TSK52x MCU

Summary

Core Reference
CR0116 (v1.5) December 12, 2005

The TSK52x is a fully functional, 8-bit microcontroller, incorporating the Harvard architecture. This core reference includes architectural and hardware descriptions, instruction sets and on-chip debugging functionality for the TSK52x family.

The TSK52x is an 8-bit embedded controller that executes all ASM51 instructions and is instruction set compatible with the 80C31.

Features

- Control Unit
  - 8-bit Instruction decoder
  - Reduced instruction cycle time up to 12 times.
- Arithmetic-Logic Unit
  - 8 bit arithmetic and logical operations
  - Boolean manipulations
  - 8 x 8 bit multiplication and 8 / 8 bit division.
- 32-bit Input/Output ports
  - Four 8-bit I/O ports
  - Alternate port functions such as external interrupts and serial interface are separated, providing extra port pins when compared with the standard 8051.
- Interrupt Controller
  - Four Priority Levels
  - 7 external interrupts
- Internal Data Memory interface
  - Can address up to 256 Bytes of Data memory Space.

- External Memory interface
  - Can address up to 64 KB of external Program memory space
  - Can address up to 64 KB of external Data memory space
  - De-multiplexed Address/Data Bus to allow easy connection to memories
  - Variable length code fetch and MOVC to access fast/slow Program memory
  - Variable length MOVX to access fast/slow RAM or peripherals
- Wishbone-compliant (**TSK52B_W and TSK52B_WD only**)

Performance

The architecture eliminates redundant bus states and implements parallel execution of fetch and execution phases. Since a cycle is aligned with memory fetch when possible, most of the 1-byte instructions are performed in a single cycle. The TSK52x uses 1 clock cycle per machine (instruction) cycle. This leads to a more enhanced and efficient performance with respect to the industry standard 8051 processor working with the same clock frequency (in fact, the execution of instructions is an average eight times faster on the TSK52x).

The standard 8051 has a 12-clock architecture. A machine (instruction) cycle needs 12 clock cycles to execute to completion and most instructions require either one or two machine cycles. Therefore, with the exception of MUL and DIV, the 8051 uses either 12 or 24 clock cycles for each instruction. Furthermore, each cycle in the 8051 uses two memory fetches. In many cases the second fetch is a dummy and extra clock cycles are wasted.

Table 1 below shows the speed advantage of the TSK52x over the standard 8051. A speed advantage of 12 means that the TSK52x performs the same instruction twelve times faster that the 8051.

Table 1. Speed advantage summary

Speed advantage	Number of instructions	Number of opcodes
24	1	1
12	27	83
9.6	2	2
8	16	38
6	44	89
4.8	1	2
4	18	31
3	2	9
Average: 8.0	Sum: 111	Sum: 255

The average speed advantage is 8.0. However, the real speed improvement seen in any system will depend on the mixture of instructions used.

Available Devices

The following two variants of the microcontroller are available:

TSK52A	-	Standard version of the core
TSK52B_W	-	Wishbone-compliant version of the core

In addition, a corresponding debug-enabled (OCD) version of each variant is also available (TSK52A_D and TSK52B_WD respectively).

Note: Throughout this document, differences between core variants are listed in terms of the standard core devices (TSK52A and TSK52B_W). Unless specified otherwise, the feature/description applies to the debug-enabled version of the variant (TSK52A_D and TSK52B_WD) in exactly the same way.

These devices can be found in the FPGA Processors integrated library (`\Program Files\Altium Designer 6\Library\Fpga\FPGA Processors.IntLib`).

Architectural overview

Symbols

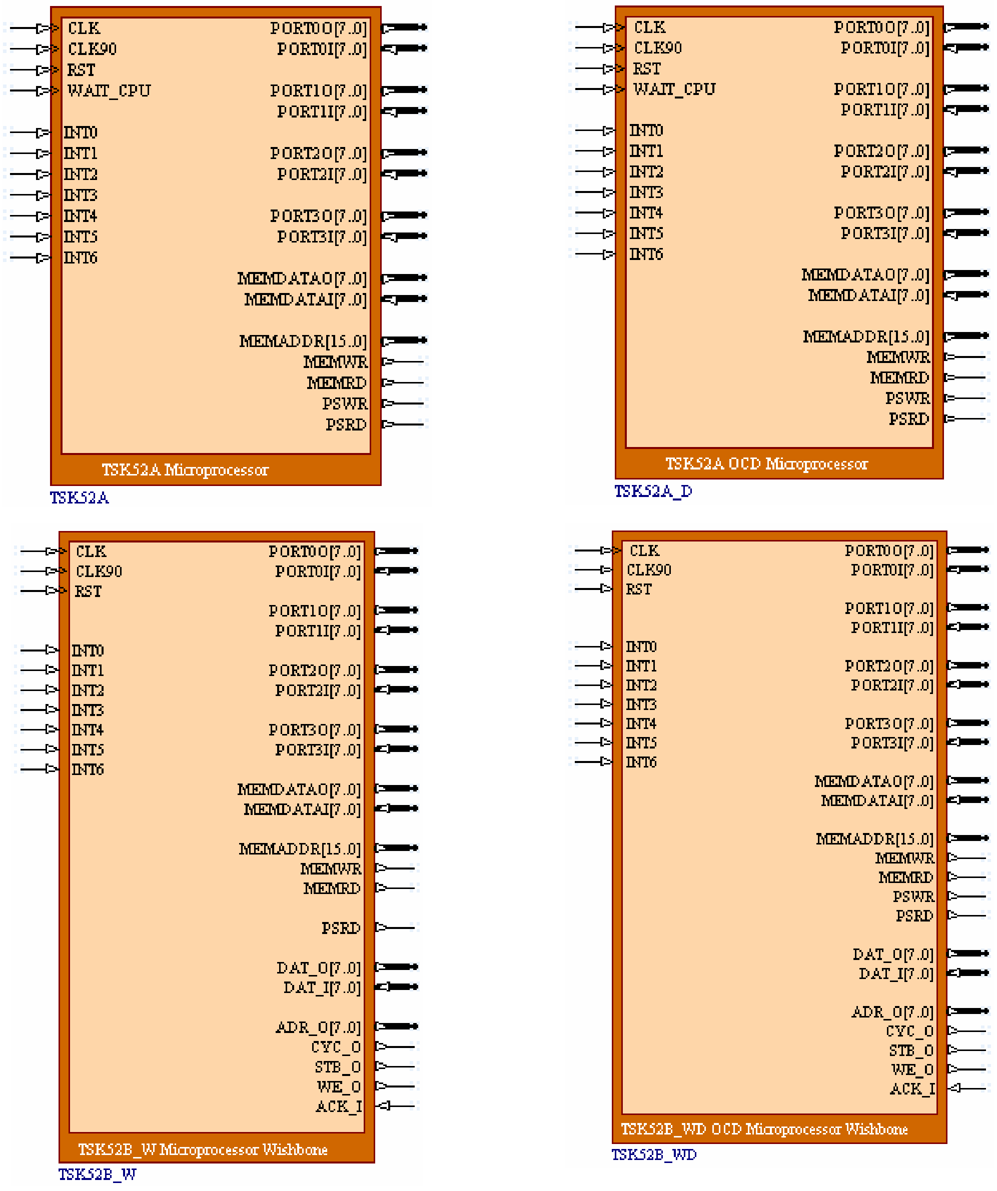


Figure 1. TSK52x family symbols

Pin description

The pinout of the TSK52x has not been fixed to any specific device I/O, thereby allowing flexibility with user application. The TSK52x contains only unidirectional pins - inputs or outputs. To simplify using the bidirectional ports (PORT0-3), the schematic symbol includes a bus pin for each direction, allowing them to be wired independently. Configuration of bus direction is performed under program control.

Table 2. TSK52x Pin description

Name	Type	Polarity/Bus size	Description
Control Signals			
CLK	I	Rise	External system clock (used for internal clock counters and all other synchronous circuitry)
CLK90	I	Rise	Second external clock with a phase lag of 90 Degrees in relation to CLK.
RST	I	High	External system reset. A high on this pin while the external system clock (CLK) is running resets the device
WAIT_CPU[1]	I	High	When this signal is active, operation of the CPU is halted.
Interrupt Signals			
INT0	I	High/Rise	External interrupt 0
INT1	I	High/Rise	External interrupt 1
INT2	I	Fall/Rise	External interrupt 2
INT3	I	Fall/Rise	External interrupt 3
INT4	I	Rise	External interrupt 4
INT5	I	Rise	External interrupt 5
INT6	I	Rise	External interrupt 6
I/O Port Interface Signals			
PORT0I PORT0O	I O	8 8	**Port 0** is an 8-bit bi-directional I/O port with separated inputs and outputs
PORT1I PORT1O	I O	8 8	**Port 1** is an 8-bit bi-directional I/O port with separated inputs and outputs
PORT2I PORT2O	I O	8 8	**Port 2** is an 8-bit bi-directional I/O port with separated inputs and outputs

[1] TSK52A and TSK52A_D only.

Name	Type	Polarity/Bus size	Description
PORT3I PORT3O	I O	8 8	**Port 3** is an 8-bit bi-directional I/O port with separated inputs and outputs
External Memory Interface Signals			
MEMDATAO	O	8	External memory output
MEMDATAI	I	8	External memory input
MEMADDR	O	16	External address bus
MEMWR	O	High	External Data memory write enable
MEMRD	O	High	External Data memory output enable
PSRD	O	High	External Program memory output enable
PSWR[2]	O	High	External Program memory write enable
Wishbone Interface Signals (TSK52B_W and TSK52B_WD only)			
DAT_O	O	8	Data to be sent to an external Wishbone slave device
DAT_I	I	8	Data received from an external Wishbone slave device
ADR_O	O	8	Standard Wishbone address bus, used to select an internal register of a connected Wishbone slave device for writing to/reading from
CYC_O	O	High	Cycle signal. When asserted, indicates the start of a valid Wishbone cycle
STB_O	O	High	Strobe signal. When asserted, indicates the start of a valid Wishbone data transfer cycle
WE_O	O	Level	Write enable signal. Used to indicate whether the current local bus cycle is a Read or Write cycle. 0 = Read 1 = Write
ACK_I	I	High	Standard Wishbone device acknowledgement signal. When this signal goes High, external Wishbone slave device has finished execution of the requested action and the current bus cycle is terminated

[2] This signal is not available in the TSK52B_W.

Memory organization

Memory in the TSK52x is organized into three distinct areas:

- Program memory (external ROM)
- External Data memory (external RAM)
- Internal Data memory (internal RAM).

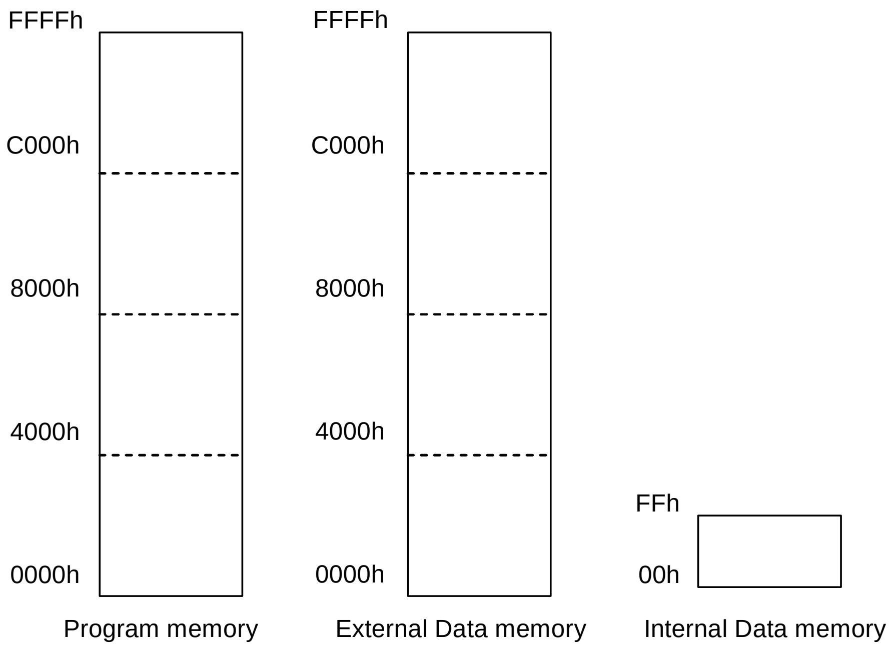


Figure 2. Memory map

Program memory

The TSK52x can address up to 64 kB of Program memory space, from 0000h to FFFFh.

The External Bus Interface services Program memory when the PSRD signal is active. Program memory is read when the CPU performs fetching instructions or MOVC.

After a reset has been issued, the CPU starts program execution from location 0000h.

The lower part of the Program memory includes the interrupt and reset vectors. The interrupt vectors are spaced at 8-byte intervals, starting from 0003h, for External Interrupt 0.

Variable length code fetching and MOVC instructions enable access to fast or slow ROM. Three high-order bits of the CKCON register (in the Clock Control Unit) control wait state memory cycles. Setting these wait state bits to '1' allows access to very slow ROM.

Table 3 shows how the signals of the External Memory Interface change when wait values are set from 0 to 7. The widths of the signals are counted in CLK cycles. The post-reset state of the CKCON register, which is in bold in the table, performs the fetching cycles or MOVC instructions without wait states.

Table 3. Wait state memory cycle width

CKCON register			**Wait value**	**Read signals width**	
CKCON.6	CKCON.5	CKCON.4		MEMADDR	PSRD
0	**0**	**0**	**0**	**1**	**1**
0	0	1	1	2	2
0	1	0	2	3	3
0	1	1	3	4	4
1	0	0	4	5	5
1	0	1	5	6	6
1	1	0	6	7	7
1	1	1	7	8	8

Data memory

External Data memory

The TSK52x can address up to 64 KB of external Data memory space, from 0000h to FFFFh.

The External Bus Interface services Data memory when the MEMRD or MEMWR signals are active. The TSK52x writes into external Data memory when the CPU executes MOVX @Ri,A or MOVX @DPTR,A instructions. The external Data memory is read when the CPU executes MOVX A,@Ri or MOVX A,@DPTR instructions.

The variable length MOVX instructions enable access to fast or slow external RAM and external peripherals. Three low-order bits of the CKCON register (in the Clock Control Unit) control stretch memory cycles. Setting these stretch bits to '1' allows access to very slow external RAM or external peripherals.

Table 4 shows how the signals of the External Memory Interface change when stretch values are set from 0 to 7. The widths of the signals are counted in CLK cycles. The post-reset state of the CKCON register, which is in bold in the table, performs the MOVX instructions with a value of stretch equal to 1.

Table 4. Stretch memory cycle width

CKCON register			**Stretch value**	**Read signals width**		**Write signal width**	
CKCON.2	CKCON.1	CKCON.0		MEMADDR	MEMRD	MEMADDR	MEMWR
0	0	0	0	1	1	2	1
0	**0**	**1**	**1**	**2**	**2**	**3**	**1**
0	1	0	2	3	3	4	2
0	1	1	3	4	4	5	3

CKCON register			Stretch value	Read signals width		Write signal width	
1	0	0	4	5	5	6	4
1	0	1	5	6	6	7	5
1	1	0	6	7	7	8	6
1	1	1	7	8	8	9	7

There are two types of MOVX instruction, differing in whether they provide an 8-bit or 16-bit indirect address to the external Data RAM.

In the first type, the contents of R0 or R1 in the current register bank provide an 8-bit address.

Eight bits are sufficient for external I/O expansion decoding or a relatively small RAM array. For somewhat larger arrays there are two methods to extend the 8-bit address to 16 bits:

- The first method is to use any output port pins to output higher-order address bits. These pins would be controlled by an output instruction preceding the MOVX instruction
- The second method is to use the external Data memory paging register, XP. Using the XP register makes accessing data within a page more efficient, since the page is held in the XP register and only R0 or R1 needs to be changed. With this method, output ports are left free to serve any other purpose.

In the second type of MOVX instruction, the data pointer generates a 16-bit address.

Internal Data memory

The TSK52x has a 512 byte block of RAM dedicated for use as internal Data memory. It should be noted, however, that although the physical size of the block is 512 bytes, only 256 bytes can be used for internal Data memory. This RAM cannot be upgraded in size. The internal Data memory interface is therefore not exposed to the user through the schematic symbol.

The 256 bytes of memory space (00h to FFh) can be accessed by either direct or indirect addressing (where supported). An internal Data memory address is always 1 byte in width.

The upper 128 bytes contain the Special Function Registers (SFRs). This area of internal Data memory is accessible only by direct addressing.

The lower 128 bytes contain work registers and bit-addressable memory. The lower 48 bytes of this area of memory space are further divided as follows:

- The lower 32 bytes (00h – 1Fh) form four banks of eight registers (R0-R7). The RS0 and RS1 bits in the Program Status Word register (PSW) select which bank is currently in use.
- The next 16 bytes (20h – 2Fh) form a block of bit-addressable memory space, covering the bit address range 00h-7Fh.

All of the bytes in this lower half of the internal Data memory space are accessible through direct or indirect addressing.

