001011 11bbbrrr	4
IR:	SET b, (HL)	10100110 11bbb1110	10 (8)
X:	SET b, (IX+d) or SET b, (IY+d)	11y11101 11001011 ----d--- 11bbb110	12 (10)

Notes:

1. The **rrr** field uses the standard register select encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY.

3. The **bbb** field uses normal binary encoding.

SLA

Shift Left Arithmetic

SLA src   src: R, IR, X

Operation: {CF, src} <= {src, 0}

Flags: **S:** Set if result is negative; cleared otherwise.
               Z: Set if result is zero; cleared otherwise.
               H: Cleared.
               P/V: Set if parity of result is even; cleared otherwise.
               N: Cleared.
               C: Data from bit 7.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SLA r	11001011 00100rrr	4
IR:	SLA (HL)	10100110 00100110	10 (8)
X:	SLA (IX+d) or SLA (IY+d)	11y11101 11001011 ----d--- 00100110	12 (10)

Notes:

1. The **rrr** field uses the standard register select encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY.

SLP

Sleep

SLP

Operation: activate SLEEP signal and wait for interrupt

Flags: **S:** Unaffected.
Z: Unaffected.
H: Unaffected.
P/V: Unaffected.
N: Unaffected.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	SLP	11101101 01110110	6 + 2n

Notes:

1. The CPU halts, with an idle bus, until an interrupt is requested. During Sleep the **mem_addr_out** and **io_addr_out** are driven with 0x0000, and the **mem_data_out** and **io_data_out** are driven with 0x00.

2. In the case of an NMI or enabled maskable interrupt the address pushed to the stack during the interrupt acknowledge is the address of the next instruction.

3. If interrupts are disabled a maskable interrupt request during Sleep causes the CPU to resume execution with the next instruction. This saves time when restarting from Sleep.

4. This instruction is not present in the original Z80, but is a feature of the Z180.

SRA

Shift Right Arithmetic

SRA src   src: R, IR, X

Operation: {src, CF} <= {src[7], src}

Flags: **S:** Set if result is negative; cleared otherwise.
            Z: Set if result is zero; cleared otherwise.
            H: Cleared.
            P/V: Set if parity of result is even; cleared otherwise.
            N: Cleared.
            C: Data from bit 0.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SRA r	11001011 00101rrr	4
IR:	SRA (HL)	10100110 00101110	10 (8)
X:	SRA (IX+d) or SRA (IY+d)	11y11101 11001011 ----d--- 00101110	12 (10)

Notes:

1. The **rrr** field uses the standard register select encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY.

SRL

Shift Right Logical

SRL src   src: R, IR, X

Operation: {src, CF} <= {0, src}

Flags: **S:** Set if result is negative; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
H: Cleared.
P/V: Set if parity of result is even; cleared otherwise.
N: Cleared.
C: Data from bit 0.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SRL r	11001011 00111rrr	4
IR:	SRL (HL)	10100110 00111110	10 (8)
X:	SRL (IX+d) or SRL (IY+d)	11y11101 11001011 ----d--- 00111110	12 (10)

Notes:

1. The **rrr** field uses the standard register select encoding.

2. **y** = 0 selects IX and **y** = 1 selects IY.

144

SUB

Subtract

SUB A, src   src: R, IM, IR, X

Operation: A <= A - src

Flags: **S:** Set if result is negative; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
H: Set if arithmetic borrow out of bit 3; cleared otherwise.
P/V: Set if arithmetic overflow; cleared otherwise.
N: Cleared.
C: Set if arithmetic borrow out of bit 7; cleared otherwise.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	SUB A, r	10010rrr	2
IM:	SUB A, n	11010110 ----n---	4
IR:	SUB A, (HL)	10010110	6 (4)
X:	SUB A, (IX+d) or SUB A, (IY+d)	11y11101 10010110 ----d---	10 (8)

Notes:

1. The **rrr** field uses the standard register select encoding

2. **y** = 0 selects IX and **y** = 1 selects IY

TST

Test

TST src   src: R, IM, IR

Operation: A & src

Flags: **S:** Set if result is negative; cleared otherwise.
Z: Set if result is zero; cleared otherwise.
H: Set.
P/V: Set if parity of result is even; cleared otherwise.
N: Cleared.
C: Cleared.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	TST r	11101101 00rrr100	4
IM:	TST n	11101101 01100100 ----n---	6
IR:	TST (HL)	11101101 00110100	8 (6)

Notes:

1. The **rrr** field uses the standard register select encoding

2. This instruction is not present in the original Z80, but is a feature of the Z180.

TSTIO

TSTIO n

Operation: I/O(0,C) & n

Flags: **S:** Set if result is negative; cleared otherwise.

                 Z: Set if result is zero; cleared otherwise.

                 H: Set.

                 P/V: Set if parity of result is even; cleared otherwise.

                 N: Cleared.

                 C: Cleared.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	TSTIO n	11101101 01110100 ----n---	10

Notes:

1. This instruction is not present in the original Z80, but is a feature of the Z180.

XOR

Logical Exclusive-OR

XOR A, src   src: R, IM, IR, X

Operation: A <= A ^ src

Flags: **S:** Set if result is negative; cleared otherwise.
            Z: Set if result is zero; cleared otherwise.
            H: Cleared.
            P/V: Set if parity of result is even; cleared otherwise.
            N: Cleared.
            C: Cleared.

Addressing Modes	Assembly Syntax	Encoding	Clocks
R:	XOR A, r	10101rrr	2
IM:	XOR A, n	11101110 ----n---	4
IR:	XOR A, (HL)	10101110	6 (4)
X:	XOR A, (IX+d) or XOR A, (IY+d)	11y11101 10101110 ----d---	10 (8)

Notes:

1. The **rrr** field uses the standard register select encoding

2. **y** = 0 selects IX and **y** = 1 selects IY

Memory Management

The Y90 MPU includes a tightly integrated Memory Management Unit (MMU). This MMU expands the 64K byte logical address space of the processor to a physical address space of up to 256M bytes. The logical address space (the address visible to the programmer) is divided into sixteen pages of 4K bytes each, using the four most-significant bits of the logical address as the page identifier. The page identifier is used to access a sixteen-entry by sixteen-bit dedicated memory, with the sixteen data bits serving as the upper physical address bits. This translation, from logical address to physical address, occurs automatically for every memory access. I/O addresses are not translated by the MMU.

The MMU registers are initialized by reset to values that correspond to "no translation". That is, the MMU register used for logical address page 0xN is intiialized to 0x000N. Thus, if the MMU is never loaded with different values the logical address and physical address are effectively identical.

Although the MMU handles 28-bit physical addresses, it is anticipated that in most cases a 20-bit pysical address will be sufficient. To account for this fact the MMU registers are divided, with the upper eight bits loaded separately from the lower eight bits. This cuts the time necessary to reprogram the MMU in half when a 1M byte physical address space is sufficient. Of course the full MMU register width is still accessable; it just takes longer to swap the MMU context in this case.

The entire set of MMU registers is buffered. This means that the MMU continues to operate, with the same context, while the next MMU context is being loaded. It also means that if the next MMU context is preloaded the actual context swap can be very fast.

Rather than using I/O locations to control the MMU registers, the MMU is accessed using dedicated instructions. This removes the possibility of DMA inadvertently reprogramming the MMU. It also makes reprogramming the MMU significantly faster, cutting in half the number of required bus transactions. Dedicated instructions also require less setup and CPU register resources than I/O instructions.

A total of six instructions are used by the MMU. One instruction swaps the MMU registers with the buffer registers. One instruction does a block load of the upper eight bits of the buffer registers, while another does a block load of the lower eight bits of the buffer registers. A separate instruction does a block load of the lower eight bits of the buffer reg-

isters followed by a context swap. There are also two instructions that read the values in the two sections of the buffer registers.

All of these block instructions are atomic, meaning that they cannot be interrupted by either an interrupt or a DMA transfer. In addition, these instructions do not sample interrupts upon completion, making it possible to load the entire MMU without interruption or load the MMU and Jump without worring about an intervening interrupt or DMA request.

The following pages detail the MMU instructions.

LDM

Load Multiple

LDM (HL),src   src: MMU, MMUH

Operation: count <= 0
                     loop: (HL) <= MMUbuffer[count]
                          HL <= HL + 1
                          count <= count + 1
                          repeat loop if count != 16

Flags: **S:** Unaffected.
                     Z: Unaffected.
                     H: Unaffected.
                     P/V: Unaffected.
                     N: Unaffected.
                     C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	LDM (HL),MMU	11101101 10000010	40
	LDM (HL),MMUH	11101101 10010010	40

Notes:

1. This instruction does not sample interrupts. The following instruction is guaranteed to be executed.

LDM

Load Multiple

LDM dst, (HL)   dst: MMU, MMUH

Operation: count <= 0

loop: MMUbuffer[count] <= (HL)

HL <= HL + 1

count <= count + 1

repeat loop if count != 16

Flags: **S:** Unaffected.

Z: Unaffected.

H: Unaffected.

P/V: Unaffected.

N: Unaffected.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	LDM MMU,(HL)	11101101 10000000	42
	LDM MMUH,(HL)	11101101 10010000	42

Notes:

1. This instruction does not sample interrupts. The following instruction is guaranteed to be executed.

LDMS

Load Multiple and Swap

LDMS dst, (HL)   dst: MMU

Operation: count <= 0
                  loop: MMUbuffer[count] <= (HL)
                        HL <= HL + 1
                        count <= count + 1
                        repeat loop if count != 16
                  MMU <=> MMUbuffer

Flags: **S:** Unaffected.
              Z: Unaffected.
              H: Unaffected.
              P/V: Unaffected.
              N: Unaffected.
              C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	LDMS MMU,(HL)	11101101 10001000	42

Notes:

1. This instruction does not sample interrupts. The following instruction is guaranteed to be executed.

SWAP

Swap MMU context

Operation: MMU <=> MMUbuffer

Flags: **S:** Unaffected.
    Z: Unaffected.
    H: Unaffected.
    P/V: Unaffected.
    N: Unaffected.
    C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	SWAP	11101101 01111111	8

Notes:

1. This instruction does not sample interrupts. The following instruction is guaranteed to be executed.

As mentioned previously, not all of the MMU register data has to be used as memory addresses, or at all. For example, one bit of the upper byte of MMU data can be used as a Write-Protect bit by connecting the corresponding **mem_addr_out** bit directly to the **wr_inh** input. This will inhibit the **mem_wr** signal for any write transaction in the corresponding memory page and generate a Trap.

Another bit of the upper byte of MMU data can be used as an Execute Protect bit by connecting the corresponding **mem_addr_out** bit directly to the **exec_inh** input. This will cause any Instruction Fetch transaction in the corresponding memory page to generate a Trap.

Systems that use a boot ROM overlayed with RAM can used two bits of the upper byte of MMU data to control the routing of the **mem_rd** and **mem_wr** signals to the two memories using external gating controlled by the corresponding **mem_addr_out** bits. Since the upper byte of MMU data is initialized to 0x00, a Low on one bit should enable the **mem_rd** to be routed to the boot ROM rather than the overlay RAM. The other bit does the same thing for the **mem_wr** signal. Then the overlay RAM can be enabled for writes and the boot ROM copied using a block move instruction. Once the overlay RAM is loaded the MMU data can be modified to enable the **mem_rd** signal to be routed to the overlay RAM, effectively disabling the boot ROM. A reset will automatically revert to the boot ROM, but it can also be restored via software by reporgramming the upper bytes of the MMU.

Both the MMU buffer and the active MMU registers are initialized for no translation. This allows upper bytes of the MMU data to be initialized to fixed values in cases where they are not used for addresses. Simply load the upper bytes of the buffer, swap the context, and the reload the upper bytes of the buffer with the same data. Then if only the lower bytes of the MMU registers are used no further action is ever necessary.

The table below shows a memory structure of MMU data when using only 20-bit memory addressing:

Address	Data
(HL+0xF)	page F mem_addr_out[19:12]
(HL+0xE)	page E mem_addr_out[19:12]
.	.
.	.
.	.
(HL+1)	page 1 mem_addr_out[19:12]
(HL)	page 0 mem_addr_out[19:12]

The table below shows a memory structure of MMU data when using full 28-bit memory addressing, assuming that the MMU is to be loaded by a LDM MMUH,(HL) followed by a LDM MMU,(HL):

Address	Data
(HL+0x1F)	page F mem_addr_out[19:12]
(HL+0x1E)	page E mem_addr_out[19:12]
.	.
.	.
.	.
(HL+0x11)	page 1 mem_addr_out[19:12]
(HL+0x10)	page 0 mem_addr_out[19:12]
(HL+0xF)	page F mem_addr_out[27:20]
(HL+0xE)	page E mem_addr_out[27:20]
.	.
.	.
.	.
(HL+0x1)	page 1 mem_addr_out[27:20]
(HL)	page 0 mem_addr_out[27:20]

Caution must be exercised if mapping Page 0. This is due to the fact that, although they both jump to logical address 0x0000, a reset and a trap affect the MMU contents differently. Reset always initializes the MMU registers for no translation, so Page 0 will be fetched from physical address 0x000000. But a trap does not affect the MMU contents, meaning that Page 0 (and all other pages) will still be mapped.

Interrupt Control

The Y90 MPU includes an interrupt controller that is optimized for use with Interrupt Mode 2. The interrupt controller supports sixteen interrupt inputs and can easily be cascaded to a total of up to 128 interrupt inputs. The controller automatically prioritizes the interrupt requests and returns the vector for the highest-priority pending interrupt during the interrupt acknowledge cycle.

The controller contains an Interrupt-Under-Service (IUS) bit for each interrupt. The appropriate IUS bit is set during the interrupt acknowledge cycle, and this IUS bit then inhibits all lower-priority interrupts from being requested. However, higher-priority interrupts can still be requested and serviced.

The interrupt service routine ends with an RETI instruction, and this instruction clears the highest-priority IUS bit that is set, automatically re-enabling lower-priority interrupts. As long as all interrupt service routines end with an RETI instruction the hardware in the interrupt controller(s) automatically take care of all interrupt prioritization.

The interrupt controller is designed to be easily cascaded. The example Verilog HDL code below shows the cascade of two interrupt controllers. The **int_prio_in** and **int_prio_out** buses (they're two bits wide) take care of the interrupt request and IUS prioritization. The **ivec_offset** must be different for each instance, because this value is the three most-significant bits of the interrupt vector for the instance.