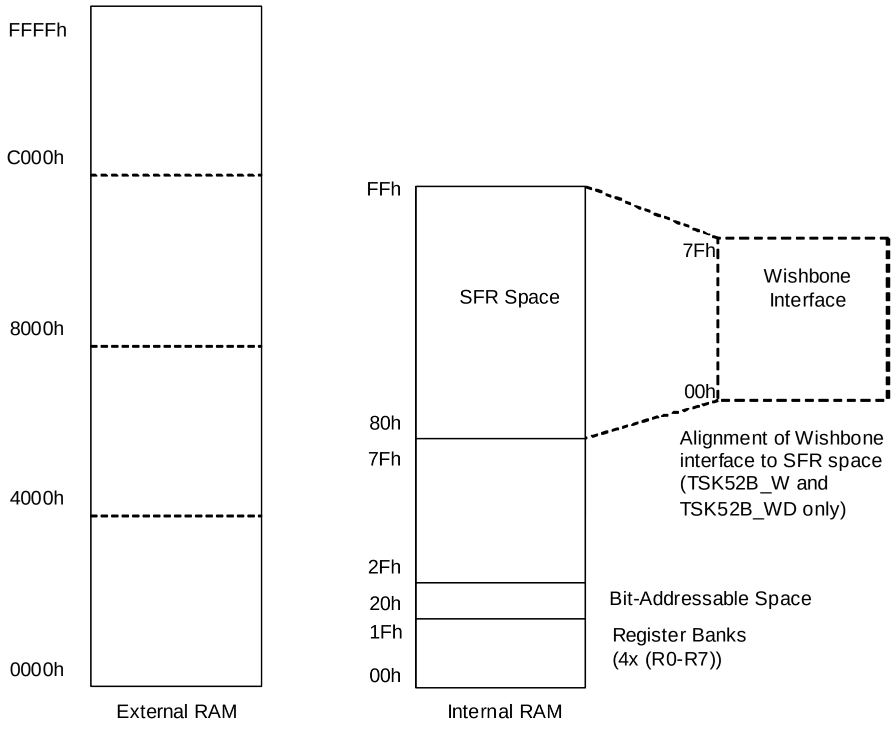


Figure 3. Data memory map showing partitioning of internal RAM space

With respect to the Wishbone-compliant versions of the microcontroller (TSK52B_W and TSK52B_WD), all Wishbone peripheral devices map into the SFR address range of the Internal RAM space – overlapping the existing SFRs. Therefore, the only addresses available for accessing Wishbone devices are those in the SFR space that do not have any predefined function. The predefined SFR addresses are disabled on the Wishbone interface.

The SFR address 80h corresponds to the address 00h on the Wishbone interface. For example, a Wishbone peripheral device at address 08h on the Wishbone interface would be accessed in software at address 88h of the SFR space.

Special Function Registers

Special Function Registers location

As illustrated in the previous section, Special Function Registers (SFRs) reside in the top 128 bytes of the internal RAM space. A map of the Special Function Registers is shown in Table 5.

Table 5. Special Function Registers location

Hex\Bin	**X000**	**X001**	**X010**	**X011**	**X100**	**X101**	**X110**	**X111**	**Bin/Hex**
F8	WBT0[3]	WBT1[3]							**FF**
F0	B								**F7**
E8									**EF**
E0	ACC								**E7**
D8									**DF**
D0	PSW								**D7**
C8									**CF**
C0	IRCON								**C7**
B8	IEN1	IP1							**BF**
B0	P3								**B7**
A8	IEN0	IP0							**AF**
A0	P2								**A7**
98								XP	**9F**
90	P1								**97**
88							CKCON		**8F**
80	P0	SP	DPL	DPH				PCON	**87**

For the non-Wishbone versions of the microcontroller (TSK52A and TSK52A_D), only 18 addresses are occupied, the others are not implemented. Read access to unimplemented addresses will return undefined data, while writes will have no effect.

For the Wishbone-compliant versions (TSK52B_W and TSK52B_WD), an additional 2 addresses are occupied by two dedicated timing registers – WBT0 and WBT1. The remaining 108 addresses in the SFR space can be used to access any 8-bit compatible Wishbone slave devices.

[3] Wishbone-compliant versions only (TSK52B_W, TSK52B_WD).

Special Function Registers reset values

Table 6. Special Function Registers reset values

Register	Location	Reset value	Description
P0	80h	FFh	Port 0
SP	81h	07h	Stack Pointer
DPL	82h	00h	Data Pointer Low 0
DPH	83h	00h	Data Pointer High 0
PCON	87h	00h	Power Control register
CKCON	8Eh	01h	Clock Control register (Stretch=1)
P1	90h	FFh	Port 1
XP	9Fh	00h	External Data memory Paging register
P2	A0h	00h	Port 2
IEN0	A8h	00h	Interrupt Enable register 0
IP0	A9h	00h	Interrupt Priority register 0
P3	B0h	FFh	Port 3
IEN1	B8h	00h	Interrupt Enable register 1
IP1	B9h	00h	Interrupt Priority register 1
IRCON	C0h	00h	Interrupt Request Control register
PSW	D0h	00h	Program Status Word register
ACC	E0h	00h	Accumulator
B	F0h	00h	B register
WBT0[4]	F8h	00h	Wishbone Timing register 0
WBT1[4]	F9h	00h	Wishbone Timing register 1

Accumulator (ACC)

Most instructions use the Accumulator to hold the operand. Note that the mnemonics for Accumulator-specific instructions refer to the Accumulator as A, not ACC.

[4] Wishbone-compliant versions only (TSK52B_W, TSK52B_WD).

B Register

The B register is used during multiply and divide instructions. It can also be used as a scratch-pad register to hold temporary data.

External Data memory Paging register (XP)

The content of the XP register is loaded onto the high order byte of the memory address bus during external Data memory access using the MOVX @Ri, A and MOVX A, @Ri instructions. The XP register is used to implement paging and can provide access to up to 256 pages in external Data memory. Each page can contain up to 256 bytes of data – dependent on the contents of the register Ri. Therefore the maximum addressable external Data memory space is 64KB.

When the XP register is not used, its default reset value of 00h ensures the processor will act as its TSK51x counterpart for these two MOVX instructions, with the upper 8-bits of the memory address bus stuck at zeros.

Program Status Word register (PSW)

Table 7. PSW register flags

MSB LSB

CY	AC	F0	RS1	RS0	OV	F1	P

Table 8. PSW register bit functions

Bit	Symbol	Function
PSW.7	CY	Carry flag
PSW.6	AC	Auxiliary Carry flag for BCD operations
PSW.5	F0	General purpose Flag 0 available for user
PSW.4	RS1	Register bank select control bit 1, used to select working register bank
PSW.3	RS0	Register bank select control bit 0, used to select working register bank
PSW.2	OV	Overflow flag
PSW.1	F1	General purpose Flag 1 available for user
PSW.0	P	Parity flag

Bits RS1 and RS0 are used to select the working register bank as follows.

Table 9. Register Bank selection

RS1:RS0	Bank selected	Location
00	Bank 0	(00h – 07h)
01	Bank 1	(08h – 0Fh)
10	Bank 2	(10h – 17h)
11	Bank 3	(18h – 1Fh)

Stack Pointer (SP)

The Stack Pointer is a 1-byte register initialized to 07h after reset. This register is incremented before PUSH and CALL instructions, causing the stack to begin at location 08h.

Data Pointer register (DPL and DPH)

The Data Pointer (DPTR) is 2 bytes wide. The lower byte is DPL and the higher DPH. It can be loaded as either a single 2 byte register:

MOV DPTR,#data16)

or as two individual, single byte registers:

MOV DPL,#data8

MOV DPH,#data8.

It is generally used to access external code or data space, for example:

MOVC A,@A+DPTR or

MOVX A,@DPTR.

Program Counter (PC)

The Program Counter (PC) is 2 bytes wide and is initialized to 0000h after reset. This register is incremented during a fetch of operation code or operation data from Program memory.

Additional Wishbone Interface Special Function Registers

The Wishbone-compliant versions of the microcontroller – the TSK52B_W and TSK52B_WD respectively – contain two additional special function registers as part of the Wishbone Interface. These two timing registers – WBT0 and WBT1 – are used to provide a 14-bit value to the built-in Wishbone Cycle Counter, which determines how many clock cycles the processor will wait for an acknowledge signal from an addressed Wishbone slave device to appear at its ACK_I input, before the current data transfer cycle is forcibly terminated.

Wishbone Timing register 0 (WBT0)

Table 10. The WBT0 register

MSB							LSB
CNT5	CNT4	CNT3	CNT2	CNT1	CNT0	ACK	WCCEN

Table 11. The WBT0 register bit functions

Bit	Symbol	Function
WBT0.7	CNT5	Counter bit 5
WBT0.6	CNT4	Counter bit 4
WBT0.5	CNT3	Counter bit 3
WBT0.4	CNT2	Counter bit 2
WBT0.3	CNT1	Counter bit 1
WBT0.2	CNT0	Counter bit 0
WBT0.1	ACK	Acknowledge flag. Updated when the Wishbone Cycle Counter reaches zero, it is used to flag how the Wishbone transmission ended: 0 = Wishbone transfer cycle terminated normally, with an acknowledge signal received from the slave Wishbone device 1 = Wishbone transfer cycle has been forcibly terminated by the processor due to no acknowledgement from slave Wishbone device.
WBT0.0	WCCEN	Wishbone Cycle Counter Enable. 0 = Wishbone interface will wait until an acknowledge is received from an external Wishbone device 1 = Wishbone interface will wait for an acknowledge for CNT13-0 clock cycles, before forcibly terminating the transfer.

Wishbone Timing register 1 (WBT1)

Table 12. The WBT1 register

MSB LSB

CNT13	CNT12	CNT11	CNT10	CNT9	CNT8	CNT7	CNT6

Table 13. The WBT1 register bit functions

Bit	Symbol	Function
WBT1.7	CNT13	Counter bit 13
WBT1.6	CNT12	Counter bit 12
WBT1.5	CNT11	Counter bit 11
WBT1.4	CNT10	Counter bit 10
WBT1.3	CNT9	Counter bit 9
WBT1.2	CNT8	Counter bit 8
WBT1.1	CNT7	Counter bit 7
WBT1.0	CNT6	Counter bit 6

Note: Bits 7-0 of the WBT1 register and bits 7-2 of the WBT0 register are concatenated to form the 14-bit value, CNT13-0. This value is automatically re-loaded into the Wishbone Cycle Counter each time the processor initiates a Wishbone data transfer. The counter starts counting down automatically when a transfer is initiated and the initial value of the counter is greater than zero.

Hardware description

The structure of the TSK52x consists of:

- Control Processor Unit
- Arithmetic Logic Unit
- Clock Control Unit
- Memory Control Unit
- RAM and SFR Control Unit
- Ports Registers Unit
- Interrupt Service Routine Unit
- Wishbone Interface (TSK52B_W and TSK52B_WD only)

Block Diagram

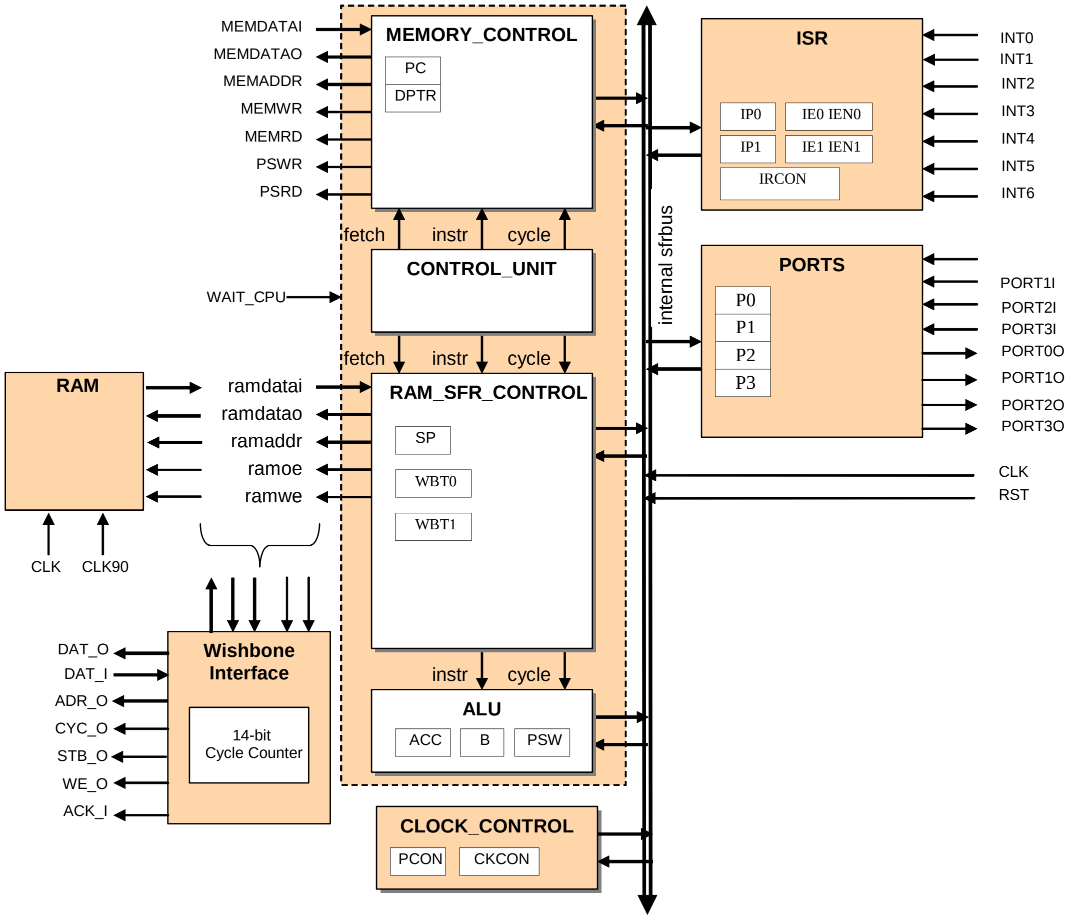


Figure 4. TSK52x Block Diagram

TSK52x Engine

The core engine of the TSK52x is composed of four components:

- Control Unit
- Arithmetic Logic Unit
- Memory Control Unit
- RAM and SFR Control Unit.

The TSK52x engine allows instructions to be fetched from Program memory and to execute using either RAM or SFR.

Ports

Ports P0, P1, P2 and P3 are Special Function Registers. The contents of the SFR can be observed on the corresponding component symbol interface pins. Writing a ‘1’ to any of the ports causes the corresponding pin to be at the high level and writing a ‘0’ causes the corresponding pin to be held at the low level.

All four ports on the chip are bi-directional. Each of them consists of a Latch (SFR P0 to P3), an output drive and an input buffer, so the CPU can output or read data through any of these ports if they are not used for alternate purposes.

Reset control

All registers and flip-flops are synchronously reset by the (active high) internal reset (rst) signal. An external hardware reset (RST) can activate the internal reset state. A high on the RST pin while the external clock is running, resets the device.

Interrupt Service Routine Unit

The TSK52x provides seven external interrupts with four priority levels. Each interrupt has its own request flag located in the special function register IRCON or IEN1. Each interrupt requested by the corresponding flag could individually be enabled or disabled by the enable bits in the special function register IEN0.

Interrupt overview

When the interrupt occurs, the engine will vector to a predetermined address (see Table 26). Once interrupt service has begun, it can be interrupted only by a higher priority interrupt. The interrupt service is terminated by a return from instruction RETI. When a RETI instruction is performed, the processor will return to the instruction that would have been next when the interrupt occurred.

When the interrupt condition occurs, the processor will also indicate this by setting a flag bit. This bit is set regardless of whether the interrupt is enabled or disabled. Each interrupt flag is sampled once per machine cycle, then samples are polled by hardware. If the sample indicates a pending interrupt when the interrupt is enabled, then the interrupt request flag is set. On the next multi-cycle instruction the interrupt will be acknowledged by hardware, forcing an LCALL to the appropriate vector address.

Interrupt response will require a varying amount of time depending on the state of the microcontroller when the interrupt occurs. If the microcontroller is performing an interrupt service with equal or greater priority, the new interrupt will not be invoked. In other cases, the response time depends on the current instruction. The fastest possible response to an interrupt is 7 machine cycles. This includes one machine cycle for detecting the interrupt and six cycles for performing the LCALL.

Interrupt-based Special Function Registers

Interrupt Enable register 0 (IEN0)

Table 14. The IEN0 register

MSB LSB

EAL	EX6	EX5	EX4	EX3	EX2	EX1	EX0

Table 15. The IEN0 register bit functions

Bit	Symbol	Function
IEN0.7	EAL	0 – disable all interrupts 1 – enable all interrupts
IEN0.6	EX6	0 – disable external interrupt 6 1 – enable external interrupt 6
IEN0.5	EX5	0 – disable external interrupt 5 1 – enable external interrupt 5
IEN0.4	EX4	0 – disable external interrupt 4 1 – enable external interrupt 4
IEN0.3	EX3	0 – disable external interrupt 3 1 – enable external interrupt 3
IEN0.2	EX2	0 – disable external interrupt 2 1 – enable external interrupt 2
IEN0.1	EX1	0 – disable external interrupt 1 1 – enable external interrupt 1
IEN0.0	EX0	0 – disable external interrupt 0 1 – enable external interrupt 0

Interrupt Enable register 1 (IEN1)

Table 16. The IEN1 register

MSB LSB

F3	F2	I3FR	I2FR	IE1	IT1	IE0	IT0

Table 17. The IEN1 register bit functions

Bit	Symbol	Function
IEN1.7	F3	General purpose Flag 3 available for user
IEN1.6	F2	General purpose Flag 2 available for user
IEN1.5	I3FR	External interrupt 3 falling/rising edge flag. 0 – external interrupt negative transition active 1 – external interrupt positive transition active
IEN1.4	I2FR	External interrupt 2 falling/rising edge flag. 0 – external interrupt negative transition active 1 – external interrupt positive transition active
IEN1.3	IE1	External interrupt 1 flag
IEN1.2	IT1	External interrupt 1 type control bit. 0 – external interrupt high level active 1 – external interrupt positive transition active
IEN1.1	IE0	External interrupt 0 flag
IEN1.0	IT0	External interrupt 0 type control bit. 0 – external interrupt high level active 1 – external interrupt positive transition active

Interrupt Request register (IRCON)

Table 18. The IRCON register

MSB							LSB
F6	F5	IEX2	IEX3	IEX4	IEX5	IEX6	F4

Table 19. The IRCON register bit functions

Bit	Symbol	Function
IRCON.7	F6	General purpose Flag 6 available for user
IRCON.6	F5	General purpose Flag 5 available for user
IRCON.5	IEX2	External interrupt 2 edge flag
IRCON.4	IEX3	External interrupt 3 edge flag
IRCON.3	IEX4	External interrupt 4 edge flag
IRCON.2	IEX5	External interrupt 5 edge flag

Bit	Symbol	Function
IRCON.1	IEX6	External interrupt 6 edge flag
IRCON.0	F4	General purpose Flag 4 available for user

Priority level structure

All interrupt sources have predefined priority level.