The **int_req** signals and **ivec_data_out** buses require no multiplexing, but are merely OR'ed together and connected to the corresponding Y90 processor inputs.

```
int_ctl INT_1 ( .int_ack_bus(int_ack_bus[31:16]),
                .int_prio_out(int_prio_out_1), .int_req(int_req_1),
                .ivec_data_out(ivec_data_out_1),
                .clkc(clkc), .iack_tran(iack_tran),
                .int_prio_in(2'b11),
                .int_req_bus(int_req_bus[31:16]),
                .ivec_offset(3'b001), .ivec_rd(ivec_rd),
                .resetb(resetb), reti_tran(reti_tran) );

int_ctl INT_0 ( .int_ack_bus(int_ack_bus[15:0]), .int_prio_out(),
                .int_req(int_req_0), .ivec_data_out(ivec_data_out_0),
                .clkc(clkc), .iack_tran(iack_tran),
```

```
            .int_prio_in(int_prio_out_1),
            .int_req_bus(int_req_bus[15:0]),
            .ivec_offset(3'b000), .ivec_rd(ivec_rd),
            .resetb(resetb), .reti_tran(reti_tran) );

assign int_req      = int_req_1      | int_req_0;
assign ivec_data_out = ivec_data_out_1 | ivec_data_out_0;
```

The **ivec_offset** value may be fixed, as shown in the above example, or controlled via an external I/O register defined by the user.

Although Interrupt Mode 2 is the preferred mode for use with the interrupt controller, the design allows the use of any interrupt mode. In Interrupt Mode 0 or 1 it will be necessary to use the **int_ack_bus** signals to externally latch the information about which interrupt is being acknowledged. This is because Interrupt Modes 0 and 1 will branch to a common interrupt service routine, rather than the individual routines possible in Interrupt Mode 2.

The value returned during Interrupt Acknowledge transactions on the **ivec_data_out** bus is different depending on the interrupt mode, according to the table below:

bit	Interrupt Mode 2	Interrupt Mode 1	Interrupt Mode 0
7	ivec_offset[2]	1	1
6	ivec_offset[1]	1	1
5	ivec_offset[0]	ivec_offset[2]	ivec_offset[2]
4	4-bit binary code	ivec_offset[1]	ivec_offset[1]
3	for	ivec_offset[0]	ivec_offset[0]
2	highest priority	1	1
1	interrupt pending	1	1
0	0	1	1

In Interrupt Mode 1 the value on the **ivec_data_out** bus is ignored. In Interrupt Mode 0 the data returned is one of the eight RST instructions. It is not advisable to use the RST 0 instruction, as that will vector through location 0x0000, which is in common with the reset and trap cases.

DMA Request Control

The Y90 MPU includes a DMA request controller and DMA request limiter. The controller supports eight DMA request (**dma_req**) inputs and can easily be cascaded to any number of DMA request inputs. The controller automatically prioritizes the DMA requests and routes the DMA acknowledge (**dma_ack**) signal from the processor to the highest-priority requestor. The controller uses a fixed priority-resolution scheme, so the DMA requests must be connected appropriately.

The controller contains a DMA-Under-Service (DUS) bit for each request. The DUS bit is set by the **dma_ack** signal from the processor, and inhibits all lower-priority requests from being acknowledged. During the time that the **dma_ack** signal is active, the controller automatically switches between DMA requests as necessary according to the request priority. Only one DUS bit will be active at a time.

The DMA request limiter performs two functions. First, it limits the total time that the **dma_ack** signal can be active, preventing DMA requestors from hogging the bus. It does this by de-asserting the priority-chain input to the highest-priority DMA bus controller. The time limit for the **dma_ack** signals can be set, in multiples of 16 clock cycles, from 16 clock cycles to 512 clock cycles. This function can be disabled by setting this limit to zero.

The second function of the DMA request limiter sets the minimum inactive time for the **dma_ack** signal, guaranteeing a minimum number of clock cycles available to the processor between DMA requests. It uses also uses the priority-chain input to inhibit DMA requests. This time limit can be set from 16 to 512 clock cycles, in multiples of 16 clocks. This function is disabled by setting the limit to zero.

The DMA request controller allows immediate pre-emption by higher-priority DMA requests. If the DMA devices in the system cannot tolerate this type of operation all higher-priority requests should be disabled whenever a **dma_ack** signal is active.

The DMA request limit should always be set high enough that it will only be activated in the case of an error condition. This is because the limit function immediately de-asserts all **dma_ack** signals and restores bus control to the processor, independent of any DMA transfers in progress.

The DMA request controller is designed to be easily cascaded. The example Verilog HDL code below shows the cascade of two controllers. The **drq_prio_in** and **drq_prio_out**

signals take care of the DMA request prioritization. The **drq_prio_in** signal of the highest-priority controller is connected to the output of the DMA request limiter.

The **dma_req** signals require no multiplexing, but are merely OR'ed together and connected to the corresponding Y90 processor input.

```
drq_ctl DRQ_1 ( .dma_ack_bus(dma_ack_bus[15:8]), .dma_req(dma_req_1),
                .drq_prio_out(drq_prio_out_1), .clkc(clkc),
                .dma_ack(dma_ack), .dma_req_bus(dma_req_bus[15:8]),
                .drq_prio_in(drq_prio_out_lim), .resetb(resetb) );

drq_ctl DRQ_0 ( .dma_ack_bus(dma_ack_bus[7:0]), .dma_req(dma_req_0),
                .drq_prio_out(), .clkc(clkc),
                .dma_ack(dma_ack), .dma_req_bus(dma_req_bus[7:0]),
                .drq_prio_in(drq_prio_out_1), .resetb(resetb) );

assign dma_req  = dma_req_1 || dma_req_0;


drq_lim DRQ_LIM ( .drq_prio_out(drq_prio_out_lim),
                  .drq_timeout(drq_timeout), .clkc(clkc),
                  .dma_ack(dma_ack), .drq_act_lim(drq_act_lim),
                  .drq_idl_lim(drq_idl_lim), .resetb(resetb) );
```

The two time limits may be hard-coded or controlled by external I/O registers defined by the user.

There is a status output (**drq_timeout**) from the DMA request limiter that is activated for one clock cycle whenever the active time limit is reached and the bus is restored to the processor. This signal can be used as a system diagnostic, for example, by counting these occurances over time. The signal may also be used to indicate a fault in a DMA controller that is causing the DMA to attempt to tie up the bus.

Wait Request Control

The Y90 MPU includes a Wait Request controller to prevent bus hang-ups. This controller accepts and monitors three separate Wait Request inputs (**wait_iack**, **wait_io** and **wait_mem**) for the different transaction types. The appropriate input is selected to create the **wait_req** signal to the processor.

The controller limits the total time that the **wait_req** signal can be active to prevent a bus hang-up. It does this by automatically de-asserting the **wait_req** signal into the processor after a programmed time.

The time limit for the **wait_iack**, **wait_io** and **wait_mem** signals can be set from 1 to 31 clock cycles. The limit function can be disabled by setting the limit to zero. Each transaction type has an independent setting. The three time limits may be hard-coded or controlled by external I/O registers defined by the user.

There is a status output (**wait_timeout**) from the controller that is activated whenever the active time limit is reached and the bus is restored to the processor. This signal can be used as a system diagnostic, for example, by counting these occurrences over time. The signal may also be used to indicate a fault in the logic that is generating the Wait requests.

The figure below shows the timing for a memory read cycle with a Wait state limit of eight. There is one extra Wait state where the **wait_timeout** is asserted and the **wait_req** is de-asserted to the processor.
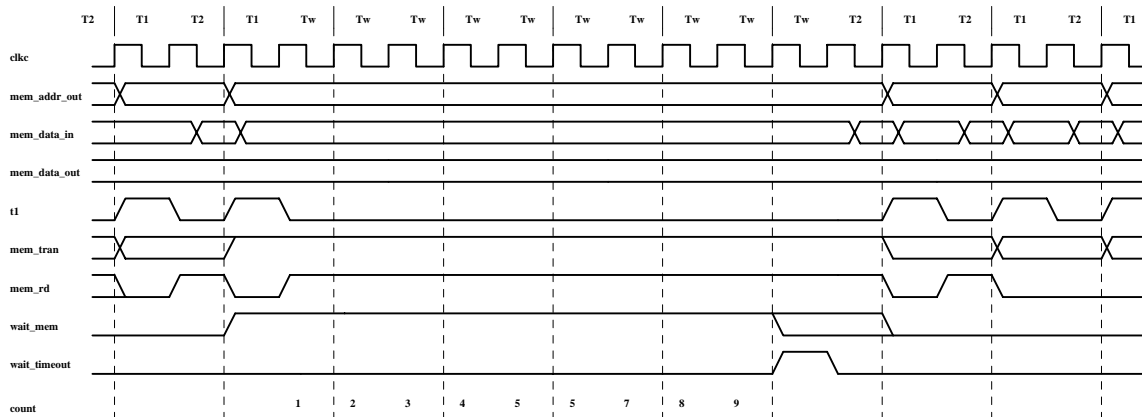
Note that the Wait controller can also be used to generate a specific number of Wait states for each transaction type by tieing the corresponding Wait request signal active and relying on the limit function to de-assert the **wait_req** signal to the processor. Obviously in this case the **wait_timeout** signal is not used. The limit value is sampled by the **t1** signal, so external logic can control the number of Wait states on a cycle-by-cycle basis (for different memory regions, for example). The minimum number of Wait states for this type of operation is two (with a limit of one and the extra one from the **wait_timeout** signal).

Watch-Dog Timer

The Y90 MPU includes a Watch-Dog Timer to reset the processor in the case of run-away code or an infinite loop. This timer is disabled after reset, and once enabled cannot be disabled by software. Once enabled, software must periodically reset the timer before it times out.

The time-out period is a multiple of 2^20 clock cycles. With a 10MHz system clock, for example, this sets the minimum time-out period to roughly 104mS. The actual Watch-Dog time-out period can be set from 1 to 256 times this period via the 8-bit **wdt_lim** value (0x00 sets the limit to 1, 0x01 sets the limit to 2, and so on). This value may be hard-coded or controlled by external I/O registers defined by the user.

Reseting the timer is a two-step process. First, the reset must be "armed", and then the actual watch-dog reset applied. The watch-dog reset must be applied within 256 clock cycles of the "arm" or it will be ignored. Similarly, any further "arm" commands while the time-out for the "arm" command is running will also be ignored. The two-step process reduces the possibility of errant code reseting the watch-dog timer. Limiting the time that the "arm" function is active does the same thing.

Rather than using I/O locations to control the watch-dog reset, the Watch-Dog Timer is accessed using dedicated instructions. This removes the possibility of DMA inadvertently interfering with the watch-dog function. It also makes reseting the timer simpler, because dedicated instructions require less setup and CPU register resources than I/O instructions.

A total of two instructions are used by the Watch-Dog Timer. One instruction "arms" the watch-dog reset function, while the other does the actual reset. The following pages detail the Watch-Dog Timer instructions.

WDARM

Watch-Dog Timer Arm

WDARM

Operation: if (WDT Arm Timer not running) begin
        start WDT Arm Timer
        end

Flags: **S:** Unaffected.
Z: Unaffected.
H: Unaffected.
P/V: Unaffected.
N: Unaffected.
C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	WDARM	11101101 01101110	4

Notes:

1. The WDT Arm Timer runs for 256 clock cyles and then stops.

WDRES

WDRES

Operation: if (WDT Arm Timer running) begin

          reset WDT Timer

          end

Flags: **S:** Unaffected.

        Z: Unaffected.

        H: Unaffected.

        P/V: Unaffected.

        N: Unaffected.

        C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	WDRES	11101101 01111110	4

System Status

The Y90 MPU includes a System Status controller to provide access to various pieces of status information to the software. It does this via a 16-byte block of information that is transferred to memory using a dedicated instruction. The structure of the System Status Block is shown below:

Address	Data
(HL+0xF)	External Status
(HL+0xE)	RTI Generator Status
(HL+0xD)	**wait_req** Timeout Status
(HL+0xC)	**dma_req** Timeout Status
(HL+0xB)	Reserved (always 0x00)
(HL+0xA)	Reserved (always 0x00)
(HL+0x9)	Fault Physical Address [27:20]
(HL+0x8)	Fault Physical Address [19:12]
(HL+0x7)	Fault Logical Address [15:8]
(HL+0x6)	Fault Logical Address [7:0]
(HL+0x5)	Trap Physical Address [27:20]
(HL+0x4)	Trap Physical Address [19:12]
(HL+0x3)	Trap Logical Address [15:8]
(HL+0x2)	Trap Logical Address [7:0]
(HL+0x1)	Trap Status
(HL)	Reset Status

The Trap Address information is the address that caused the last trap condition. The trap could have been caused by an Illegal Instruction, an active **exec_inh** signal during the fetch of the first byte of an instruction, or an active **wr_inh** signal during a memory write. The specific cause of the trap is indicated in the Trap Status byte.

The Trap Address information is only updated by a trap condition, and is not affected by a reset. The table below shows how the latched Trap Address information is related to the address of the instruction that caused the trap:

Trap condition	Latched Trap Address	Stack contents
2-byte Illegal Instruction	address of the first byte of illegal instruction	address of the first byte of illegal instruction
4-byte Illegal Instruction	address of the third byte of illegal instruction	address of the third byte of illegal instruction
Execute Inhibit	address of the first byte of execute-protected instruction	address of the byte following execute-protected first byte of instruction
Write Protect	address of the write-protected byte	address of the second byte following the last byte of the instruction that generated the trap

The Fault Address information is the memory address present on the **mem_addr_out** bus at the time that the **fault_detect** signal was activated. This information is only updated by the **fault_detect** signal, and is not affected by reset.

The remaining individual status bytes are detailed below:

Reset Status		(offset 0x0)
Bit(s)	**Value**	**Description**
7	0	No Fatal Detect since the last read.
	1	A Fatal Detect has occurred since the last read.
6	0	No Fault Detect since the last read.
	1	A Fault Detect has occurred since the last read.
5	0	No WDT Time-out since the last read.
	1	A WDT Time-out has occurred since the last read.
4	0	No Power-On Reset since the last read.
	1	A Power-On Reset has occurred. This bit will only be set for the first read after the power-on condition, as this bit is set only by the **clearb** signal.
3	0	The **reset_bus[1]** signal is currently Low.
	1	The **reset_bus[1]** signal is currently High.
2	0	The **reset_bus[1]** signal did not transition since the last read.
	1	The **reset_bus[1]** signal transitioned since the last read.
1	0	The **reset_bus[0]** signal is currently Low.
	1	The **reset_bus[0]** signal is currently High.
0	0	The **reset_bus[0]** signal did not transition since the last read
	1	The **reset_bus[1]** signal transitioned since the last read.