Table 20. Priority level

External interrupt 0
External interrupt 2
External interrupt 1
External interrupt 3
External interrupt 4
External interrupt 5
External interrupt 6

Each interrupt source can be programmed individually to one of four priority levels by setting or clearing the appropriate bit in the special function registers IP0 and IP1. If requests of the same priority level are received simultaneously, an internal polling sequence determines which request is serviced first.

Interrupt Priority register 0 (IP0)

Table 21. The IP0 register

MSB LSB

F7	IP0.6	IP0.5	IP0.4	IP0.3	IP0.2	IP0.1	IP0.0

Interrupt Priority register 1 (IP1)

Table 22. The IP1 register

MSB LSB

F8	IP1.6	IP1.5	IP1.4	IP1.3	IP1.2	IP1.1	IP1.0

Note: Bit 7 of register IP0 (F7) and bit 7 of register IP1 (F8) are general purpose flags available for the user.

Table 23. Priority levels

IP1.x	IP0.x	Priority Level
0	0	Level0 (lowest)
0	1	Level1
1	0	Level2
1	1	Level3 (highest)

Table 24. Priority level control bits

Bit	Interrupt Source
IP1.0, IP0.0	External interrupt 0
IP1.1, IP0.1	External interrupt 1
IP1.2, IP0.2	External interrupt 2
IP1.3, IP0.3	External interrupt 3
IP1.4, IP0.4	External interrupt 4
IP1.5, IP0.5	External interrupt 5
IP1.6, IP0.6	External interrupt 6

Table 25. Polling sequence

External interrupt 0	Polling sequence
External interrupt 2	
External interrupt 1	
External interrupt 3	
External interrupt 4	
External interrupt 5	
External interrupt 6	

Interrupt sources and vectors

Table 26. Interrupt vectors

Interrupt Request Flags	Interrupt Vector Address
IE0 – External interrupt 0	0003h
IE1 – External interrupt 1	0013h
IEX2 – External interrupt 2	004Bh

Interrupt Request Flags	Interrupt Vector Address
IEX3 – External interrupt 3	0053h
IEX4 – External interrupt 4	005Bh
IEX5 – External interrupt 5	0063h
IEX6 – External interrupt 6	006Bh

External interrupt edge detect

The external interrupts 2 and 3 can be programmed to be negative or positive transition-activated by setting or clearing bit I2FR or I3FR respectively, in register IEN1. The external interrupts 4, 5 and 6 are activated by a positive transition. The external source has to hold the request pin low (high for INT2 and INT3, if it is programmed to be negative transition-active) for at least one period of CLK. After this period, it must then be held high (low) for at least one period of CLK to ensure that the transition is recognized and that the corresponding interrupt request flag will be set.

Wishbone Interface (TSK52B_W and TSK52B_WD)

The same internal RAM interface signals are used to connect the Wishbone Interface to the RAM and SFR Unit. On the other side of this interface, standard Wishbone interface signals are used to connect the processor to any 8-bit compatible Wishbone slave device.

When accessing a Wishbone slave device through the Wishbone Interface, an 8-bit address is put on the ADR_O bus. Since a maximum of 108 addresses in SFR space can be used to address external Wishbone slave devices, bit 8 of ADR_O is always zero.

Writing to a Wishbone Slave Device

Data is written from the host microcontroller (Wishbone Master) to a Wishbone-compliant peripheral device (Wishbone Slave) in accordance with the standard Wishbone data transfer handshaking protocol. This data transfer cycle can be summarized as follows:

- The host presents an address on its ADR_O output for the register it wants to write to and a valid byte of data on its DAT_O output. It then asserts its WE_O output to specify a Write cycle
- The slave device receives the address at its ADR_I input and prepares to receive the data
- The host asserts its STB_O and CYC_O outputs, indicating that the transfer is to begin. The slave device, monitoring its STB_I and CYC_I inputs, reacts to this assertion by latching the byte of data appearing at its DAT_I input and asserting its ACK_O signal – to indicate to the host that the data has been received
- The host, monitoring its ACK_I input, responds by negating the STB_O and CYC_O signals. At the same time, the slave device negates the ACK_O signal and the data transfer cycle is naturally terminated.

Reading from a Wishbone Slave Device

Data is read by the host microcontroller (Wishbone Master) from a Wishbone-compliant peripheral device (Wishbone Slave) in accordance with the standard Wishbone data transfer handshaking protocol. This data transfer cycle can be summarized as follows:

- The host presents an address on its ADR_O output for the register it wishes to read. It then negates its WE_O output to specify a Read cycle
- The slave device receives the address at its ADR_I input and prepares to transmit the data from the selected register
- The host asserts its STB_O and CYC_O outputs, indicating that the transfer is to begin. The slave device, monitoring its STB_I and CYC_I inputs, reacts to this assertion by presenting the valid byte of data at its DAT_O output and asserting its ACK_O signal – to indicate to the host that valid data is present
- The host, monitoring its ACK_I input, responds by latching the byte of data appearing at its DAT_I input and negating the STB_O and CYC_O signals. At the same time, the slave device negates the ACK_O signal and the data transfer cycle is naturally terminated.

During Wishbone transmission the processor is stopped until an acknowledgement is received from a slave device. This can be a problem when a slave device disconnects from the Wishbone bus due to failure, leaving the processor waiting indefinitely for an acknowledge signal that will never come. To prevent this situation from happening, the Wishbone-compliant versions of the TSK52 have a built-in timer, that will automatically cancel a pending transmission after a given number of clock cycles. By default, this timer is inactive when the processor starts and the processor waits until there is an acknowledge from a slave device.

Communicating with Multiple Wishbone Slave Devices

Typically in a design, the microcontroller will need to interface to multiple Wishbone-compliant peripherals (slave devices). Each of these peripherals may contain any number of internal registers with which to write to/read from. It is not possible to communicate directly, and simultaneously, with each of these slave devices. A means of multiplexing must be used, allowing the microcontroller to talk to any number of slaves over the one interface. Typically, this involves the use of a Wishbone Decoder, as illustrated in the example image of Figure 5.

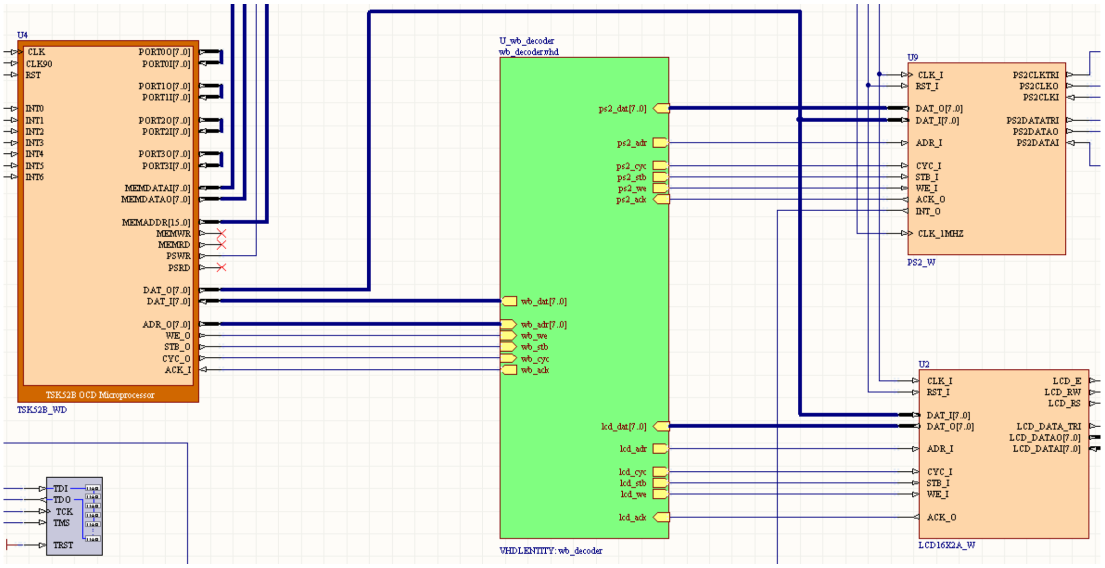
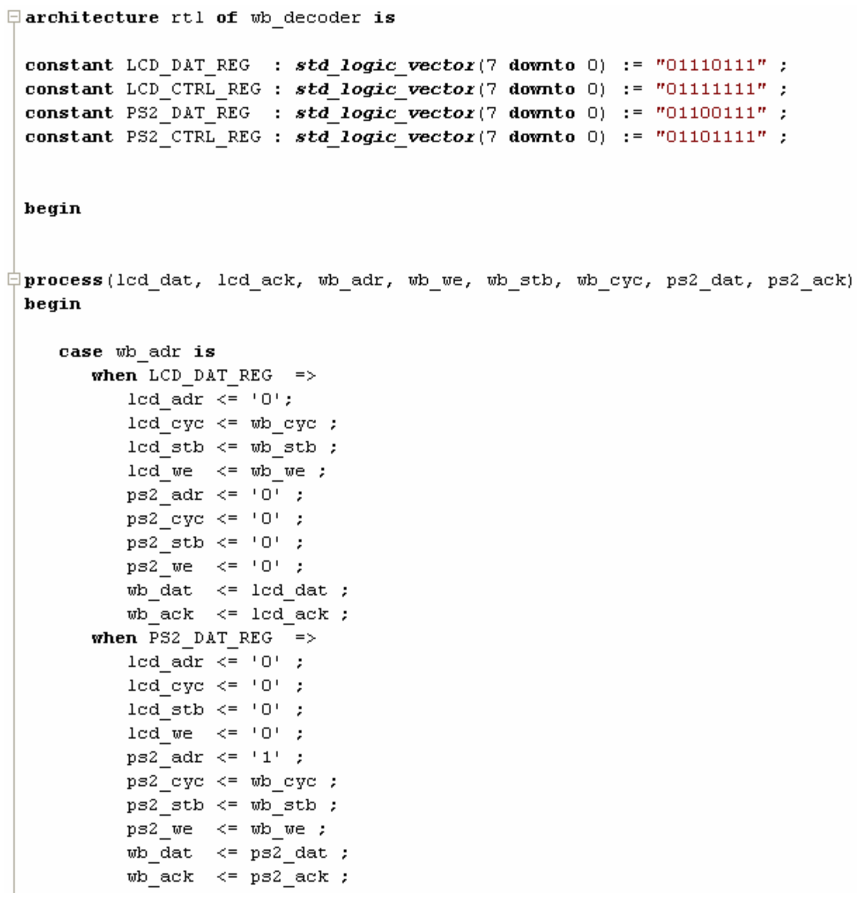


Figure 5. Multiplexing the Wishbone interface using a Wishbone Decoder

In the example circuit above, the Wishbone Decoder enables a single microcontroller device (TSK52B_WD) to communicate with two Wishbone-compliant peripheral devices (a PS/2 Controller and an LCD Controller). These two peripheral Controllers, in turn, each have two internal Wishbone registers that can be accessed by the microcontroller.

The Decoder itself is defined in an underlying VHDL file, which is used to decode the 8-bit address supplied by the microcontroller and enable communications with the relevant slave device and register therein, accordingly (Figure 6)



```
architecture rtl of wb_decoder is

constant LCD_DAT_REG  : std_logic_vector(7 downto 0) := "01110111" ;
constant LCD_CTRL_REG : std_logic_vector(7 downto 0) := "01111111" ;
constant PS2_DAT_REG  : std_logic_vector(7 downto 0) := "01100111" ;
constant PS2_CTRL_REG : std_logic_vector(7 downto 0) := "01101111" ;


begin


process(lcd_dat, lcd_ack, wb_adr, wb_we, wb_stb, wb_cyc, ps2_dat, ps2_ack)
begin

   case wb_adr is
      when LCD_DAT_REG  =>
         lcd_adr <= '0';
         lcd_cyc <= wb_cyc ;
         lcd_stb <= wb_stb ;
         lcd_we  <= wb_we ;
         ps2_adr <= '0' ;
         ps2_cyc <= '0' ;
         ps2_stb <= '0' ;
         ps2_we  <= '0' ;
         wb_dat  <= lcd_dat ;
         wb_ack  <= lcd_ack ;
      when PS2_DAT_REG  =>
         lcd_adr <= '0' ;
         lcd_cyc <= '0' ;
         lcd_stb <= '0' ;
         lcd_we  <= '0' ;
         ps2_adr <= '1' ;
         ps2_cyc <= wb_cyc ;
         ps2_stb <= wb_stb ;
         ps2_we  <= wb_we ;
         wb_dat  <= ps2_dat ;
         wb_ack  <= ps2_ack ;
```

Figure 6. Wishbone Decoder – under-the-bonnet code snippet

The exact configuration of a Wishbone Decoder and its underlying VHDL code, will vary depending on individual design requirements – the number of slave devices to be addressed, the number of accessible registers within each slave, etc – but the basic principle remains the same.

On-Chip Debugging

The debug-enabled versions of the microcontroller (TSK52A_D and TSK52B_WD) provide the following set of additional functional features that facilitate real-time debugging of the microcontroller:

- Reset, Go, Halt processor control
- Single or multi-step debugging
- Read-write access for internal processor registers including SFRs and PC
- Read-write access for Program memory and Data memory
- Unlimited software breakpoints

Adding debug functionality to a standard core variant

For the TSK52A_D and TSK52BW_D (henceforth referred to as TSK52xD) debug functionality is provided through the use of an On-Chip Debug System unit (OCDS). The simplified block diagram of Figure 7 shows the connection between this unit and the standard TSK52A core.

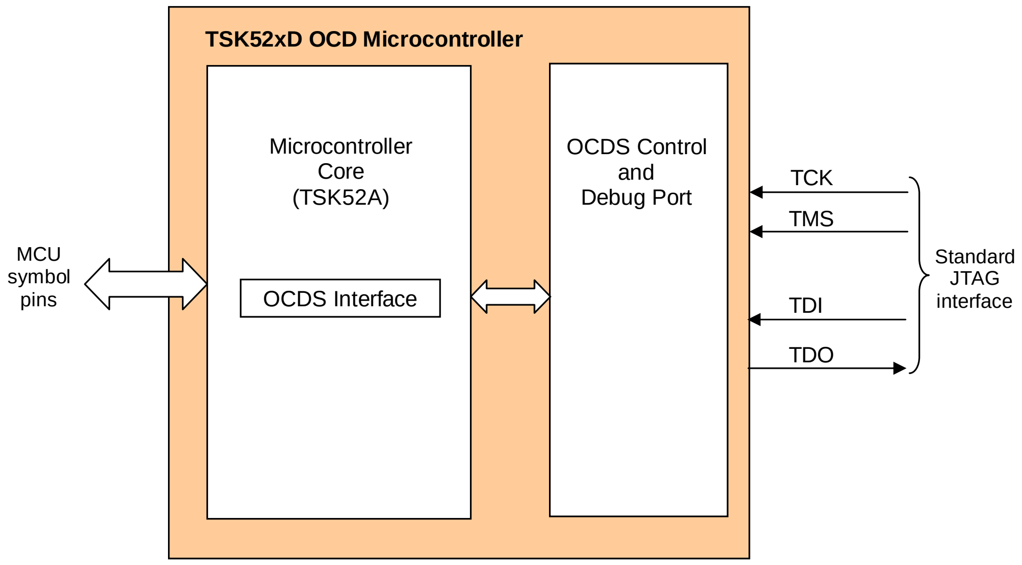

TSK52xD OCD Microcontroller
Microcontroller Core (TSK52A)
OCDS Interface
OCDS Control and Debug Port
MCU symbol pins
TCK
TMS
TDI
TDO
Standard JTAG interface

Figure 7. Simplified TSK52xD block diagram

The host computer is connected to the target core using the IEEE 1149.1 (JTAG) standard interface. This is the physical interface, providing connection to physical pins of the FPGA device in which the core has been embedded.

The Nexus 5001 standard is used as the protocol for communications between the host and all devices that are debug-enabled with respect to this protocol. This includes all OCD-version microcontrollers, as well as other Nexus-compliant devices such as frequency generators, logic analyzers, counters, etc.

All such devices are connected in a chain – the Soft Devices chain – which is determined when the design has been implemented within the target FPGA device and presents in the **Devices** view (Figure 8). It is not a physical chain, in the sense that you can see no external wiring – the connections required between the Nexus-enabled devices are made internal to the FPGA itself.

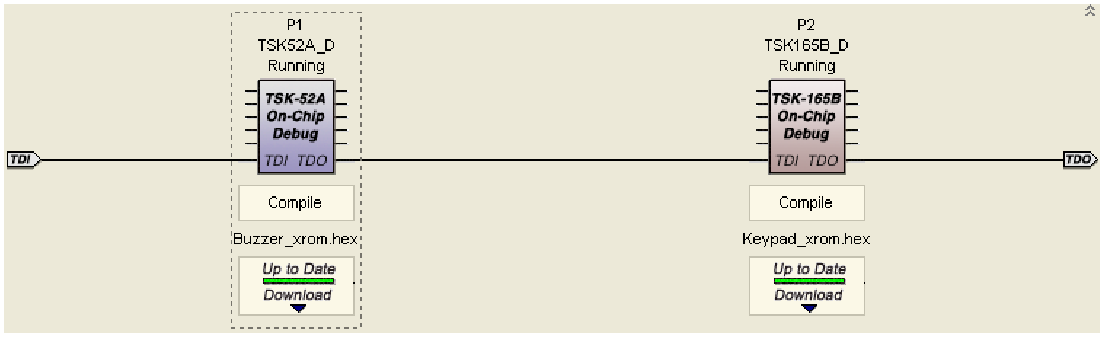


Figure 8. Nexus-enabled microcontrollers appearing in the Soft Devices chain

For microcontrollers such as the TSK52xD, the Nexus protocol enables you to debug the core through communication with the OCDS Unit.