Trap Status		(offset (0x1)
Bit(s)	**Value**	**Description**
7:4	00	These bits are reserved and will always return zeros.
3	0	No Write Protect Trap since the last read.
	1	Write Protect Trap since the last read
2	0	No Execute Protect Trap since the last read.
	1	Execute Protect Trap since the last read.
1	0	No Illegal Instruction Trap (3rd byte) since the last read.
	1	Illegal Instruction Trap (3rd byte) since the last read
0	0	No Illegal Instruction Trap (2nd byte) since the last read.
	1	Illegal Instruction Trap (2nd byte) since the last read

dma_req Time-out Status		(offset 0xC)
Bit(s)	**Value**	**Description**
7:0		This byte reports the **dma_req** Time-out status. The read of the System Status Block clears the register. Two compile-time options, selected by a top-level 'define value, are available: DREQ_LOG or DREQ_ACC.
_LOG (default)	0x00	No **dma_req** time-out since the last read.
	0x01	At least one **dma_req** time-out since the last read.
_ACC	nn	The number of **dma_req** Timeouts since the last read of the System Status Block. The count saturates at 0xFF.

wait_req Time-out Status		(offset 0xD)
Bit(s)	**Value**	**Description**
7:0		This byte reports the **wait_req** Time-out status. The read of the System Status Block clears the register. Two compile-time options, selected by a top-level 'define value, are available: WAIT_LOG or WAIT_ACC.
_LOG (default)	0x00	No **wait_req** time-out since the last read.
	0x01	At least one **wait_req** time-out since the last read.
_ACC	nn	The number of **wait_req** Timeouts since the last read of the System Status Block. The count saturates at 0xFF.

RTI Generator Status		(offset 0xE)
Bit(s)	**Value**	**Description**
7:6		These bits are reserved and will always return zeros.
5		Current state of the **ref_clock** signal (after two stages of synchronization).
4	0	The divider value calculated using the **rti_divisor** value is 1024 or greater.
	1	The divider value calculated using the **rti_divisor** value is less than 1024. This is not allowed, and a value of 1024 will be forced for the **rti_divisor** divider value internally.
3	0	RTI Generator is free-running, having missed two or more successive **ref_clock** rising edges.
	1	RTI Generator is receiving a continuous **ref_clock** signal.
2	0	RTI Generator is not synchronized to the **ref_clock** signal. This is either because the **rti_divisor** value is zero or one, disabling the sync function, or the RTI Generator has not yet synchronized to the **ref_clock** signal.
	1	RTI Generator is synchronized to the **ref_clock** signal.
1	0	The **rti_default** value is 1024 or greater.
	1	The **rti_default** value is less than 1024. This is not allowed, and a value of 1024 will be forced for the **rti_default** internally.
0	0	RTI Generator is not using the **rti_default** value. This is either because the **rti_default** value is zero, disabling this function, or the RTI Generator has synchronized to the **ref_clock** signal.
	1	RTI Generator is using the **rti_default** value.

External Status		(offset 0xF)
Bit(s)	**Value**	**Description**
7:0		This byte reports the current state of the **ext_stat_bus** inputs, sampled during the **t1** time of the instruction reading the System Status Block. At the completion of this instruction the **ext_stat_rd** signal is asserted for one clock cycle, in case latches connected to the **ext_stat_bus** need to be cleared. This status byte can be used to report on the health of other parts of the system.

The System Status Block can only be accessed via a dedicated instruction. This instruction is detailed below:

LDM

Load Multiple

LDM (HL),src   src: SYS

Operation: count <= 0

loop: (HL) <= SysStat[count]

　　　　HL <= HL + 1

　　　　count <= count + 1

　　　　repeat loop if count != 16

Flags: **S:** Unaffected.

Z: Unaffected.

H: Unaffected.

P/V: Unaffected.

N: Unaffected.

C: Unaffected.

Addressing Modes	Assembly Syntax	Encoding	Clocks
	LDM (HL),SYS	11101101 10011010	40

Notes:

1. This instruction does not sample interrupts. The following instruction is guaranteed to be executed.

2. Attempting to load the System Status Block to a write-protected page may corrupt the System Status Block, depending on which write transactions are write-protected. In addition, since the write-protect trap status bit is cleared at the end of this instruction, the fact that a write-protect trap occured during this instruction may not be latched even though the trap is generated.

A number of conditions cause the processor to begin execution at address 0x0000, and the status block can be used to determine which condition occurred. Only the HL register pair needs to be set up (with the memory address for writing the status block) to make this determination. Determining the cause should proceeed in a specific order, because some conditions will mean that parts of the status block do not apply or are not valid:

1. Test bit 6 of the Reset Status byte. If this bit is set a Fault Condition has been detected by the hardware. Although a Fault Condition does not automatically vector through location 0x0000, external logic should always initiate a reset in the case of a Fault Condition, because after a Fault Condition nothing in the processor state can be assumed to be valid. Proceed with the steps necessary to recover from a hardware fault.

2. Test bit 4 of the Reset Status byte. If this bit is set a power-on condition (via the **clearb** signal) has occurred and the software should proceed with the initial system configuration.

3. Test bit 2 of the Reset Status byte. If this bit is set a transition on the **reset_bus[1]** signal caused a reset of the processor. The current state of the **reset_bus[1]** signal is available in bit 3 of the Reset Status byte. Proceed with the appropriate initialization code.

4. Test bit 0 of the Reset Status byte. If this bit is set a transition on the **reset_bus[0]** signal caused a reset of the processor. The current state of the **reset_bus[0]** signal is available in bit 1 of the Reset Status byte. Proceed with the appropriate initialization code.

5. Test bit 5 of the Reset Status byte. If this bit is set a WDT time-out has occurred, causing a reset. If the WDT was never enabled this step can be skipped. Proceed to determine why software did not properly prevent the WDT time-out.

6. Test bit 7 of the Reset Status byte. If this bit is set a Fatal Error condition has occurred. A Fatal Error condition is either a stack write to a write-protected page or an execute-inhibit trap in Page 0. Bits 3 and 2 of the Trap Status byte will indicate which of these conditions caused the Fatal Error condition. In either case, the most likely problem is with the programming of the MMU, although runaway stack writes because of errant code is also a possibility. The Trap Address information provides both the logical and physical address where the Fatal Error condition occurred. The Fatal Error is a trap, so the address of the offending instruction is pushed to the stack. However, this doesn't help if the stack writes are in a write-protected page.

7. Test bit 3 of the Trap Status byte. If this bit is set a Write Protect trap occurred, where software attempted to write to a protected page. The physical and logical address of the offending write is stored in the Trap Address status bytes. Because this is a

trap condition, an address related to the offending instruction is pushed to the stack, and can be retrieved from there.

8. Test bit 2 of the Trap Status byte. If this bit is set an Execute Inhibit trap occurred, where software attempted to execute code in a protected page. The physical and logical address of the offending instruction is stored in the Trap Address status bytes. Because this is a trap condition, the address of the offending instruction is also pushed to the stack.

9. Test bit 1 of the Trap Status byte. If this bit is set an Illegal Instruction trap occurred, where software attempted to execute an illegal 4-byte instruction. The physical and logical address of the offending instruction is stored in the Trap Address status bytes. Because this is a trap condition, the address of the offending instruction is also pushed to the stack.

10. Test bit 0 of the Trap Status byte. If this bit is set an Illegal Instruction trap occurred, where software attempted to execute an illegal 2-byte instruction. The physical and logical address of the offending instruction is stored in the Trap Address status bytes. Because this is a trap condition, the address of the offending instruction is also pushed to the stack.

11. If none of the preceeding tests are true, then either an RST 0 instruction causes the branch to address 0x0000 or the software branched to this location. In the case of an RST 0 instruction the return address will be on the stack. In general branches to location 0x0000 should probably be avoided, because this location is used for so many error conditions.

Real-Time Interrupt

The Y90 MPU includes a specialized timer for generating a Real-Time Interrupt (via the **rti_pulse** signal) suitable for use as the "heartbeat" for a Real-Time Operating System (RTOS). The **rti_pulse** signal may be either free-running or synchronized to an external time reference. The timer that generates the **rti_pulse** signal consists of a 24-bit counter that counts from a starting value down to one and then reloads. The reload condition is the **rti_pulse** signal. Normally the **rti_pulse** signal will be connected to the **nmi_req** input.

The free-running timer option for the **rti_pulse** signal uses the 24-bit **rti_default** value as the time constant. Power-on reset selects the free-running option, and this mode will remain in effect until at least two rising edges on the **ref_clock** signal (which is the time reference for the synchronized mode) have been recognized.

Normally the **rti_default** value will be constant, but it is certainly possible to modify it using external logic. The minimum value for **rti_default** is 0x000400 (1024), to prevent interrupt overload. This minimum is enforced by the hardware. The one exception to this minimum value is a value of 0x000000, which disables the timer to lower power consumption. The **rti_default** value is sampled when the power-on reset is removed and every time that the timer counts down to zero.

The synchronized mode for the **rti_pulse** signal is automatically selected after two rising edges on the **ref_clock** signal have been recognized. The **rti_synced** signal is activated when synchronization is achieved. The synchronization logic divides the **ref_clock** period into a number of equal-sized (+/- one CPU clock) periods for the **rti_pulse** signal.

The number of **rti_pulse** periods per **ref_clock** period is set by the **rti_divisor** value, which may be either static or controlled by an external I/O port. The synchronization logic counts the number of CPU clocks per **ref_clock** period according to the following formula (where the remainder "**R**" is obviously less than **rti_divisor**):

$$(\textbf{ref\_clock} \text{ period})/\textbf{rti\_divisor} = (\textbf{rti\_pulse} \text{ period}) + \textbf{R}$$

This calculation is done for each **ref_clock** period, and the results are used during the next **ref_clock** period. This is why two **ref_clock** rising edges are required before the synchronized mode is enabled. Rather than having one **rti_pulse** period that is longer by "**R**"

clock cycles, the first "**R**" periods of the **rti_pulse** signal are lengthened by one clock cycle.

Since it is unlikely that the **ref_clock** period will be an exact multiple of CPU clocks, there will always be some number of nanoseconds remaining at the end of an **rti_divisor** number of **rti_pulse** periods. This number of nanoseconds will always be less than the CPU clock period and will effectively be added to the next **ref_clock** period for the next calculation. In this manner the calculation will always track the **ref_clock** period.

However, because of inevitable circuit asymmetries, jitter, or drift in the CPU clock and jitter or drift in the **ref_clock** signal, it is possible for a phase error to accumulate over time, causing the **rti_pulse** position to "drift" relative to the position of the **ref_clock** signal. To compensate for this possibility, there is an independent digital phase-locked loop circuit that will always keep the relative positions of the **ref_clock** signal rising edge and the generated **rti_pulse** signal within one CPU clock period of each other. This circuit will adjust the last **rti_pulse** period in a **ref_clock**-relative frame by +/- one clock as required to bring the **rti_pulse** to within one clock (before or after) the detected ref_clock signal edge.

The continuous calculation, plus the digital phase-locked loop, allows the **rti_pulse** period to track variations in either the **ref_clock** period or in the period of the CPU clock. In the event that the **ref_clock** signal disappears the **rti_default** value will be used until **ref_clock** signal rising edges are again detected. The **rti_synced** signal is deasserted if three successive **ref_clock** edges are missed to indicate that the **rti_pulse** is being generated automatically without input from the **ref_clock** period.

The **rti_pulse** signal is offset (delayed) from the **ref_clock** signal by about thirteen clock cycles. This is because the **ref_clock** signal passes through a digital filter that rejects pulses less than eleven clock cycles wide, which leads to a delay in recognizing edges on the **ref_clock** signal. The synchronization logic does not attempt to compensate for this delay.

Setting the **rti_divisor** value to 0x01 forces the synchronization logic to pass the filtered **ref_clock** edge-detect signal directly to the **rti_pulse** signal. In this case if the **ref_clock** signal disappears, so will the **rti_pulse** signal.

Setting the **rti_divisor** value to 0x00 will disable the synchronized mode. This disable function takes effect immediately, and the free-running option takes over (assuming that the **rti_default** value is non-zero).

The 24-bit width of the timer used to generated the **rti_pulse** signal, the 7-bit width of the **rti_divisor** value, the period of the **ref_clock** signal, and the CPU clock frequency determine the limits for the period of the **rti_pulse** signal. The table below shows some examples of these limits.

176

CPU clock	ref_clock period	rti_divisor	rti_pulse period	rti_pulse count
10 MHz	500mS	20	25mS	0x03D090
	500mS	4	125mS	0x1312D0
	1 S	50	20mS	0x030D40
	1 S	5	200mS	0x1E8480
	2 S	125	16mS	0x027100
30 MHz	500mS	80	6.25mS	0x02DC6C
	500mS	20	25mS	0x0B71B0
	1 S	100	10mS	0x0493E0
	1 S	10	100mS	0x2DC6C0
	2 S	125	16mS	0x075300
	2 S	4	500mS	0xE4E1C0