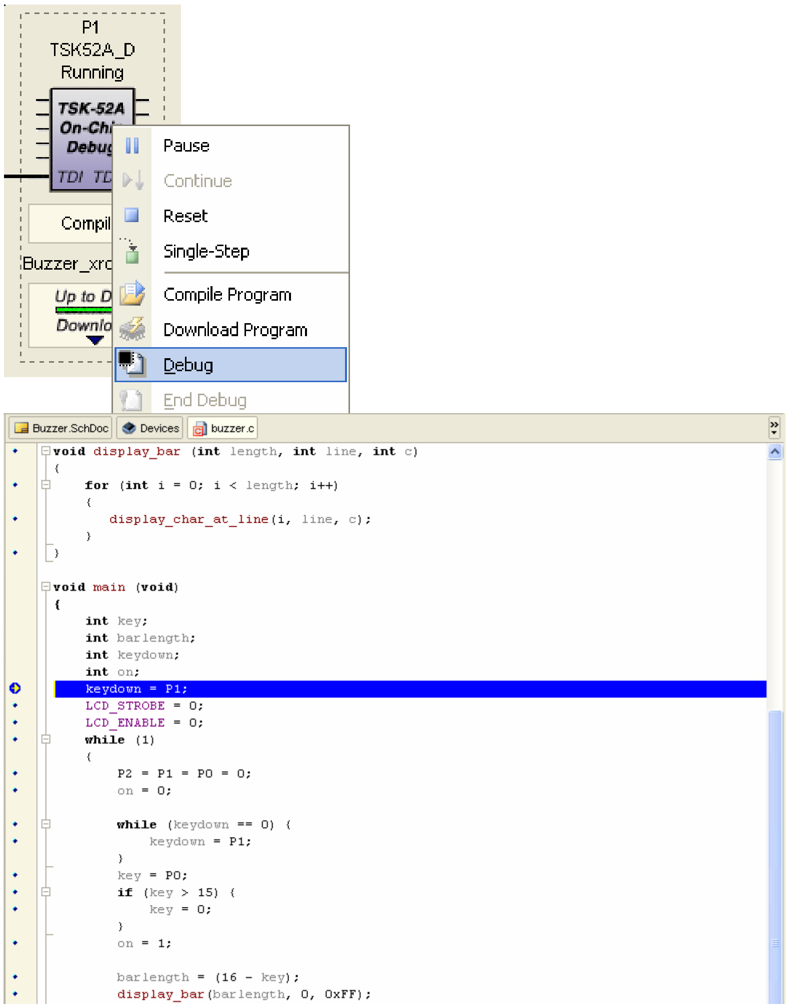
Accessing the debug environment

Debugging of the embedded code within an OCD-version microcontroller is carried out by starting a debug session. Prior to starting the session, you must ensure that the design, including one or more OCD-version microcontrollers and their respective embedded code, has been downloaded to the target physical FPGA device.

To start a debug session for the embedded code of a specific microcontroller in the design, simply right-click on the icon for that microcontroller, in the Soft Devices region of the view, and choose the **Debug** command from the pop-up menu that appears. Alternatively, click on the icon for the microcontroller (to focus it) and choose **Processors » Pn » Debug** from the main menus, where n corresponds to the number for the processor in the Soft Devices chain.

The embedded project for the software running in the processor will initially be recompiled and the debug session will commence. The relevant source code document (either Assembly or C) will be opened and the current execution point will be set to the first line of executable code (see Figure 9).

Note: You can have multiple debug sessions running simultaneously – one per embedded software project associated with a microcontroller in the Soft Devices chain.



```
void display_bar (int length, int line, int c)
{
    for (int i = 0; i < length; i++)
    {
        display_char_at_line(i, line, c);
    }
}

void main (void)
{
    int key;
    int barlength;
    int keydown;
    int on;
    keydown = P1;
    LCD_STROBE = 0;
    LCD_ENABLE = 0;
    while (1)
    {
        P2 = P1 = P0 = 0;
        on = 0;

        while (keydown == 0) {
            keydown = P1;
        }
        key = P0;
        if (key > 15) {
            key = 0;
        }
        on = 1;

        barlength = (16 - key);
        display_bar(barlength, 0, 0xFF);
```

Figure 9. Starting an embedded code debug session.

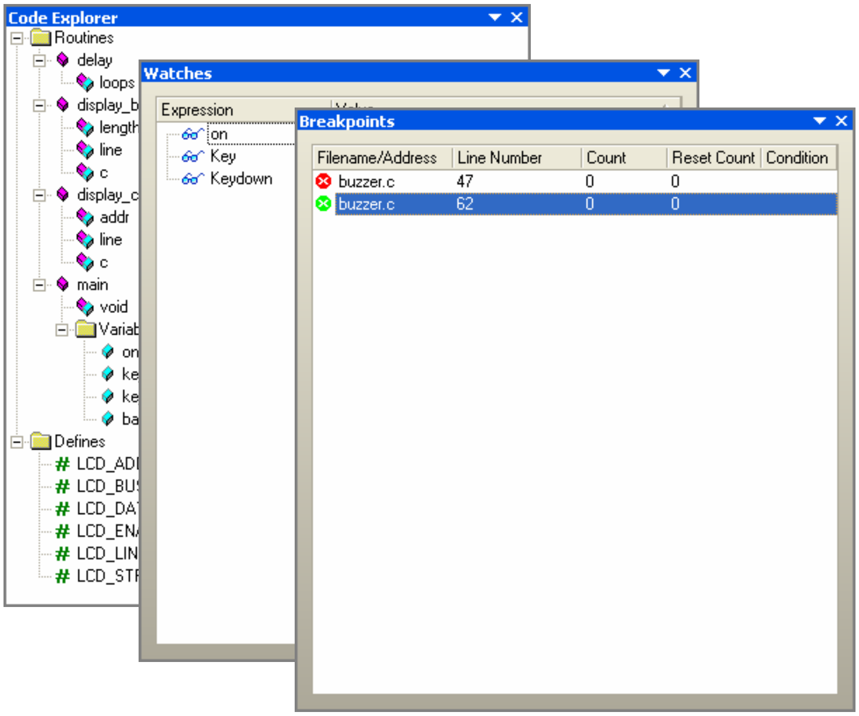
The debug environment offers the full suite of tools you would expect to see in order to efficiently debug the embedded code. These features include:

- Setting Breakpoints
- Adding Watches
- Stepping into and over at both the source (`*.C`) and instruction (`*.asm`) level
- Reset, Run and Halt code execution
- Run to cursor

All of these and other feature commands can be accessed from the **Debug** menu or the associated **Debug** toolbar.

Various workspace panels are accessible in the debug environment, allowing you to view/control code-specific features, such as Breakpoints, Watches and Local variables, as well as information specific to the microcontroller in which the code is running, such as memory spaces and registers.

These panels can be accessed from the **View » Workspace Panels » Embedded** sub menu, or by clicking on the **Embedded** button at the bottom of the application window and choosing the required panel from the subsequent pop-up menu.



Code Explorer

```
Routines
  delay
    loops
  display_b
    length
    line
    c
  display_c
    addr
    line
    c
  main
    void
    Variab
      on
      ke
      ke
      ba
Defines
  LCD_ADI
  LCD_BUS
  LCD_DAT
  LCD_ENA
  LCD_LIN
  LCD_STI
```

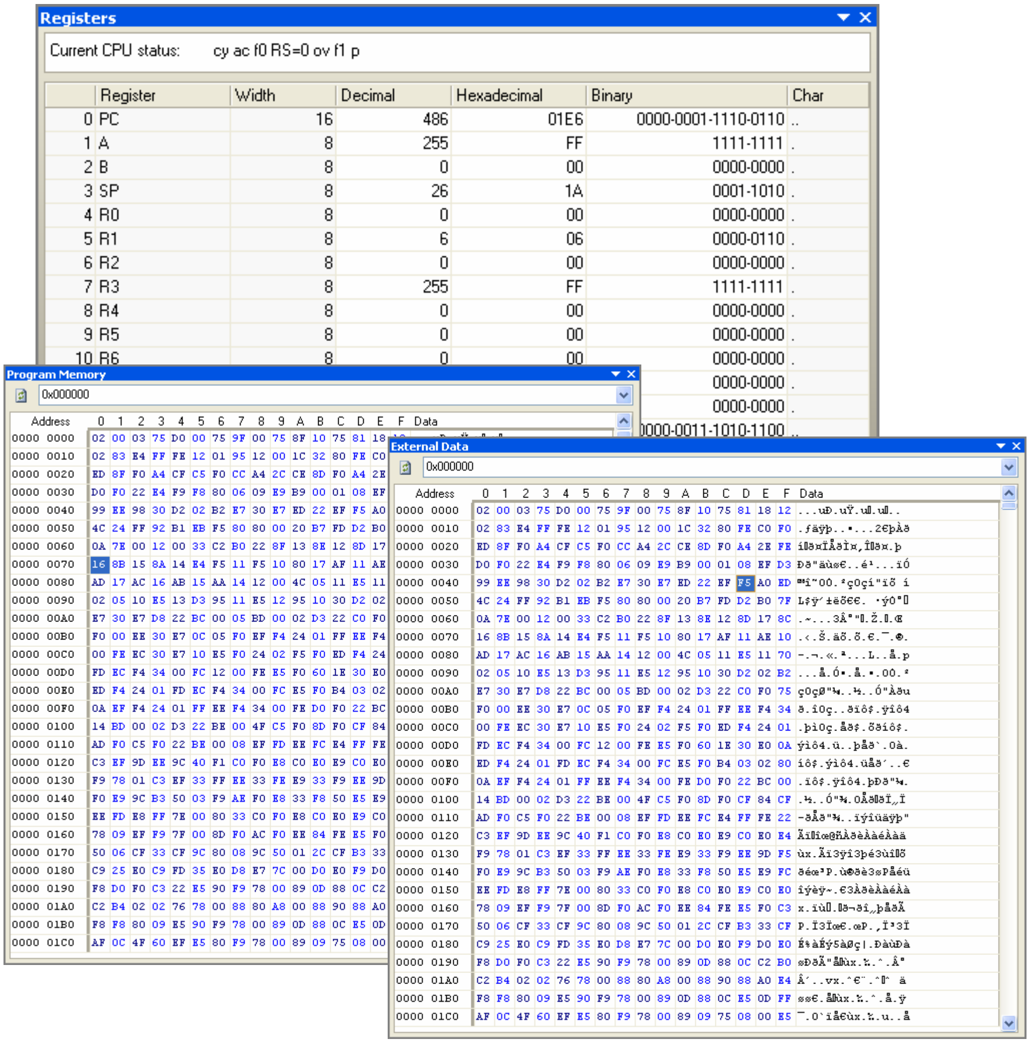
Watches

Expression
on
Key
Keydown

Breakpoints

Filename/Address	Line Number	Count	Reset Count	Condition
buzzer.c	47	0	0	
buzzer.c	62	0	0	

Figure 10. Workspace panels offering code-specific information and controls



```
Registers
Current CPU status:     cy ac f0 RS=0 ov f1 p

   Register  Width  Decimal  Hexadecimal  Binary                Char
 0 PC           16      486         01E6  0000-0001-1110-0110   ..
 1 A             8      255           FF            1111-1111   .
 2 B             8        0           00            0000-0000   .
 3 SP            8       26           1A            0001-1010   .
 4 R0            8        0           00            0000-0000   .
 5 R1            8        6           06            0000-0110   .
 6 R2            8        0           00            0000-0000   .
 7 R3            8      255           FF            1111-1111   .
 8 R4            8        0           00            0000-0000   .
 9 R5            8        0           00            0000-0000   .
10 R6            8        0           00            0000-0000   .
                                                    0000-0000   .
                                                    0000-0000   .
                                          0000-0011-1010-1100   ..
```

```
Program Memory
0x000000
Address     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  Data
0000 0000  02 00 03 75 D0 00 75 9F 00 75 8F 10 75 81 18
0000 0010  02 83 E4 FF FE 12 01 95 12 00 1C 32 80 FE C0
0000 0020  ED 8F F0 A4 CF C5 F0 CC A4 2C CE 8D F0 A4 2E
0000 0030  D0 F0 22 E4 F9 F8 80 06 09 E9 B9 00 01 08 EF
0000 0040  99 EE 98 30 D2 02 B2 E7 30 E7 ED 22 EF F5 A0
0000 0050  4C 24 FF 92 B1 EB F5 80 80 00 20 B7 FD D2 B0
0000 0060  0A 7E 00 12 00 33 C2 B0 22 8F 13 8E 12 8D 17
0000 0070  16 8B 15 8A 14 E4 F5 11 F5 10 80 17 AF 11 AE
0000 0080  AD 17 AC 16 AB 15 AA 14 12 00 4C 05 11 E5 11
0000 0090  02 05 10 E5 13 D3 95 11 E5 12 95 10 30 D2 02
0000 00A0  E7 30 E7 D8 22 BC 00 05 BD 00 02 D3 22 C0 F0
0000 00B0  F0 00 EE 30 E7 0C 05 F0 EF F4 24 01 FF EE F4
0000 00C0  00 FE EC 30 E7 10 E5 F0 24 02 F5 F0 ED F4 24
0000 00D0  FD EC F4 34 00 FC 12 00 FE E5 F0 60 1E 30 E0
0000 00E0  ED F4 24 01 FD EC F4 34 00 FC E5 F0 B4 03 02
0000 00F0  0A EF F4 24 01 FF EE F4 34 00 FE D0 F0 22 BC
0000 0100  14 BD 00 02 D3 22 BE 00 4F C5 F0 8D F0 CF 84
0000 0110  AD F0 C5 F0 22 BE 00 08 EF FD EE FC E4 FF FE
0000 0120  C3 EF 9D EE 9C 40 F1 C0 F0 E8 C0 E0 E9 C0 E0
0000 0130  F9 78 01 C3 EF 33 FF EE 33 FE E9 33 F9 EE 9D
0000 0140  F0 E9 9C B3 50 03 F9 AE F0 E8 33 F8 50 E5 E9
0000 0150  EE FD E8 FF 7E 00 80 33 C0 F0 E8 C0 E0 E9 C0
0000 0160  78 09 EF F9 7F 00 8D F0 AC F0 EE 84 FE E5 F0
0000 0170  50 06 CF 33 CF 9C 80 08 9C 50 01 2C CF B3 33
0000 0180  C9 25 E0 C9 FD 35 E0 D8 E7 7C 00 D0 E0 F9 D0
0000 0190  F8 D0 F0 C3 22 E5 90 F9 78 00 89 0D 88 0C C2
0000 01A0  C2 B4 02 02 76 78 00 88 80 A8 00 88 90 88 A0
0000 01B0  F8 F8 80 09 E5 90 F9 78 00 89 0D 88 0C E5 0D
0000 01C0  AF 0C 4F 60 EF E5 80 F9 78 00 89 09 75 08 00
```

```
External Data
0x000000
Address     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  Data
0000 0000  02 00 03 75 D0 00 75 9F 00 75 8F 10 75 81 18 12  ...uÐ.uŸ.u.u..
0000 0010  02 83 E4 FF FE 12 01 95 12 00 1C 32 80 FE C0 F0  .ƒäÿþ..•...2€þÀð
0000 0020  ED 8F F0 A4 CF C5 F0 CC A4 2C CE 8D F0 A4 2E FE  í.ð¤ÏÅðÌ¤,Î.ð¤.þ
0000 0030  D0 F0 22 E4 F9 F8 80 06 09 E9 B9 00 01 08 EF D3  Ðð"äùø€..é¹...ïÓ
0000 0040  99 EE 98 30 D2 02 B2 E7 30 E7 ED 22 EF F5 A0 ED  ™î˜0Ò.²ç0çí"ïõ í
0000 0050  4C 24 FF 92 B1 EB F5 80 80 00 20 B7 FD D2 B0 7F  L$ÿ'±ëõ€€. ·ýÒ°.
0000 0060  0A 7E 00 12 00 33 C2 B0 22 8F 13 8E 12 8D 17 8C  .~...3Â°".Ž...Œ
0000 0070  16 8B 15 8A 14 E4 F5 11 F5 10 80 17 AF 11 AE 10  .‹.Š.äõ.õ.€.¯.®.
0000 0080  AD 17 AC 16 AB 15 AA 14 12 00 4C 05 11 E5 11 70  -.¬.«.ª...L..å.p
0000 0090  02 05 10 E5 13 D3 95 11 E5 12 95 10 30 D2 02 B2  ...å.Ó•.å.•.0Ò.²
0000 00A0  E7 30 E7 D8 22 BC 00 05 BD 00 02 D3 22 C0 F0 75  ç0çØ"¼..½..Ó"Àðu
0000 00B0  F0 00 EE 30 E7 0C 05 F0 EF F4 24 01 FF EE F4 34  ð.î0ç..ðïô$.ÿîô4
0000 00C0  00 FE EC 30 E7 10 E5 F0 24 02 F5 F0 ED F4 24 01  .þì0ç.åð$.õðíô$.
0000 00D0  FD EC F4 34 00 FC 12 00 FE E5 F0 60 1E 30 E0 0A  ýìô4.ü..þåð`.0à.
0000 00E0  ED F4 24 01 FD EC F4 34 00 FC E5 F0 B4 03 02 80  íô$.ýìô4.üåð´..€
0000 00F0  0A EF F4 24 01 FF EE F4 34 00 FE D0 F0 22 BC 00  .ïô$.ÿîô4.þÐð"¼.
0000 0100  14 BD 00 02 D3 22 BE 00 4F C5 F0 8D F0 CF 84 CF  .½..Ó"¾.OÅð.ðÏ„Ï
0000 0110  AD F0 C5 F0 22 BE 00 08 EF FD EE FC E4 FF FE 22  -ðÅð"¾..ïýîüäÿþ"
0000 0120  C3 EF 9D EE 9C 40 F1 C0 F0 E8 C0 E0 E9 C0 E0 E4  Ãï.îœ@ñÀðèÀàéÀàä
0000 0130  F9 78 01 C3 EF 33 FF EE 33 FE E9 33 F9 EE 9D F5  ùx.Ãï3ÿî3þé3ùî.õ
0000 0140  F0 E9 9C B3 50 03 F9 AE F0 E8 33 F8 50 E5 E9 FC  ðéœ³P.ù®ðè3øPåéü
0000 0150  EE FD E8 FF 7E 00 80 33 C0 F0 E8 C0 E0 E9 C0 E0  îýèÿ~.€3ÀðèÀàéÀà
0000 0160  78 09 EF F9 7F 00 8D F0 AC F0 EE 84 FE E5 F0 C3  x.ïù..ð¬ðî„þåðÃ
0000 0170  50 06 CF 33 CF 9C 80 08 9C 50 01 2C CF B3 33 CF  P.Ï3Ïœ€.œP.,Ï³3Ï
0000 0180  C9 25 E0 C9 FD 35 E0 D8 E7 7C 00 D0 E0 F9 D0 E0  É%àÉý5àØç|.ÐàùÐà
0000 0190  F8 D0 F0 C3 22 E5 90 F9 78 00 89 0D 88 0C C2 B0  øÐðÃ"å.ùx.‰.ˆ.Â°
0000 01A0  C2 B4 02 02 76 78 00 88 80 A8 00 88 90 88 A0 E4  Â´..vx.ˆ€¨.ˆ.ˆ ä
0000 01B0  F8 F8 80 09 E5 90 F9 78 00 89 0D 88 0C E5 0D FF  øø€.å.ùx.‰.ˆ.å.ÿ
0000 01C0  AF 0C 4F 60 EF E5 80 F9 78 00 89 09 75 08 00 E5  ¯.O`ïå€ùx.‰.u..å
```

Figure 11. Workspace panels offering information specific to the parent processor.