Top-level Verilog

The Verilog code for the **y90_mpu** design is shown below to illustrate how the individual modules connect.

```
/****************************************************************************************/
/**                                                                                  **/
/** COPYRIGHT (C) 2012, SYSTEMYDE INTERNATIONAL CORPORATION, ALL RIGHTS RESERVED      **/
/**                                                                                  **/
/** processor top level                                         Rev 0.0  11/10/2012 **/
/**                                                                                  **/
/****************************************************************************************/
module y90_mpu (dma_ack_bus, drq_timeout, ext_stat_rd, fatal_detect, fault_detect, halt_tran,
                iack_tran, int_ack_bus, io_addr_out, io_data_out, io_read, io_strobe,
                io_tran, mem_addr_out, mem_data_out, mem_rd, mem_tran, mem_wr, nmiack_tran,
                resetb, rti_pulse, rti_synced, sleep_tran, t1, wait_timeout, clearb, clkc,
                dma_req_bus, drq_act_lim, drq_idl_lim, en_prftch, exec_inh, ext_stat_bus,
                iack_wait_lim, int_req_bus, io_data_in, io_wait_lim, ivec_offset,
                mem_data_in, mem_wait_lim, nmi_req, ref_clock, reset_bus, rti_default,
                rti_divisor, wait_iack, wait_io, wait_mem, wdt_lim, wr_inh);

  input         clearb;        /* master (test) reset                                */
  input         clkc;          /* main cpu clock                                     */
  input         en_prftch;     /* prefetch enable                                    */
  input         exec_inh;      /* execute inhibit                                    */
  input         nmi_req;       /* nmi request                                        */
  input         ref_clock;     /* rti reference clock                                */
  input         wait_iack;     /* iack wait request                                  */
  input         wait_io;       /* i/o wait request                                   */
  input         wait_mem;      /* memory wait request                                */
  input         wr_inh;        /* memory write inhibit                               */
  input   [1:0] reset_bus;     /* system reset bus                                   */
  input   [2:0] ivec_offset;   /* interrupt vector offset                            */
  input   [4:0] drq_act_lim;   /* dma_req active maximum                             */
  input   [4:0] drq_idl_lim;   /* dma_req idle minimum                               */
  input   [4:0] iack_wait_lim; /* wait_iack active maximum                           */
  input   [4:0] io_wait_lim;   /* wait_io active maximum                             */
  input   [4:0] mem_wait_lim;  /* wait_mem active maximum                            */
  input   [6:0] rti_divisor;   /* rti divisor                                        */
  input   [7:0] dma_req_bus;   /* dma request input bus                              */
  input   [7:0] ext_stat_bus;  /* external status bus                                */
  input   [7:0] io_data_in;    /* i/o input data bus                                 */
  input   [7:0] mem_data_in;   /* memory input bus                                   */
  input   [7:0] wdt_lim;       /* watchdog timeout limit (n+1 x 2^20-1 clocks)       */
  input  [15:0] int_req_bus;   /* interrupt request input bus                        */
  input  [15:0] int_req_en;    /* interrupt request enable bus                       */
  input  [15:0] nmi_addr;      /* nmi service routine start address                  */
  input  [23:0] rti_default;   /* rti default period                                 */
  output        drq_timeout;   /* dma request timeout                                */
  output        ext_stat_rd;   /* external status read                               */
  output        fatal_detect;  /* fatal error detect                                 */
  output        fault_detect;  /* fault detect                                       */
  output        halt_tran;     /* halt transaction                                   */
  output        iack_tran;     /* interrupt acknowledge transaction                  */
  output        io_read;       /* i/o read enable                                    */
  output        io_strobe;     /* i/o data strobe                                    */
  output        io_tran;       /* i/o transaction                                    */
  output        mem_rd;        /* memory read enable                                 */
  output        mem_tran;      /* memory transaction                                 */
  output        mem_wr;        /* memory write enable                                */
  output        nmiack_tran;   /* nmi acknowledge transaction                        */
```

```
output          resetb;         /* internal (user) reset                          */
output          rti_pulse;      /* rti (real-time interrupt) pulse                */
output          rti_synced;     /* rti is synced                                  */
output          sleep_tran;     /* sleep transaction                              */
output          t1;             /* first clock of transaction                     */
output          wait_timeout;   /* wait_req timeout pulse                          */
output  [7:0] dma_ack_bus;      /* dma acknowledge output bus                      */
output  [7:0] io_data_out;      /* i/o output data bus                             */
output  [7:0] mem_data_out;     /* memory output data bus                          */
output [15:0] int_ack_bus;      /* interrupt acknowledge output bus                */
output [15:0] io_addr_out;      /* i/o address bus                                 */
output [27:0] mem_addr_out;     /* memory address bus                              */


/**********************************************************************************/
/*                                                                              */
/* signal declarations                                                          */
/*                                                                              */
/**********************************************************************************/
wire          burst_done;                               /* burst done               */
wire          dma_ack;                                  /* dma acknowledge          */
wire          dma_req;                                  /* dma request              */
wire          drq_prio_in;                              /* dma priority chain input */
wire          drq_timeout;                              /* dma request timeout      */
wire          ext_stat_rd;                              /* external status read     */
wire          fatal_detect;                             /* fatal error detected     */
wire          fault_detect;                             /* illegal state detected   */
wire          ftch_tran;                                /* inst fetch transaction   */
wire          halt_tran;                                /* halt transaction         */
wire          iack_tran;                                /* int ack transaction      */
wire          imd2_reg;                                 /* int mode 2               */
wire          inst2_trap;                               /* illegal 2-byte instruction */
wire          inst3_trap;                               /* illegal 3-byte instruction */
wire          int_req;                                  /* interrupt request        */
wire          io_read;                                  /* i/o read enable          */
wire          io_strobe;                                /* i/o data strobe          */
wire          io_tran;                                  /* i/o transaction          */
wire          ivec_rd;                                  /* interrupt vector enable  */
wire          ld_if1;                                   /* load inst byte 1         */
wire          ld_mem_addr;                              /* update memory address    */
wire          mem_rd;                                   /* memory read enable       */
wire          mem_tran;                                 /* memory transaction       */
wire          mem_wr;                                   /* memory write enable      */
wire          mem_wr_c;                                 /* memory write enable (cpu) */
wire          mmu_msb;                                  /* msb select               */
wire          mmu_swap;                                 /* swap active/inactive line */
wire          mwr_tran;                                 /* memory write transaction */
wire          nmiack_tran;                              /* nmi ack transaction      */
wire          output_inh;                               /* disable cpu outputs      */
wire          resetb;                                   /* internal (user) reset    */
wire          reti_tran;                                /* reti transaction         */
wire          rti_pulse;                                /* rti pulse                */
wire          rti_synced;                               /* rti is synced            */
wire          sleep_tran;                               /* sleep transaction        */
wire          stk_tran;                                 /* stack transaction        */
wire          t1;                                       /* first clock of transaction */
wire          wait_req;                                 /* wait request             */
wire          wait_st;                                  /* wait state identifier    */
wire          wait_timeout;                             /* wait_req timeout pulse    */
wire          wdt_arm;                                  /* watchdog arm pulse       */
wire          wdt_hit;                                  /* watchdog hit pulse       */
wire          wdt_timeout;                              /* watchdog timeout pulse   */
wire          wr_brst;                                  /* burst write              */
wire    [1:0] ld_init;                                  /* st mach init (sys, mmu)  */
wire    [3:0] ctr_reg;                                  /* burst/mlt counter        */
wire    [3:0] log_addr_page;                            /* logical address page     */
wire    [5:0] rti_stat_bus;                             /* rti status bus           */
wire    [7:0] io_data_out;                              /* i/o output data bus      */
wire    [7:0] ivec_data_in;                             /* interrupt vector bus     */
wire    [7:0] mem_data_out;                             /* memory output data bus   */
wire    [7:0] mmu_data;                                 /* mmu read data            */
wire    [7:0] mem_data_in_s;                            /* mem input data bus (sys) */
wire   [15:0] addr_reg_in;                              /* processor logical address */
wire   [15:0] int_ack_bus;                              /* int ack output bus       */
wire   [15:0] io_addr_out;                              /* i/o address bus          */
wire   [27:0] mem_addr_out;                             /* memory address bus       */


/**********************************************************************************/
/*                                                                              */
```

```
/* processor                                                                    */
/*                                                                              */
/***********************************************************************************/
y90_core CPU       ( .addr_reg_in(addr_reg_in), .burst_done(burst_done), .ctr_reg(ctr_reg),
                     .dma_ack(dma_ack), .fault_detect(fault_detect), .ftch_tran(ftch_tran),
                     .halt_tran(halt_tran), .iack_tran(iack_tran), .imd2_reg(imd2_reg),
                     .inst2_trap(inst2_trap), .inst3_trap(inst3_trap),
                     .io_addr_out(io_addr_out), .io_data_out(io_data_out),
                     .io_read(io_read), .io_strobe(io_strobe), .io_tran(io_tran),
                     .ivec_rd(ivec_rd), .ld_if1(ld_if1), .ld_init(ld_init),
                     .ld_mem_addr(ld_mem_addr),
                     .mem_addr_out({log_addr_page, mem_addr_out[11:0]}),
                     .mem_data_out(mem_data_out), .mem_rd(mem_rd), .mem_tran(mem_tran),
                     .mem_wr(mem_wr_c), .mmu_msb(mmu_msb), .mmu_swap(mmu_swap),
                     .mwr_tran(mwr_tran), .nmiack_tran(nmiack_tran),
                     .output_inh(output_inh), .reti_tran(reti_tran),
                     .sleep_tran(sleep_tran), .stk_tran(stk_tran), .t1(t1),
                     .wait_st(wait_st), .wdt_arm(wdt_arm), .wdt_hit(wdt_hit),
                     .wr_brst(wr_brst), .clearb(clearb), .clkc(clkc), .dma_req(dma_req),
                     .en_prftch(en_prftch), .frc_imd2(1'b0), .int_req(int_req),
                     .io_data_in(io_data_in), .ivec_data_in(ivec_data_in),
                     .mem_data_in(mem_data_in_s), .nmi_addr(nmi_addr), .nmi_req(nmi_req),
                     .resetb(resetb), .wait_req(wait_req) );


/***********************************************************************************/
/*                                                                              */
/* memory management unit                                                       */
/*                                                                              */
/***********************************************************************************/
mmu_pag MMU        ( .mmu_addr_out(mem_addr_out[27:12]), .mmu_data(mmu_data),
                     .addr_reg_in(addr_reg_in[15:12]), .clkc(clkc), .ctr_reg(ctr_reg),
                     .input_bus(mem_data_in_s), .ld_mem_addr(ld_mem_addr),
                     .mmu_msb(mmu_msb), .mmu_swap(mmu_swap), .output_inh(output_inh),
                     .resetb(resetb), .wr_brst(wr_brst) );


/***********************************************************************************/
/*                                                                              */
/* system management                                                            */
/*                                                                              */
/***********************************************************************************/
sys_mgt SYS        ( .ext_stat_rd(ext_stat_rd), .resetb(resetb),
                     .mem_data_in_s(mem_data_in_s), .mem_wr_s(mem_wr), .clearb(clearb),
                     .burst_done(burst_done), .clkc(clkc), .ctr_reg(ctr_reg),
                     .drq_timeout(drq_timeout), .exec_inh(exec_inh),
                     .ext_stat_bus(ext_stat_bus), .fatal_detect(fatal_detect),
                     .fault_detect(fault_detect), .ftch_tran(ftch_tran),
                     .inst2_trap(inst2_trap), .inst3_trap(inst3_trap), .ld_if1(ld_if1),
                     .ld_init(ld_init), .log_addr_page(log_addr_page),
                     .mem_addr_out(mem_addr_out), .mem_data_in(mem_data_in),
                     .mem_wr(mem_wr_c), .mmu_data(mmu_data), .mwr_tran(mwr_tran),
                     .reset_bus(reset_bus), .rti_stat_bus(rti_stat_bus),
                     .stk_tran(stk_tran), .t1(t1), .wait_timeout(wait_timeout),
                     .wdt_timeout(wdt_timeout), .wr_inh(wr_inh) );


/***********************************************************************************/
/*                                                                              */
/* interrupt control                                                            */
/*                                                                              */
/***********************************************************************************/
int_ctl INT_CTL    ( .int_ack_bus(int_ack_bus), .int_prio_out(), .int_req(int_req),
                     .ivec_data_out(ivec_data_in), .clkc(clkc), .iack_tran(iack_tran),
                     .imd2_reg(imd2_reg), .int_prio_in(2'b11), .int_req_bus(int_req_bus),
                     .int_req_en(int_req_en), .ivec_offset(ivec_offset),
                     .ivec_rd(ivec_rd), .resetb(resetb), .reti_tran(reti_tran),
                     .wait_st(wait_st) );


/***********************************************************************************/
/*                                                                              */
/* dma request control                                                          */
/*                                                                              */
/***********************************************************************************/
drq_lim DRQ_LIM    ( .drq_prio_out(drq_prio_in), .drq_timeout(drq_timeout), .clkc(clkc),
                     .dma_ack(dma_ack), .drq_act_lim(drq_act_lim),
                     .drq_idl_lim(drq_idl_lim), .resetb(resetb) );

drq_ctl DRQ_CTL    ( .dma_ack_bus(dma_ack_bus), .dma_req(dma_req), .drq_prio_out(),
                     .clkc(clkc), .dma_ack(dma_ack), .dma_req_bus(dma_req_bus),
                     .drq_prio_in(drq_prio_in), .resetb(resetb) );
```

```verilog
/***************************************************************************/
/*                                                                         */
/* real-time interrupt generator                                           */
/*                                                                         */
/***************************************************************************/
rti_top RTI_GEN  ( .rti_pulse(rti_pulse), .rti_stat_bus(rti_stat_bus),
                   .rti_synced(rti_synced), .clearb(clearb), .clkc(clkc),
                   .ref_clock(ref_clock), .rti_default(rti_default),
                   .rti_divisor(rti_divisor) );


/***************************************************************************/
/*                                                                         */
/* wait-state limit                                                        */
/*                                                                         */
/***************************************************************************/
wait_lim WAIT_LIM ( .wait_req(wait_req), .wait_timeout(wait_timeout), .clkc(clkc),
                    .iack_tran(iack_tran), .iack_wait_lim(iack_wait_lim),
                    .io_tran(io_tran), .io_wait_lim(io_wait_lim), .mem_tran(mem_tran),
                    .mem_wait_lim(mem_wait_lim), .nmiack_tran(nmiack_tran),
                    .resetb(resetb), .t1(t1), .wait_iack(wait_iack), .wait_io(wait_io),
                    .wait_mem(wait_mem) );


/***************************************************************************/
/*                                                                         */
/* watch-dog timer                                                         */
/*                                                                         */
/***************************************************************************/
wdt_top WDT_TOP  ( .wdt_timeout(wdt_timeout), .clkc(clkc), .wdt_arm(wdt_arm),
                   .wdt_hit(wdt_hit), .wdt_lim(wdt_lim), .resetb(resetb) );

endmodule
```