Full-feature debugging is of course enjoyed at the source code level – from within the source code file itself. To a lesser extent, debugging can also be carried out from a dedicated debug panel for the processor. To access[5] this panel, first double-click on the icon representing the microcontroller to be debugged, in the **Soft Devices** region of the view. The *Instrument Rack – Soft Devices* panel will appear, with the chosen processor instrument added to the rack (Figure 12).

[5] The debug panels for each of the debug-enabled microcontrollers are standard panels and, as such, can be readily accessed from the **View » Workspace Panels » Instruments** sub-menu, or by clicking on the **Instruments** button at the bottom of the application window and choosing the required panel – for the processor you wish to debug – from the subsequent pop-up menu.

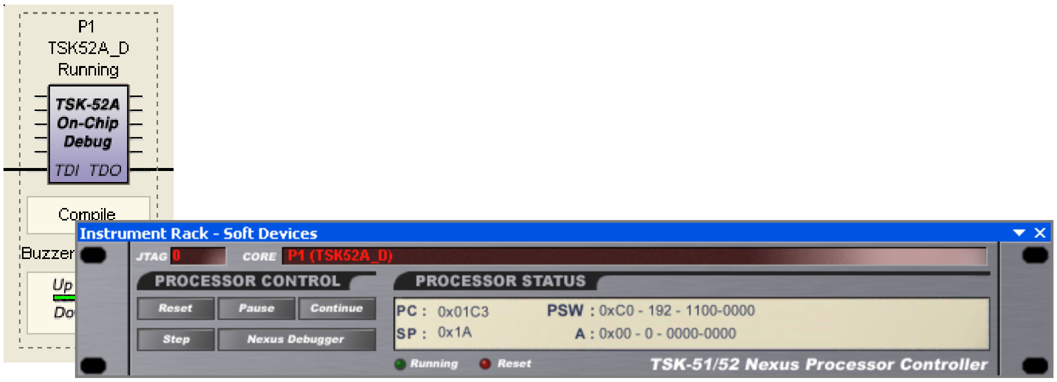


Figure 12. Accessing debug features from the microcontroller's instrument panel

Note: Each core microcontroller that you have included in the design will appear, when double-clicked, as an Instrument in the rack (along with any other Nexus-enabled devices).

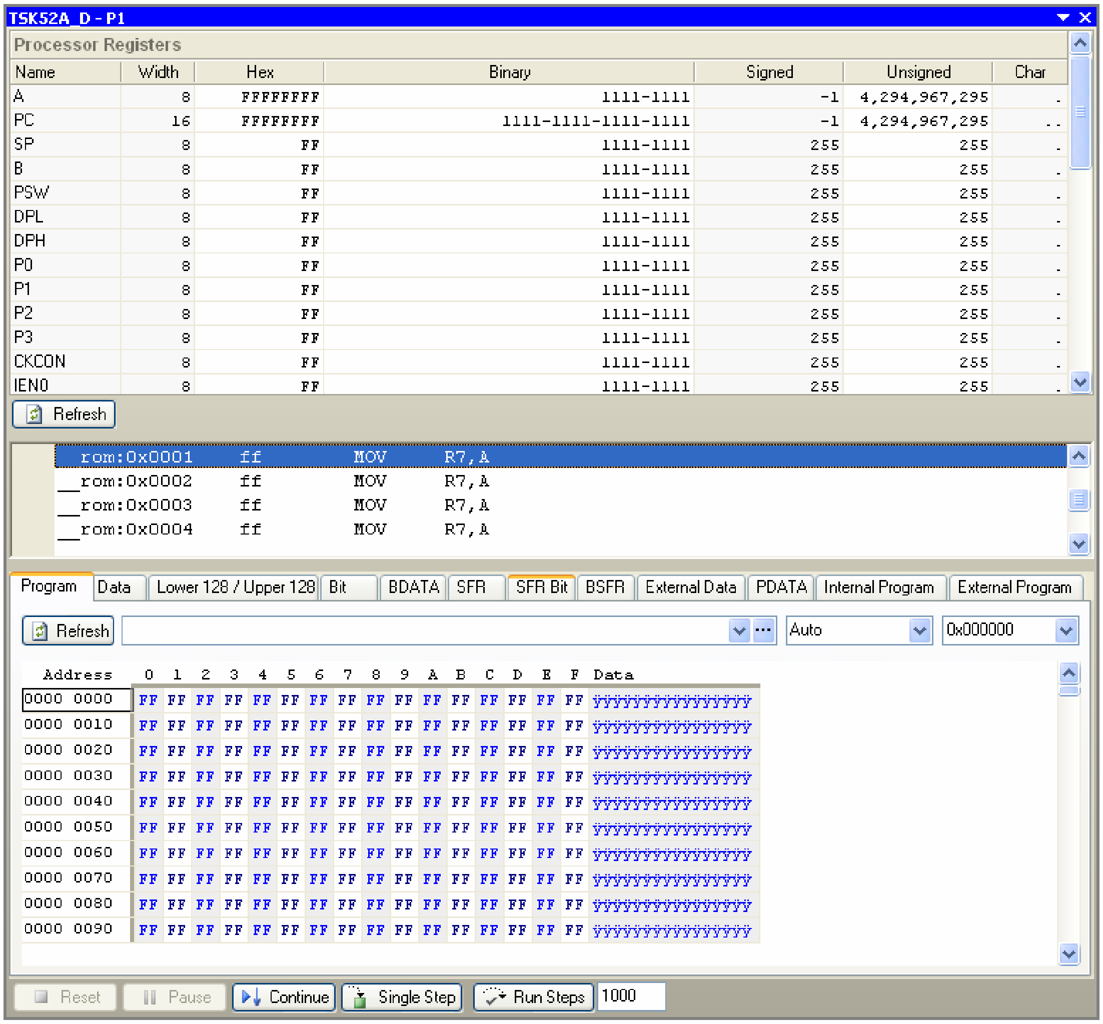
The **Nexus Debugger** button provides access to the associated debug panel (Figure 13), which in turn allows you to interrogate and to a lighter extent control, debugging of the processor and its embedded code, notably with respect to the microcontroller registers and memory.

One key feature of the debug panel is that it enables you to specify (and therefore change) the embedded code (HEX file) that is downloaded to the microcontroller, quickly and efficiently.

For more information on the content and use of processor debug panels, press **F1** when the cursor is over one of these panels.

For further information regarding the use of the embedded tools for the TSK52x, see the *Using the TSK51x/TSK52x Embedded Tools* guide.

For comprehensive information with respect to the embedded tools available for the TSK52x, see the *TSK51x/TSK52x Embedded Tools Reference*.