HDL Compile Options

The Y90 design is configured by HDL compile options contained in the Verilog file called version.v. These compile options are controlled by Verilog 'define constructs, which are the tested using the 'ifdef keyword to select different sections of Verilog code for compilation. In all but one case there are just two choices, and the code is set up so that if no 'define is actually defined the default is selected. The available compile options are described below.

The primary option selects between the Y90 CPU, the Y90 MPU and the Z180-compatible version called the Y90-180, which is documented in a separate specification. This is the only case where one of three options must be selected. The three options are:

Y90_CPU is the default selection, and selects a CPU-only implementation.

Y90_MPU selects the full MPU implementation, and all of the other compile options are also available in this case.

Y90-180 selects the version compatible with the Z180. This version contains a segmented MMU and the full complement of Z180 peripherals. This option is documented in a separate specification.

The CCF option selects between the one documented difference (except for the Z180 instruction set additions) in the instruction operation for the Z80 and the Z180. The two options are:

Z80_CCF is the default and selects the Z80-style operation for the H flag in the case of the CCF instruction.

Z180_CCF selects the Z180-style operation for the H flag in the case of the CCF instruction.

The Multiply option is used only in the case of the Y90 CPU or Y90 MPU, and controls the implementation of the multiply (MLT) instruction. The default case is slower, in terms of clock cycles, but actually allows for a smaller clock cycle time. The two options are:

183

MUL_NORM is the default and selects the normal MLT implementation, which requires 14 clock cycles. This option should always be selected unless the technology provides for fast parallel multipliers. This option should always be selected for FPGA implementations.

MUL_FAST reduces the MLT execution time to just 4 clock cycles, but requires that the technology support fast multipliers. This option can be considered with ASIC implementations.

The DREQ option applies only to the Y90 MPU and controls the error logging of the DMA request limit function. The two options are:

DREQ_LOG is the default and selects the option to merely log the occurance of a DMA request time-out in the System Status Block.

DREQ_ACC selects the option to accumulate a count of the occurances of DMA request time-outs in the System Status Block. This count is eight bits wide and saturates at 0xFF.

The WAIT option applies only to the Y90 MPU and controls the error logging of the Wait request limit function. The two options are:

WAIT_LOG is the default and selects the option to merely log the occurance of a Wait request time-out in the System Status Block.

WAIT_ACC selects the option to accumulate a count of the occurances of Wait request time-outs in the System Status Block. This count is eight bits wide and saturates at 0xFF.