Name	Width	Hex	Binary	Signed	Unsigned	Char
A	8	FFFFFFFF	1111-1111	-1	4,294,967,295	.
PC	16	FFFFFFFF	1111-1111-1111-1111	-1	4,294,967,295	..
SP	8	FF	1111-1111	255	255	.
B	8	FF	1111-1111	255	255	.
PSW	8	FF	1111-1111	255	255	.
DPL	8	FF	1111-1111	255	255	.
DPH	8	FF	1111-1111	255	255	.
P0	8	FF	1111-1111	255	255	.
P1	8	FF	1111-1111	255	255	.
P2	8	FF	1111-1111	255	255	.
P3	8	FF	1111-1111	255	255	.
CKCON	8	FF	1111-1111	255	255	.
IEN0	8	FF	1111-1111	255	255	.

```
   rom:0x0001    ff       MOV       R7,A
__rom:0x0002    ff       MOV       R7,A
__rom:0x0003    ff       MOV       R7,A
__rom:0x0004    ff       MOV       R7,A
```

```
 Address    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  Data
0000 0000  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000 0010  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000 0020  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000 0030  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000 0040  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000 0050  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000 0060  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000 0070  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000 0080  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0000 0090  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

Figure 13. Processor debugging using an associated processor debug panel.

Instruction set

All TSK52x instructions are binary code compatible with the TSK51 processor.

Table 27. Notes on data addressing modes

Rn	Working register R0-R7
direct	256 internal RAM locations, any Special Function Registers
@Ri	Indirect internal or external RAM location addressed by register R0 or R1
#data	8-bit constant included in instruction
#data16	16-bit constant included as bytes 2 and 3 of instruction
bit	any bit-addressable I/O pin, control or status bit
A	Accumulator

Table 28. Notes on program addressing modes

addr16	Destination address for LCALL and LJMP may be anywhere within the 64KB of Program memory address space.
addr11	Destination address for ACALL and AJMP will be within the same 2KB page of Program memory as the first byte of the following instruction.
rel	SJMP and all conditional jumps include an 8-bit offset byte. Range is +127/-128 bytes relative to the first byte of the following instruction

Instruction set – functional groupings

Table 29. Arithmetic operations

Mnemonic	**Description**	**Hex Opcode**	**Width (in bytes)**	**No. of Instruction Cycles for execution**
ADD A,#data	Add immediate data to Accumulator	24	2	2
ADD A,@Ri	Add indirect RAM to Accumulator	26-27	1	2
ADD A,direct	Add direct byte to Accumulator	25	2	2
ADD A,Rn	Add register to Accumulator	28-2F	1	1
ADDC A,#data	Add immediate data to Accumulator with carry flag	34	2	2
ADDC A,@Ri	Add indirect RAM to Accumulator with carry flag	36-37	1	2
ADDC A,direct	Add direct byte to Accumulator with carry flag	35	2	2

Mnemonic	Description	Hex Opcode	Width (in bytes)	No. of Instruction Cycles for execution
ADDC A,Rn	Add register to Accumulator with carry flag	38-3F	1	1
DEC @Ri	Decrement indirect RAM	16-17	1	3
DEC A	Decrement Accumulator	14	1	1
DEC direct	Decrement direct byte	15	2	3
DEC Rn	Decrement register	18-1F	1	2
DIV AB	Divide A by B	84	1	6
INC @Ri	Increment indirect RAM	06-07	1	3
INC A	Increment Accumulator	04	1	1
INC direct	Increment direct byte	05	2	3
INC DPTR	Increment data pointer	A3	1	1
INC Rn	Increment register	08-0F	1	2
MUL AB	Multiply A and B	A4	1	2
SUBB A,#data	Subtract immediate data from Accumulator with borrow	94	2	2
SUBB A,@Ri	Subtract indirect RAM from Accumulator with borrow	96-97	1	2
SUBB A,direct	Subtract direct byte from Accumulator with borrow	95	2	2
SUBB A,Rn	Subtract register from Accumulator with borrow	98-9F	1	1
DA A	Decimal adjust Accumulator	D4	1	1

Table 30. Logic operations

Mnemonic	Description	Hex Opcode	Width (in bytes)	No. of Instruction Cycles for execution
ANL A,#data	AND immediate data to Accumulator	54	2	2
ANL A,@Ri	AND indirect RAM to Accumulator	56-57	1	2
ANL A,direct	AND direct byte to Accumulator	55	2	2
ANL A,Rn	AND register to Accumulator	58-5F	1	1
ANL direct,#data	AND immediate data to direct byte	53	3	3
ANL direct,A	AND Accumulator to direct byte	52	2	3
CLR A	Clear Accumulator	E4	1	1
CPL A	Complement Accumulator	F4	1	1
ORL A,#data	OR immediate data to Accumulator	44	2	2
ORL A,@Ri	OR indirect RAM to Accumulator	46-47	1	2
ORL A,direct	OR direct byte to Accumulator	45	2	2
ORL A,Rn	OR register to Accumulator	48-4F	1	1
ORL direct,#data	OR immediate data to direct byte	43	3	3
ORL direct,A	OR A to direct byte	42	2	3
RL A	Rotate Accumulator left	23	1	1
RLC A	Rotate Accumulator left through carry	33	1	1
RR A	Rotate Accumulator right	03	1	1
RRC A	Rotate Accumulator right through carry	13	1	1
SWAP A	Swap nibbles within Accumulator	C4	1	1
XRL A,#data	Exclusive OR immediate data to Accumulator	64	2	2
XRL A,@Ri	Exclusive OR indirect RAM to Accumulator	66-67	1	2
XRL A,direct	Exclusive OR direct byte to Accumulator	65	2	2
XRL A,Rn	Exclusive OR register to Accumulator	68-6F	1	1
XRL direct,#data	Exclusive OR immediate data to direct byte	63	3	3

Mnemonic	Description	Hex Opcode	Width (in bytes)	No. of Instruction Cycles for execution
XRL direct,A	Exclusive OR Accumulator to direct byte	62	2	3

Table 31. Data transfer

Mnemonic	Description	Hex Opcode	Width (in bytes)	No. of Instruction Cycles for execution
MOV @Ri,#data	Move immediate data to indirect RAM	76-77	2	3
MOV @Ri,A	Move Accumulator to indirect RAM	F6-F7	1	3
MOV @Ri,direct	Move direct byte to indirect RAM	A6-A7	2	5
MOV A,#data	Move immediate data to Accumulator	74	2	2
MOV A,@Ri	Move indirect RAM to Accumulator	E6-E7	1	2
MOV A,direct	Move direct byte to Accumulator	E5	2	2
MOV A,Rn	Move register to Accumulator	E8-EF	1	1
MOV direct,#data	Move immediate data to direct byte	75	3	3
MOV direct,@Ri	Move indirect RAM to direct byte	86-87	2	4
MOV direct,A	Move Accumulator to direct byte	F5	2	3
MOV direct,Rn	Move register to direct byte	88-8F	2	3
MOV direct1,direct2	Move direct byte to direct byte	85	3	4
MOV DPTR,#data16	Load data pointer with a 16-bit constant	90	3	3
MOV Rn,#data	Move immediate data to register	78-7F	2	2
MOV Rn,A	Move Accumulator to register	F8-FF	1	2
MOV Rn,direct	Move direct byte to register	A8-AF	2	4
MOVC A,@A+DPTR	Move code byte relative to DPTR to Accumulator	93	1	3
MOVC A,@A+PC	Move code byte relative to PC to Accumulator	83	1	3

Mnemonic	Description	Hex Opcode	Width (in bytes)	No. of Instruction Cycles for execution
MOVX @DPTR,A	Move Accumulator to external RAM (16-bit addr.)	F0	1	4-11
MOVX @Ri,A	Move Accumulator to external RAM (8-bit addr.)	F2-F3	1	4-11
MOVX A,@DPTR	Move external RAM (16-bit addr.) to Accumulator	E0	1	3-10
MOVX A,@Ri	Move external RAM (8-bit addr.) to Accumulator	E2-E3	1	3-10
POP direct	Pop direct byte from stack	D0	2	3
PUSH direct	Push direct byte onto stack	C0	2	4
XCH A,@Ri	Exchange indirect RAM with Accumulator	C6-C7	1	3
XCH A,direct	Exchange direct byte with Accumulator	C5	2	3
XCH A,Rn	Exchange register with Accumulator	C8-CF	1	2
XCHD A,@Ri	Exchange low-order nibble of indirect RAM with Accumulator	D6-D7	1	3

Table 32. Program branches

Mnemonic	Description	Hex Opcode	Width (in bytes)	No. of Instruction Cycles for execution
ACALL addr11	Absolute subroutine call	D1	2	6
AJMP addr11	Absolute jump	E1	2	3
CJNE @Ri,#data,rel	Compare immed. to ind. and jump if not equal	B6-B7	3	4
CJNE A,#data, rel	Compare immediate to Accumulator and jump if not equal	B4	3	4
CJNE A,direct,rel	Compare direct byte to Accumulator and jump if not equal	B5	3	4
CJNE Rn,#data rel	Compare immed. to reg. and jump if not equal	B8-BF	3	4

Mnemonic	Description	Hex Opcode	Width (in bytes)	No. of Instruction Cycles for execution
DJNZ direct	Decrement direct byte and jump if not zero	D5	3	4
DJNZ Rn	Decrement register and jump if not zero	D8-DF	2	3
JB bit,rel	Jump if direct bit is set	20	3	4
JBC bit,rel	Jump if direct bit is set and clear bit	10	3	4
JC rel	Jump if carry flag is set	40	2	3
JMP @A+DPTR	Jump indirect relative to the DPTR	73	1	2
JNB bit,rel	Jump if direct bit is not set	30	3	4
JNC bit,rel	Jump if carry flag is not set	50	2	3
JNZ rel	Jump if Accumulator is not zero	70	2	3
JZ rel	Jump if Accumulator is zero	60	2	3
LCALL addr16	Long subroutine call	12	3	6
LJMP addr16	Long jump	02	3	4
NOP	No operation	00	1	1
RET	From subroutine	22	1	4
RETI	From interrupt	32	1	4
SJMP rel	Short jump (relative addr.)	80	2	3

Table 33. Boolean manipulation

Mnemonic	Description	Hex Opcode	Width (in bytes)	No. of Instruction Cycles for execution
ANL C,/bit	AND complement of direct bit to carry flag	B0	2	3
ANL C,bit	AND direct bit to carry flag	82	2	3
CLR bit	Clear direct bit	C2	2	3
CLR C	Clear carry flag	C3	1	1
CPL bit	Complement direct bit	B2	2	3
CPL C	Complement carry flag	B3	1	2

MOV bit,C	Move carry flag to direct bit	92	2	3
MOV C,bit	Move direct bit to carry flag	A2	2	3
ORL C,/bit	OR complement of direct bit to carry flag	A0	2	3
ORL C,bit	OR direct bit to carry flag	72	2	3
SETB bit	Set direct bit	D2	2	3
SETB C	Set carry flag	D3	1	1

Hexadecimal ordered instructions

Table 34. Instruction Set in hexadecimal order

Opcode	Mnemonic	Opcode	Mnemonic
00h	NOP	10 H	JBC bit,rel
01h	AJMP addr11	11 H	ACALL addr11
02h	LJMP addr16	12 H	LCALL addr16
03h	RR A	13h	RRC A
04h	INC A	14h	DEC A
05h	INC direct	15h	DEC direct
06h	INC @R0	16h	DEC @R0
07h	INC @R1	17h	DEC @R1
08h	INC R0	18h	DEC R0
09h	INC R1	19h	DEC R1
0Ah	INC R2	1Ah	DEC R2
0Bh	INC R3	1Bh	DEC R3
0Ch	INC R4	1Ch	DEC R4
0Dh	INC R5	1Dh	DEC R5
0Eh	INC R6	1Eh	DEC R6
0Fh	INC R7	1Fh	DEC R7
20h	JB bit.rel	30h	JNB bit.rel
21h	AJMP addr11	31h	ACALL addr11
22h	RET	32h	RETI

Opcode	Mnemonic	Opcode	Mnemonic
23h	RL A	33h	RLC A
24h	ADD A,#data	34h	ADDC A,#data
25h	ADD A,direct	35h	ADDC A,direct
26h	ADD A,@R0	36h	ADDC A,@R0
27h	ADD A,@R1	37h	ADDC A,@R1
28h	ADD A,R0	38h	ADDC A,R0
29h	ADD A,R1	39h	ADDC A,R1
2Ah	ADD A,R2	3Ah	ADDC A,R2
2Bh	ADD A,R3	3Bh	ADDC A,R3
2Ch	ADD A,R4	3Ch	ADDC A,R4
2Dh	ADD A,R5	3Dh	ADDC A,R5
2Eh	ADD A,R6	3Eh	ADDC A,R6
2Fh	ADD A,R7	3Fh	ADDC A,R7
40h	JC rel	50h	JNC rel
41h	AJMP addr11	51h	ACALL addr11
42h	ORL direct,A	52h	ANL direct,A
43h	ORL direct,#data	53h	ANL direct,#data
44h	ORL A,#data	54h	ANL A,#data
45h	ORL A,direct	55h	ANL A,direct
46h	ORL A,@R0	56h	ANL A,@R0
47h	ORL A,@R1	57h	ANL A,@R1
48h	ORL A,R0	58h	ANL A,R0
49h	ORL A,R1	59h	ANL A,R1
4Ah	ORL A,R2	5Ah	ANL A,R2
4Bh	ORL A,R3	5Bh	ANL A,R3
4Ch	ORL A,R4	5Ch	ANL A,R4
4Dh	ORL A,R5	5Dh	ANL A,R5
4Eh	ORL A,R6	5Eh	ANL A,R6

Opcode	Mnemonic	Opcode	Mnemonic
4Fh	ORL A,R7	5Fh	ANL A,R7
60h	JZ rel	70h	JNZ rel
61h	AJMP addr11	71h	ACALL addr11
62h	XRL direct,A	72h	ORL C,bit
63h	XRL direct,#data	73h	JMP @A+DPTR
64h	XRL A,#data	74h	MOV A,#data
65h	XRL A,direct	75h	MOV direct,#data
66h	XRL A,@R0	76h	MOV @R0,#data
67h	XRL A,@R1	77h	MOV @R1,#data
68h	XRL A,R0	78h	MOV R0.#data
69h	XRL A,R1	79h	MOV R1.#data
6Ah	XRL A,R2	7Ah	MOV R2.#data
6Bh	XRL A,R3	7Bh	MOV R3.#data
6Ch	XRL A,R4	7Ch	MOV R4.#data
6Dh	XRL A,R5	7Dh	MOV R5.#data
6Eh	XRL A,R6	7Eh	MOV R6.#data
6Fh	XRL A,R7	7Fh	MOV R7.#data
80h	SJMP rel	90h	MOV DPTR,#data16
81h	AJMP addr11	91h	ACALL addr11
82h	ANL C,bit	92h	MOV bit,C
83h	MOVC A,@A+PC	93h	MOVC A,@A+DPTR
84h	DIV AB	94h	SUBB A,#data
85h	MOV direct,direct	95h	SUBB A,direct
86h	MOV direct,@R0	96h	SUBB A,@R0
87h	MOV direct,@R1	97h	SUBB A,@R1
88h	MOV direct,R0	98h	SUBB A,R0
89h	MOV direct,R1	99h	SUBB A,R1
8Ah	MOV direct,R2	9Ah	SUBB A,R2

Opcode	Mnemonic	Opcode	Mnemonic
8Bh	MOV direct,R3	9Bh	SUBB A,R3
8Ch	MOV direct,R4	9Ch	SUBB A,R4
8Dh	MOV direct,R5	9Dh	SUBB A,R5
8Eh	MOV direct,R6	9Eh	SUBB A,R6
8Fh	MOV direct,R7	9Fh	SUBB A,R7
A0h	ORL C,/bit	B0h	ANL C,/bit
A1h	AJMP addr11	B1h	ACALL addr11
A2h	MOV C,bit	B2h	CPL bit
A3h	INC DPTR	B3h	CPL C
A4h	MUL AB	B4h	CJNE A,#data,rel
A5h	-	B5h	CJNE A,direct,rel
A6h	MOV @R0,direct	B6h	CJNE @R0,#data,rel
A7h	MOV @R1,direct	B7h	CJNE @R1,#data,rel
A8h	MOV R0,direct	B8h	CJNE R0,#data,rel
A9h	MOV R1,direct	B9h	CJNE R1,#data,rel
AAh	MOV R2,direct	BAh	CJNE R2,#data,rel
ABh	MOV R3,direct	BBh	CJNE R3,#data,rel
ACh	MOV R4,direct	BCh	CJNE R4,#data,rel
ADh	MOV R5,direct	BDh	CJNE R5,#data,rel
AEh	MOV R6,direct	BEh	CJNE R6,#data,rel
AFh	MOV R7,direct	BFh	CJNE R7,#data,rel
C0h	PUSH direct	D0h	POP direct
C1h	AJMP addr11	D1h	ACALL addr11
C2h	CLR bit	D2h	SETB bit
C3h	CLR C	D3h	SETB C
C4h	SWAP A	D4h	DA A
C5h	XCH A,direct	D5h	DJNZ direct,rel
C6h	XCH A,@R0	D6h	XCHD A,@R0

Opcode	Mnemonic	Opcode	Mnemonic
C7h	XCH A,@R1	D7h	XCHD A,@R1
C8h	XCH A,R0	D8h	DJNZ R0,rel
C9h	XCH A,R1	D9h	DJNZ R1,rel
CAh	XCH A,R2	DAh	DJNZ R2,rel
CBh	XCH A,R3	DBh	DJNZ R3,rel
CCh	XCH A,R4	DCh	DJNZ R4,rel
CDh	XCH A,R5	DDh	DJNZ R5,rel
CEh	XCH A,R6	DEh	DJNZ R6,rel
CFh	XCH A,R7	DFh	DJNZ R7,rel
E0h	MOVX A,@DPTR	F0h	MOVX @DPTR,A
E1h	AJMP addr11	F1h	ACALL addr11
E2h	MOVX A,@R0	F2h	MOVX @R0,A
E3h	MOVX A,@R1	F3h	MOVX @R1,A
E4h	CLR A	F4h	CPL A
E5h	MOV A,direct	F5h	MOV direct,A
E6h	MOV A,@R0	F6h	MOV @R0,A
E7h	MOV A,@R1	F7h	MOV @R1,A
E8h	MOV A,R0	F8h	MOV R0,A
E9h	MOV A,R1	F9h	MOV R1,A
EAh	MOV A,R2	FAh	MOV R2,A
EBh	MOV A,R3	FBh	MOV R3,A
ECh	MOV A,R4	FCh	MOV R4,A
EDh	MOV A,R5	FDh	MOV R5,A
EEh	MOV A,R6	FEh	MOV R6,A
EFh	MOV A,R7	FFh	MOV R7,A

Instruction set – detailed reference

A brief example of how the instruction might be used is given as well as its effect on the PSW flags.

Only the carry, auxiliary carry, and overflow flags are discussed. The parity bit is always computed from the actual content of the accumulator.

Similarly, instructions, which alter directly addressed registers, could affect the other status flags if the instruction is applied to the PSW. Status flags can also be modified by bit manipulation.

In the following detailed instruction set listing, @Ri is an indirect internal or external RAM location addressed by register R0 or R1. When this operand is used, the encoding for the instruction contains an entry ‘I’. This will be replaced by a 0 or 1, depending on whether the register used is R0 or R1 respectively.

Similarly, the operand Rn can represent any of the eight working registers (R0-R7). The table below shows the registers that Rn can represent. The listed 3-bit value for each register replaces the rrr entry in the encoding for an instruction that uses this operand.

Register	**rrr**
R0	000
R1	001
R2	010
R3	011
R4	100
R5	101
R6	110
R7	111

ACALL addr11

Function:	Absolute call
Description:	ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the following instruction, then pushes the 16-bit result onto the stack (low-order byte first) and increments the stack pointer twice. The destination address is obtained by successively concatenating the five high-order bits of the incremented PC, op code bits 7-5, and the second byte of the instruction. The subroutine called must therefore start within the same 2K block of Program memory as the first byte of the instruction following ACALL. No flags are affected.

Operation: ACALL

(PC) ← (PC) + 2

(SP) ← (SP) + 1

((SP)) ← (PC7-0)

(SP) ← (SP) + 1

((SP)) ← (PC15-8)

(PC10-0) ← page address

Bytes: 2

Encoding:

a10	a9	a8	1	0	0	0	1	a7	a6	a5	a4	a3	a2	a1	a0

ADD A, <src-byte>

Function: Add

Description: ADD adds the byte variable indicated to the accumulator, leaving the result in the accumulator. The carry and auxiliary carry flags are set, respectively, if there is a carry out of bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred. OV is set if there is a carry out of bit 6 but not out of bit 7, or a carry out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands. Four source operand addressing modes are allowed: register, direct, register- indirect, or immediate.

ADD A, Rn

Operation: ADD

(A) ← (A) + (Rn)

Bytes: 1

Encoding:

0	0	1	0	1	r	r	r

ADD A, direct

Operation: ADD

(A) ← (A) + (direct)

Bytes: 2

Encoding:

0	0	1	0	0	1	0	1	direct address

ADD A, @Ri

Operation:	ADD
	(A) ← (A) + ((Ri))
Bytes:	1
Encoding:	

0	0	1	0	0	1	1	i

ADD A, #data

Operation:	ADD
	(A) ← (A) + #data
Bytes:	2
Encoding:	

0	0	1	0	0	1	0	0	immediate data

ADDC A, < src-byte>

Function:	Add with carry
Description:	ADDC simultaneously adds the byte variable indicated, the carry flag and the Accumulator contents, leaving the result in the Accumulator. The carry and auxiliary carry flags are set, respectively, if there is a carry out of bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred. OV is set if there is a carry out of bit 6 but not out of bit 7, or a carry out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands. Four source operand-addressing modes are allowed: register, direct, register- indirect, or immediate.

ADDC A, Rn

Operation:	ADDC
	(A) ← (A) + (C) + (Rn)
Bytes:	1
Encoding:	

0	0	1	1	1	r	r	r

ADDC A, direct

Operation:	ADDC
	(A) ← (A) + (C) + (direct)
Bytes:	2

Encoding:

0	0	1	1	0	1	0	1	direct address

ADDC A, @Ri

Operation: ADDC

$(A) \leftarrow (A) + (C) + ((Ri))$

Bytes: 1

Encoding:

0	0	1	1	0	1	1	i

ADDC A, #data

Operation: ADDC

$(A) \leftarrow (A) + (C) + \#data$

Bytes: 2

Encoding:

0	0	1	1	0	1	0	0	immediate data

AJMP addr11

Function: Absolute jump

Description: AJMP transfers program execution to the indicated address, which is formed at run-time by concatenating the high-order five bits of the PC (*after* incrementing the PC twice), op code bits 7-5, and the second byte of the instruction. The destination must therefore be within the same 2K block of Program memory as the first byte of the instruction following AJMP.

Operation: AJM P

$(PC) \leftarrow (PC) + 2$

$(PC10\text{-}0) \leftarrow$ page address

Bytes: 2

Encoding:

a10	a9	a8	0	0	0	0	1	a7	a6	a5	a4	a3	a2	a1	a0

ANL <dest-byte>, <src-byte>

Function: Logical AND for byte variables

Description: ANL performs the bit wise logical AND operation between the variables indicated and stores the result in the destination variable. No flags are affected (except P (Parity bit), if <dest-byte> = A). The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

ANL A,Rn

Operation:	ANL
	(A) ← (A) ^ (Rn)
Bytes:	1
Encoding:	

0	1	0	1	1	r	r	r

ANL A,direct

Operation:	ANL
	(A) ← (A) ^ (direct)
Bytes:	2
Encoding:	

0	1	0	1	0	1	0	1	direct address

ANL A, @Ri

Operation:	ANL
	(A) ← (A) ^ ((Ri))
Bytes:	1
Encoding:	

0	1	0	1	0	1	1	i

ANL A, #data

Operation:	ANL
	(A) ← (A) ^ #data
Bytes:	2
Encoding:	

0	1	0	1	0	1	0	0	immediate data

ANL direct,A

Operation:	ANL
	(direct) ← (direct) ^ (A)
Bytes:	2

Encoding:

0	1	0	1	0	0	1	0	direct address

ANL direct, #data

Operation: ANL

(direct) ← (direct) ^ #data

Bytes: 3

Encoding:

0	1	0	1	0	0	1	1
direct address							
immediate data							

ANL C, <src-bit>

Function: Logical AND for bit variables

Description: If the Boolean value of the source bit is a logic 0 then clear the carry flag; otherwise leave the carry flag in its current state. A slash (“/”) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, *but the source bit itself is not affected*. No other flags are affected. Only direct bit addressing is allowed for the source operand.

ANL C,bit

Operation: ANL

(C) ← (C) ^ (bit)

Bytes: 2

Encoding:

1	0	0	0	0	0	1	0	bit address

ANL C,/bit

Operation: ANL

(C) ← (C) ^ / (bit)

Bytes: 2

Encoding:

1	0	1	1	0	0	0	0	bit address

CJNE <dest-byte >, < src-byte >, rel

Function: Compare and jump if not equal

Description: CJNE compares the magnitudes of the first two operands, and branches if their values are not equal. The branch destination is computed by adding the signed relative displacement in the last instruction byte to the PC, after incrementing the PC to the start of the next instruction. The carry flag is set if the unsigned integer value of <dest-byte> is less than the unsigned integer value of <src-byte>; otherwise, the carry is cleared. Neither operand is affected. The first two operands allow four addressing mode combinations: the Accumulator may be compared with any directly addressed byte or immediate data, and any indirect RAM location or working register can be compared with an immediate constant.

CJNE A,direct,rel

Operation:

```
CJNE
(PC) ← (PC) + 3
if (A) < > (direct)
then (PC) ← (PC) + relative offset
if (A) < (direct)
then (C) ←1
else (C) ←0
```

Bytes: 3

Encoding:

1	0	1	1	0	1	0	1
direct address							
relative address							

CJNE A, #data,rel

Operation:

```
CJNE
(PC) ← (PC) + 3
if (A) < > data
then (PC) ← (PC) + relative offset
if (A) < data
then (C) ←1
else (C) ← 0
```

Bytes: 3

Encoding:

1	0	1	1	0	1	0	0
immediate data							
relative address							

CJNE RN, #data, rel

Operation:	CJNE
	(PC) ← (PC) + 3
	if (Rn) < > data
	then (PC) ← (PC) + relative offset
	if (Rn) < data
	then (C) ← 1
	else (C) ← 0
Bytes:	3
Encoding:	

1	0	1	1	1	r	r	r
immediate data							
relative address							

CJNE @Ri, #data, rel

Operation:	CJNE
	(PC) ← (PC) + 3
	if ((Ri)) < > data
	then (PC) ← (PC) + relative offset
	if ((Ri)) < data
	then (C) ← 1
	else (C) ← 0
Bytes:	3
Encoding:	

1	0	1	1	0	1	1	i
immediate data							
relative address							

CLR A

Function: Clear Accumulator

Description: The Accumulator is cleared (all bits set to zero). No flags are affected.

Operation: CLR

(A) ← 0

Bytes: 1

Encoding:

1	1	1	0	0	1	0	0

CLR bit

Function: Clear bit

Description: The indicated bit is cleared (reset to zero). No other flags are affected. CLR can operate on any directly addressable bit.

Operation: CLR

(bit) ← 0

Bytes: 2

Encoding:

1	1	0	0	0	0	1	0	bit address

CLR C

Function: Clear carry flag

Description: The carry flag is cleared (reset to zero). No other flags are affected.

Operation: CLR

(C) ← 0

Bytes: 1

Encoding:

1	1	0	0	0	0	1	1

CPL A

Function: Complement Accumulator

Description: Each bit of the Accumulator is logically complemented (one’s complement). Bits which previously contained a one are changed to zero and vice versa. No flags are affected.

Operation: CPL

(A) ← / (A)

Bytes: 1

Encoding:

1	1	1	1	0	1	0	0

CPL bit

Function: Complement bit

Description: The bit variable specified is complemented. A bit which had been a one is changed to zero and vice versa. No other flags are affected. CPL can operate on any directly addressable bit.

Note: When this instruction is used to modify an output pin, the value used as the original data will be read from the output data latch, not the input pin.

Operation: CPL

(bit) ← / (bit)

Bytes: 2

Encoding:

1	0	1	1	0	0	1	0	bit address

CPL C

Function: Complement carry flag

Description: The carry flag is complemented. If the flag had been a one it is changed to zero and vice versa. No other flags are affected.

Operation: CPL

(C) ← / (C)

Bytes: 1

Encoding:

1	0	1	1	0	0	1	1

DA A

Function: Decimal adjust accumulator for addition

Description: DA A adjusts the eight-bit value in the accumulator resulting from the earlier addition of two variables (each in packed BCD format), producing two four-bit digits. Any ADD or ADDC instruction may have been used to perform the addition. If accumulator bits 3-0 are greater than nine (xxxx1010-xxxx1111), of if the AC flag is one, six is added to the accumulator producing the proper BCD digit in the low- order nibble. This internal addition would set the carry flag if a carry-out of the low-order four-bit field propagated through all high-order bits, but it would not clear the carry flag otherwise.

If the carry flag is now set, or if the four high-order bits now exceed nine (1010xxxx-

1111xxxx), these high-order bits are inceremented by six, producing the proper BCD digit in the high-order nibble. Again, this would set the carry flag thus indicating that the sum of the original two BCD variables is greater than 100, allowing multiple precision decimal additions. OV is not affected.

All of this occurs during the one instruction cycle. Essentially, this instruction performs the decimal conversion by adding 00h, 06h, 60h or 66h to the accumulator, depending on initial accumulator and PSW conditions.

Note: DA A cannot simply convert a hexadecimal number in the accumulator to BCD notation, nor does DA A apply to decimal subtraction.

Operation: DA

Content of accumulator is BCD

if [[(A3-0) > 9] ^ [(AC) = 1]]

then (A3-0) ← (A3-0) + 6

and

if [[(A7-4) > 9] ^ [(C) = 1]]

then (A7-4) ← (A7-4) + 6

Bytes: 1

Encoding:

1	1	0	1	0	1	0	0

DEC byte

Function: Decrement

Description: The variable indicated is decremented by 1. An original value of 00h will underflow to 0FFh. No flags are affected. Four operand addressing modes are allowed: Accumulator, register, direct, or register-indirect.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

DEC A

Operation: DEC

(A) ← (A) - 1

Bytes: 1

Encoding:

0	0	0	1	0	1	0	0

DEC Rn

Operation: DEC

(Rn) ← (Rn) - 1

Bytes: 1

Encoding:

0	0	0	1	1	r	r	r

DEC direct

Operation: DEC

(direct) ← (direct) - 1

Bytes: 2

Encoding:

0	0	0	1	0	1	0	1	direct address

DEC @Ri

Operation: DEC

((Ri)) ← ((Ri)) - 1

Bytes: 1

Encoding:

0	0	0	1	0	1	1	i

DIV AB

Function: Divide

Description: DIV AB divides the unsigned eight-bit integer in the Accumulator by the unsigned eight-bit integer in register B. The Accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags will be cleared.

Exception: If B had originally contained 00 h, the values returned in the Accumulator and B register will be undefined and the overflow flag will be set. The carry flag is cleared in any case.

Operation: DIV

(A) ← 15-8

(A) / (B)

(B) ← 7-0

Bytes: 1

Encoding:

1	0	0	0	0	1	0	0

DJNZ <byte>, <rel-addr>

Function: Decrement and jump if not zero

Description: DJNZ decrements the location indicated by 1, and branches to the address indicated by the second operand if the resulting value is not zero. An original value of 00h will underflow to 0FFh. No flags are affected. The branch destination would be computed by adding the signed relative-displacement value in the last instruction byte to the PC, after incrementing the PC to the first byte of the following instruction. The location decremented may be a register or directly addressed byte.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

DJNZ Rn,rel

Operation: DJNZ

(PC) ← (PC) + 2

(Rn) ← (Rn) - 1

if (Rn) > 0 or (Rn) < 0

then (PC) ← (PC) + rel

Bytes: 2

Encoding:

1	1	0	1	1	r	r	r	relative address

DJNZ direct,rel

Operation: DJNZ

(PC) ← (PC) + 2

(direct) ← (direct) - 1

if (direct) > 0 or (direct) < 0

then (PC) ← (PC) + rel

Bytes: 3

Encoding:

1	1	0	1	0	1	0	1
direct address							
relative address							

INC <byte>

Function: Increment

Description: INC increments the indicated variable by 1. An original value of 0FFh will overflow to 00h. No flags are affected. Four operand addressing modes are allowed: Accumulator, register, direct, or register-indirect.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

INC A

Operation: INC

(A) ← (A) + 1

Bytes: 1

Encoding:

0	0	0	0	0	1	0	0

INC Rn

Operation: INC

(Rn) ← (Rn) + 1

Bytes: 1

Encoding:

0	0	0	0	1	r	r	r

INC direct

Operation: INC

(direct) ← (direct) + 1

Bytes: 2

Encoding:

0	0	0	0	0	1	0	1	direct address

INC @Ri

Operation: INC

((Ri)) ← ((Ri)) + 1

Bytes: 1

Encoding:

0	0	0	0	0	1	1	i

INC DPTR

Function: Increment data pointer

Description: Increment the 16-bit data pointer by 1. A 16-bit increment (modulo 2^{16}) is performed; an overflow of the low-order byte of the data pointer (DPL) from 0FFh to 00h will increment the high-order byte (DPH). No flags are affected. This is the only 16-bit register which can be incremented.

Operation: INC

(DPTR) ← (DPTR) + 1

Bytes: 1

Encoding:

1	0	1	0	0	0	1	1

JB bit, rel

Function: Jump if bit is set

Description: If the indicated bit is a one, jump to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. The bit tested is not modified. No flags are affected.

Operation: JB

(PC) ← (PC) + 3

if (bit) = 1

then (PC) ← (PC) + rel

Bytes: 3

Encoding:

0	0	1	0	0	0	0	0
bit address							
relative address							

JBC bit,rel

Function: Jump if bit is set and clear bit

Description: If the indicated bit is one, branch to the address indicated; otherwise proceed with the next instruction. *In either case, clear the designated bit.* The branch destination is computed by adding the signed relative displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. No flags are affected.

Note: When this instruction is used to test an output pin, the value used as the original data will be read from the output data latch, not the input pin.

Operation: JBC

(PC) ← (PC) + 3

if (bit) = 1

then (bit) ← 0

(PC) ← (PC) + rel

Bytes: 3

Encoding:

0	0	0	1	0	0	0	0
bit address							
relative address							

JC rel

Function: Jump if carry is set

Description: If the carry flag is set, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. No flags are affected.

Operation: JC

(PC) ← (PC) + 2

if (C) = 1

then (PC) ← (PC) + rel

Bytes: 2

Encoding:

0	1	0	0	0	0	0	0	relative address

JMP @A + DPTR

Function: Jump indirect relative to DPTR

Description: Add the eight-bit unsigned contents of the Accumulator with the sixteen-bit data pointer (DPTR), and load the resulting sum into the Program Counter. This will be the address for subsequent instruction fetches. Sixteen-bit addition is performed (modulo 2^{16}): a carry-out from the low-order eight bits propagates through the higher-order bits. Neither the Accumulator nor the data pointer is altered. No flags are affected.

Operation: JMP

(PC) ← (A) + (DPTR)

Bytes: 1

Encoding:

0	1	1	1	0	0	1	1

JNB bit,rel

Function: Jump if bit is not set

Description: If the indicated bit is a zero, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. *The bit tested is not modified.* No flags are affected.

Operation: JNB

(PC) ← (PC) + 3

if (bit) = 0

then (PC) ← (PC) + rel.

Bytes: 3

Encoding:

0	0	1	1	0	0	0	0
bit address							
relative address							

JNC rel

Function: Jump if carry flag is not set

Description: If the carry flag is a zero, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice to point to the next instruction. The carry flag is not modified.

Operation: JNC

(PC) ← (PC) + 2

if (C) = 0

then (PC) ← (PC) + rel

Bytes: 2

Encoding:

0	1	0	1	0	0	0	0	relative address

JNZ rel

Function: Jump if Accumulator is not zero

Description: If any bit of the Accumulator is a one, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The Accumulator is not modified. No flags are affected.

Operation: JNZ

(PC) ← (PC) + 2

if (A) ≠ 0

then (PC) ← (PC) + rel.

Bytes: 2

Encoding:

0	1	1	1	0	0	0	0	relative address

JZ rel

Function: Jump if Accumulator is zero

Description: If all bits of the Accumulator are zero, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The Accumulator is not modified. No flags are affected.

Operation: JZ

(PC) ← (PC) + 2

if (A) = 0

then (PC) ← (PC) + rel

Bytes: 2

Encoding:

0	1	1	0	0	0	0	0	relative address

LCALL addr16

Function: Long call

Description: LCALL calls a subroutine located at the indicated address. The instruction adds three to the Program Counter to generate the address of the next instruction and then pushes the 16-bit result onto the Stack (low byte first), incrementing the Stack Pointer by two. The high-order and low-order bytes of the PC are then loaded, respectively, with the second and third bytes of the LCALL instruction. Program execution continues with the instruction at this address. The subroutine may therefore begin anywhere in the full 64KB Program memory address space. No flags are affected.

Operation: LCALL

(PC) ← (PC) + 3

(SP) ← (SP) + 1

((SP)) ← (PC_{7-0})

(SP) ← (SP) + 1

((SP)) ← (PC_{15-8})

(PC) ← addr15-0

Bytes: 3

Encoding:

0	0	0	1	0	0	1	0
addr15-8							
addr7-0							

LJMP addr16

Function: Long jump

Description: LJMP causes an unconditional branch to the indicated address, by loading the high-order and low-order bytes of the PC (respectively) with the second and third instruction bytes. The destination may therefore be anywhere in the full 64KB Program memory address space. No flags are affected.

Operation: LJMP

(PC) ← addr15... addr0

Bytes: 3

Encoding:

0	0	0	0	0	0	1	0
addr15-8							
addr7-0							

MOV <dest-byte>, <src-byte>

Function: Move byte variable

Description: The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected. This is by far the most flexible operation. Fifteen combinations of source and destination addressing modes are allowed.

MOV A,Rn

Operation: MOV

(A) ← (Rn)

Bytes: 1

Encoding:

1	1	1	0	1	r	r	r

MOV A,direct

Operation: MOV

(A) ← (direct)

Bytes: 2

Note: MOV A,ACC is not a valid instruction. The content of the Accumulator after the execution of this instruction is undefined.

Encoding:

1	1	1	0	0	1	0	1	direct address

MOV A,@Ri

Operation:	MOV
	$(A) \leftarrow ((Ri))$
Bytes:	1
Encoding:	

1	1	1	0	0	1	1	i

MOV A, #data

Operation:	MOV
	$(A) \leftarrow$ #data
Bytes:	2
Encoding:	

0	1	1	1	0	1	0	0	immediate data

MOV Rn,A

Operation:	MOV
	$(Rn) \leftarrow (A)$
Bytes:	1
Encoding:	

1	1	1	1	1	r	r	r

MOV Rn,direct

Operation:	MOV
	$(Rn) \leftarrow$ (direct)
Bytes:	2
Encoding:	

1	0	1	0	1	r	r	r	direct address

MOV Rn, #data

Operation:	MOV
	$(Rn) \leftarrow$ #data

Bytes: 2

Encoding:

0	1	1	1	1	r	r	r	immediate data

MOV direct,A

Operation: MOV

(direct) ← (A)

Bytes: 2

Encoding:

1	1	1	1	0	1	0	1	direct address

MOV direct,Rn

Operation: MOV

(direct) ← (Rn)

Bytes: 2

Encoding:

1	0	0	0	1	r	r	r	direct address

MOV direct,direct

Operation: MOV

(direct) ← (direct)

Bytes: 3

Encoding:

1	0	0	0	0	1	0	1
Direct address (source)							
Direct address (destination)							

MOV direct, @ Ri

Operation: MOV

(direct) ← ((Ri))

Bytes: 2

Encoding:

1	0	0	0	0	1	1	i	direct address

MOV direct, #data

Operation: MOV

(direct) ← #data

Bytes: 3

Encoding:

0 1 1 1	0 1 0 1
direct address	
immediate data	

MOV @ Ri,A

Operation: MOV

((Ri)) ← (A)

Bytes: 1

Encoding:

1 1 1 1	0 1 1 i

MOV @ Ri,direct

Operation: MOV

((Ri)) ← (direct)

Bytes: 2

Encoding:

1 0 1 0	0 1 1 i	direct address

MOV @ Ri,#data

Operation: MOV

((Ri)) ← #data

Bytes: 2

Encoding:

0 1 1 1	0 1 1 i	immediate data

MOV <dest-bit>, <src-bit>

Function: Move bit data

Description: The Boolean variable indicated by the second operand is copied into the location specified by the first operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.

MOV C,bit

Operation: MOV

(C) ← (bit)

Bytes: 2

Encoding:

1	0	1	0	0	0	1	0	bit address

MOV bit,C

Operation: MOV

(bit) ← (C)

Bytes: 2

Encoding:

1	0	0	1	0	0	1	0	bit address

MOV DPTR, #data16

Function: Load data pointer with a 16-bit constant

Description: The data pointer is loaded with the 16-bit constant indicated. The 16 bit constant is loaded into the second and third bytes of the instruction. The second byte (DPH) is the high-order byte, while the third byte (DPL) holds the low-order byte. No flags are affected.

This is the only instruction which moves 16 bits of data at once.

Operation: MOV

(DPTR) ← #data15..0

DPH DPL ← #data15...8 #data7..0

Bytes: 3

Encoding:

1	0	0	1	0	0	0	0
immediate data 15-8							
immediate data 7-0							

MOVC A, @A + <base-reg>

Function: Move code byte

Description: The MOVC instructions load the Accumulator with a code byte, or constant from Program memory. The address of the byte fetched is the sum of the original unsigned eight-bit Accumulator contents and the contents of a sixteen-bit base register, which may be either the data pointer or the PC. In the latter case, the PC is incremented to the address of the following instruction before being added to the

Accumulator; otherwise the base register is not altered. Sixteen-bit addition is performed so a carry-out from the low-order eight bits may propagate through higher-order bits. No flags are affected.

MOVC A, @A + DPTR

Operation: MOVC

(A) ← ((A) + (DPTR))

Bytes: 1

Encoding:

1	0	0	1	0	0	1	1

MOVC A, @A + PC

Operation: MOVC

(PC) ← (PC) + 1

(A) ← ((A) + (PC))

Bytes: 1

Encoding:

1	0	0	0	0	0	1	1

MOVX <dest-byte>, <src-byte>

Function: Move external

Description: The MOVX instructions transfer data between the Accumulator and a byte of external Data memory, hence the X appended to MOV. There are two types of instructions, differing in whether they provide an 8-bit or 16-bit indirect address to the external Data RAM.

In the first type, the contents of R0 or R1 in the current register bank provide an 8-bit address. In the second type, the data pointer generates a 16-bit address.

MOVX A,@Ri

Operation: MOVX

(A) ← ((Ri))

Bytes: 1

Encoding:

1	1	1	0	0	0	1	i

MOVX A,@DPTR

Operation: MOVX

(A) ← ((DPTR))

Bytes:	1
Encoding:	

1	1	1	0	0	0	0	0

MOVX @Ri,A

Operation:	MOVX
	((Ri)) ← (A)
Bytes:	1
Encoding:	

1	1	1	1	0	0	1	i

MOVX @DPTR,A

Operation:	MOVX
	((DPTR)) ← (A)
Bytes:	1
Encoding:	

1	1	1	1	0	0	0	0

MUL AB

Function:	Multiply
Description:	MUL AB multiplies the unsigned eight-bit integers in the Accumulator and register B. The low-order byte of the sixteen-bit product is left in the Accumulator, and the high-order byte in B. If the product is greater than 255 (0FFh) the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.
Operation:	MUL
	(A) ←7-0
	(A) x (B)
	(B) ←15-8
Bytes:	1
Encoding:	

1	0	1	0	0	1	0	0

NOP

Function:	No operation
Description:	Execution continues at the following instruction. Other than the PC, no registers or flags are affected.

Operation: NOP

(PC) ← (PC) + 1

Bytes: 1

Encoding:

0	0	0	0	0	0	0	0

ORL <dest-byte>, <src-byte>

Function: Logical OR for byte variables

Description: ORL performs the bit wise logical OR operation between the indicated variables, storing the results in the destination byte. No flags are affected (except P (Parity bit), if <dest-byte> = A).

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

ORL A,Rn

Operation: ORL

(A) ← (A) $\vee$ (Rn)

Bytes: 1

Encoding:

0	1	0	0	1	r	r	r

ORL A,direct

Operation: ORL

(A) ← (A) $\vee$ (direct)

Bytes: 2

Encoding:

0	1	0	0	0	1	0	1	direct address

ORL A,@Ri

Operation: ORL

(A) ← (A) $\vee$ ((Ri))

Bytes: 1

Encoding:

0	1	0	0	0	1	1	i

ORL A,#data

Operation: ORL

(A) ← (A) ∨ #data

Bytes: 2

Encoding:

0	1	0	0	0	1	0	0	immediate data

ORL direct,A

Operation: ORL

(direct) ← (direct) ∨ (A)

Bytes: 2

Encoding:

0	1	0	0	0	0	1	0	direct address

ORL direct, #data

Operation: ORL

(direct) ← (direct) ∨ #data

Bytes: 3

Encoding:

0	1	0	0	0	0	1	1
direct address							
Immediate data							

ORL C, <src-bit>

Function: Logical OR direct bit with carry flag

Description: Set the carry flag if the Boolean value is a logic 1; leave the carry in its current state otherwise. A slash ("/") preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.

ORL C,bit

Operation: ORL

(C) ← (C) ∨ (bit)

Bytes: 2

Encoding:

0	1	1	1	0	0	1	0	bit address

ORL C,/bit

Operation: ORL

(C) ← (C) ∨ / (bit)

Bytes: 2

Encoding:

1	0	1	0	0	0	0	0	bit address

POP direct

Function: Pop from stack

Description: The contents of the internal RAM location addressed by the Stack Pointer are read, and the Stack Pointer is decremented by one. The value read is the transfer to the directly addressed byte indicated. No flags are affected.

Operation: POP

(direct) ← ((SP))

(SP) ← (SP) - 1

Bytes: 2

Encoding:

1	1	0	1	0	0	0	0	direct address

PUSH direct

Function: Push onto stack

Description: The Stack Pointer is incremented by one. The contents of the indicated variable are then copied into the internal RAM location addressed by the Stack Pointer. Otherwise no flags are affected.

Operation: PUSH

(SP) ← (SP) + 1

((SP)) ← (direct)

Bytes: 2

Encoding:

1	1	0	0	0	0	0	0	direct address

RET

Function: Return from subroutine

Description:	RET pops the high and low-order bytes of the PC successively from the Stack, decrementing the Stack Pointer by two. Program execution continues at the resulting address, generally the instruction immediately following an ACALL or LCALL. No flags are affected.
Operation:	RET $(PC_{15-8}) \leftarrow ((SP))$ $(SP) \leftarrow (SP) - 1$ $(PC_{7-0}) \leftarrow ((SP))$ $(SP) \leftarrow (SP) - 1$
Bytes:	1
Encoding:	

0	0	1	0	0	0	1	0

RETI

Function:	Return from interrupt
Description:	RETI pops the high and low-order bytes of the PC successively from the Stack, and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The Stack Pointer is left decremented by two. No other registers are affected The PSW is *not* automatically restored to its pre-interrupt status. Program execution continues at the resulting address, which is generally the instruction immediately after the point at which the interrupt request was detected. If a lower or same-level interrupt is pending when the RETI instruction is executed, that one instruction will be executed before the pending interrupt is processed.
Operation:	RETI $(PC_{15-8}) \leftarrow ((SP))$ $(SP) \leftarrow (SP) - 1$ $(PC_{7-0}) \leftarrow ((SP))$ $(SP) \leftarrow (SP) - 1$
Bytes:	1
Encoding:	

0	0	1	1	0	0	1	0

RL A

Function:	Rotate Accumulator left
Description:	The eight bits in the Accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected.
Operation:	RL

(An + 1) ← (An) n = 0-6

(A0) ← (A7)

Bytes: 1

Encoding:

0	0	1	0	0	0	1	1

RLC A

Function: Rotate Accumulator left through carry flag

Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.

Operation: RLC

(An + 1) ← (An) n = 0-6

(A0) ← (C)

(C) ← (A7)

Bytes: 1

Encoding:

0	0	1	1	0	0	1	1

RR A

Function: Rotate Accumulator right

Description: The eight bits in the Accumulator are rotated one bit to the right. Bit 0 is rotated into the bit 7 position. No flags are affected.

Operation: RR

(An) ← (An + 1) n = 0-6

(A7) ← (A0)

Bytes: 1

Encoding:

0	0	0	0	0	0	1	1

RRC A

Function: Rotate Accumulator right through carry flag

Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the right. Bit 0 moves into the carry flag; the original value of the carry flag moves into the bit 7 position. No other flags are affected.

Operation: RRC

(An) ← (An + 1) n=0-6

(A7) ← (C)

(C) ← (A0)

Bytes: 1

Encoding:

0	0	0	1	0	0	1	1

SETB <bit>

Function: Set bit

Description: SETB sets the indicated bit to one. SETB can operate on the carry flag or any directly addressable bit. No other flags are affected.

SETB bit

Operation: SETB

(bit) ← 1

Bytes: 2

Encoding:

1	1	0	1	0	0	1	0	bit address

SETB C

Operation: SETB

(C) ← 1

Bytes: 1

Encoding:

1	1	0	1	0	0	1	1

SJMP rel

Function: Short jump

Description: Program control branches unconditionally to the address indicated. The branch destination is computed by adding the signed displacement in the second instruction byte to the PC, after incrementing the PC twice. Therefore, the range of destinations allowed is from 128 bytes preceding this instruction to 127 bytes following it.

Note: Under the above conditions the instruction following SJMP will be at 102h. Therefore, the displacement byte of the instruction will be the relative offset (0123h - 0102h) = 21h . In other words, an SJMP with a displacement of 0FEh would be a one-instruction infinite loop.

Operation: SJMP

(PC) ← (PC) + 2

(PC) ← (PC) + rel

Bytes: 2

Encoding:

1	0	0	0	0	0	0	0	relative address

SUBB A, <src-byte>

Function: Subtract with borrow

Description: SUBB subtracts the indicated variable and the carry flag together from the Accumulator, leaving the result in the Accumulator. SUBB sets the carry (borrow) flag if a borrow is needed for bit 7, and clears C otherwise. (If C was set *before* executing a SUBB instruction, this indicates that a borrow was needed for the previous step in a multiple precision subtraction, so the carry is subtracted from the Accumulator along with the source operand).

AC (Auxiliary Carry bit) is set if a borrow is needed for bit 3 and cleared otherwise. OV (Overflow flag) is set if a borrow is needed into bit 6 but not into bit 7, or into bit 7 but not bit 6.

When subtracting signed integers OV indicates a negative number produced when a negative value is subtracted from a positive value, or a positive result when a positive number is subtracted from a negative number.

The source operand allows four addressing modes: register, direct, register-indirect, or immediate.

SUBB A,Rn

Operation: SUBB

(A) ← (A) - (C) - (Rn)

Bytes: 1

Encoding:

1	0	0	1	1	r	r	r

SUBB A,direct

Operation: SUBB

(A) ← (A) - (C) - (direct)

Bytes: 2

Encoding:

1	0	0	1	0	1	0	1	direct address

SUBB A, @ Ri

Operation: SUBB

$(A) \leftarrow (A) - (C) - ((Ri))$

Bytes: 1

Encoding:

1	0	0	1	0	1	1	i

SUBB A, #data

Operation: SUBB

$(A) \leftarrow (A) - (C) - \#data$

Bytes: 2

Encoding:

1	0	0	1	0	1	0	0	immediate data

SWAP A

Function: Swap nibbles within the Accumulator

Description: SWAP A interchanges the low and high-order nibbles (four-bit fields) of the Accumulator (bits 3-0 and bits 7-4). The operation can also be thought of as a four-bit rotate instruction. No flags are affected.

Operation: SWAP

$(A_{3-0}) \leftrightarrow (A_{7-4})$

Bytes: 1

Encoding:

1	1	0	0	0	1	0	0

XCH A, <byte>

Function: Exchange Accumulator with byte variable

Description: XCH loads the Accumulator with the contents of the indicated variable, at the same time writing the original Accumulator contents to the indicated variable. The source/destination operand can use register, direct, or register-indirect addressing.

XCH A,Rn

Operation: XCH

$(A) \leftrightarrow (Rn)$

Bytes: 1

Encoding:

1	1	0	0	1	r	r	r

XCH A,direct

Operation: XCH

(A) ↔ (direct)

Bytes: 2

Encoding:

1	1	0	0	0	1	0	1	direct address

XCH A, @ Ri

Operation: XCH

(A) ↔ ((Ri))

Bytes: 1

Encoding:

1	1	0	0	0	1	1	i

XCHD A,@Ri

Function: Exchange digit

Description: XCHD exchanges the low-order nibble of the Accumulator (bits 3-0, generally representing a hexadecimal or BCD digit), with that of the internal RAM location indirectly addressed by the specified register. The high-order nibbles (bits 7-4) of each register are not affected. No flags are affected.

Operation: XCHD

(A_{3-0}) ↔ ((Ri_{3-0}))

Bytes: 1

Encoding:

1	1	0	1	0	1	1	i

XRL <dest-byte>, <src-byte>

Function: Logical Exclusive OR for byte variables

Description: XRL performs the bit wise logical Exclusive OR operation between the indicated variables, storing the results in the destination. No flags are affected (except P (Parity bit), if <dest-byte> = A).

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be Accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

XRL A,Rn

Operation:	XRL
	(A) ← (A) ∀ (Rn)
Bytes:	1
Encoding:	

0	1	1	0	1	r	r	r

XRL A,direct

Operation:	XRL
	(A) ← (A) ∀ (direct)
Bytes:	2
Encoding:	

0	1	1	0	0	1	0	1	direct address

XRL A, @ Ri

Operation:	XRL
	(A) ← (A) ∀ ((Ri))
Bytes:	1
Encoding:	

0	1	1	0	0	1	1	i

XRL A, #data

Operation:	XRL
	(A) ← (A) ∀ #data
Bytes:	2
Encoding:	

0	1	1	0	0	1	0	0	immediate data

XRL direct,A

Operation:	XRL
	(direct) ← (direct) ∀ (A)
Bytes:	2

Encoding:

0	1	1	0	0	0	1	0	direct address

XRL direct, #data

Operation: XRL

(direct) ← (direct) ∀ #data

Bytes: 3

Encoding:

0	1	1	0	0	0	1	1
direct address							
immediate data							

Memory timing

Program memory timing

The execution of instruction N is performed during the fetch of instruction N+1.

Program memory Read cycle

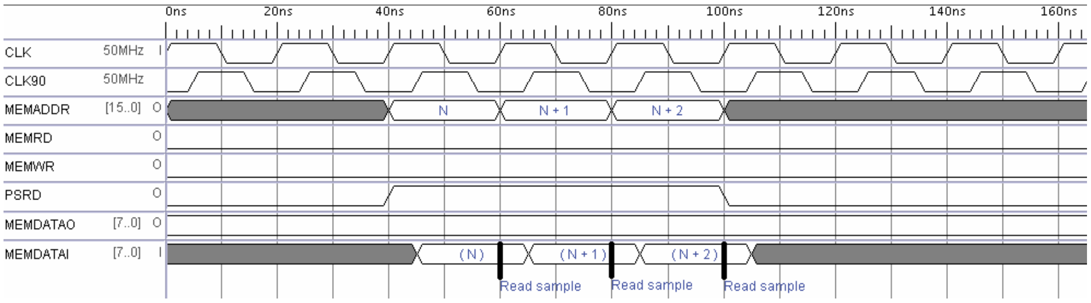


Figure 14. Program memory Read cycle without wait states

Note:		
	CLK	- system clock signal (CLK)
	CLK90	- system clock signal (CLK90)
	N	- address of current instruction to be executed
	(N)	- instruction fetched from address N
	N+1	- address of next instruction to be executed
	Read sample	- point at which data is read from bus into the internal register.

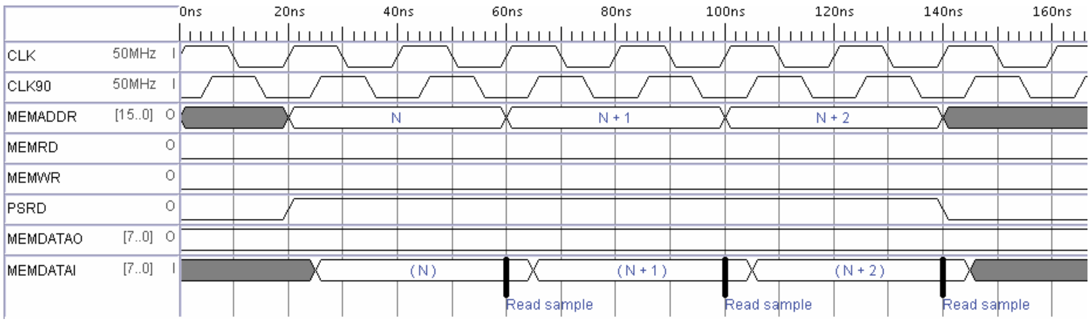


Figure 15. Program memory Read cycle with 1 wait state

Note:	CLK	- system clock signal (CLK)
	CLK90	- system clock signal (CLK90)
	N	- address of current instruction to be executed
	(N)	- instruction fetched from address N
	N+1	- address of next instruction to be executed
	read sample	- point at which data is read from bus into the internal register.

External Data memory timing

External Data memory Read cycle

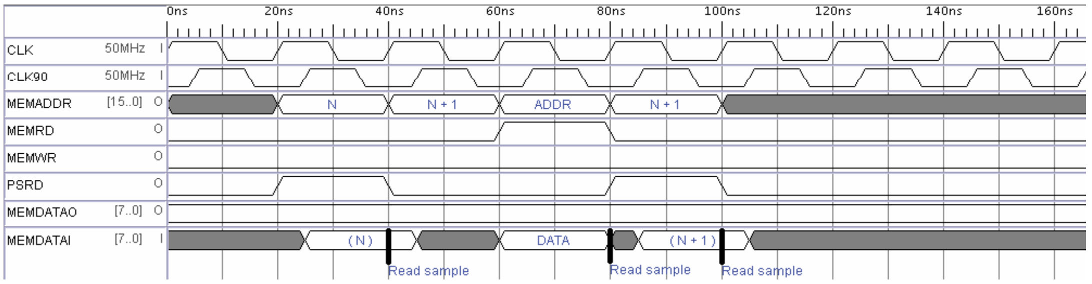


Figure 16. External Data memory Read cycle with stretch 0

Note:	CLK	- system clock signal (CLK)
	CLK90	- system clock signal (CLK90)
	N	- address of current instruction to be executed
	(N)	- instruction fetched from address N
	N+1	- address of next instruction to be executed
	ADDR	- address of memory cell
	DATA	- data to be read from address ADDR
	Read sample	- point at which data is read from bus into the internal register.

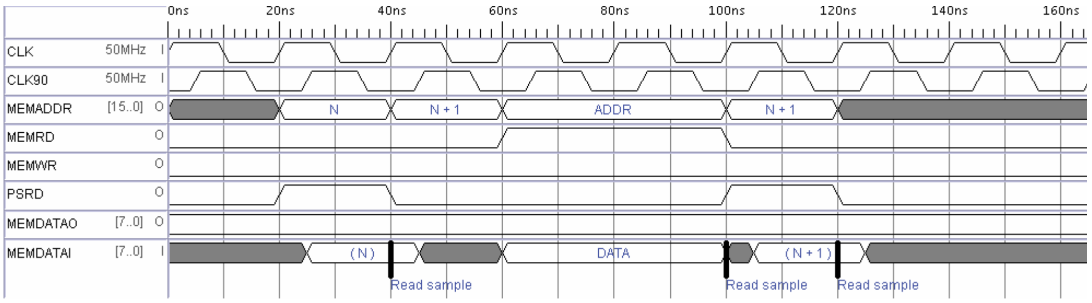


Figure 17. External Data memory Read cycle with stretch 1

Note:		
	CLK	- system clock signal (CLK)
	CLK90	- system clock signal (CLK90)
	N	- address of current instruction to be executed
	(N)	- instruction fetched from address N
	N+1	- address of next instruction to be executed
	ADDR	- address of memory cell
	DATA	- data to be read from address ADDR
	Read sample	- point at which data is read from bus into the internal register.

External Data memory Write cycle

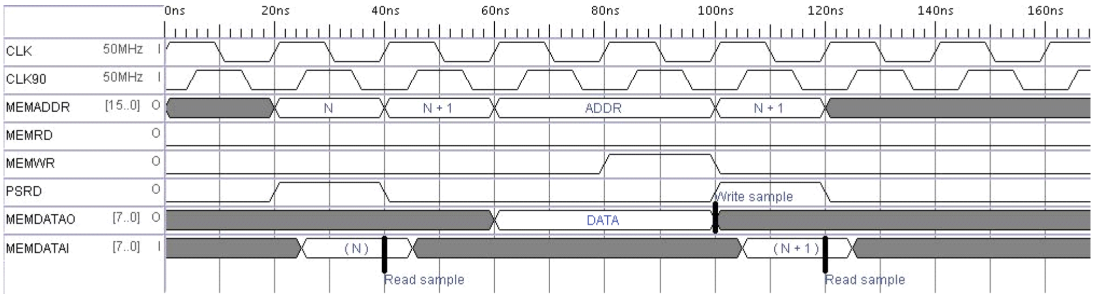


Figure 18. External Data memory Write cycle with stretch 0

Note:		
	CLK	- system clock signal (CLK)
	CLK90	- system clock signal (CLK90)
	N	- address of current instruction to be executed
	(N)	- instruction fetched from address N
	N+1	- address of next instruction to be executed
	ADDR	- address of data memory cell
	DATA	- data to be written into address ADDR
	Write sample	- point at which data is written from the bus into memory.

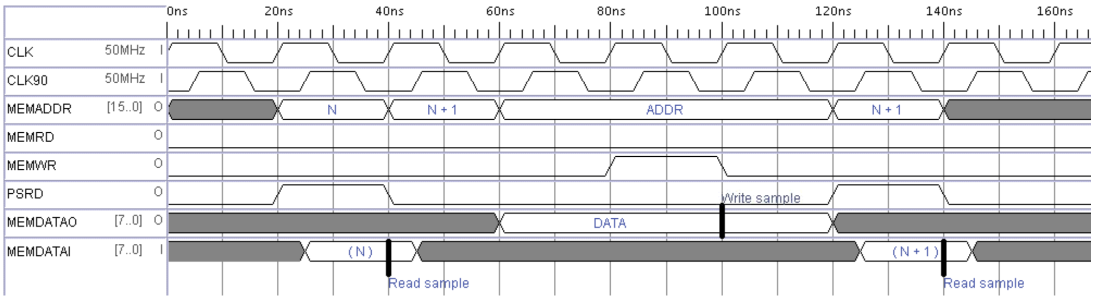


Figure 19. External Data memory Write cycle with stretch 1

Note:		
	CLK	- system clock signal (CLK)
	CLK90	- system clock signal (CLK90)
	N	- address of current instruction to be executed
	(N)	- instruction fetched from address N
	N+1	- address of next instruction to be executed
	ADDR	- address of data memory cell
	DATA	- data to be written into address ADDR
	Write sample	- point at which data is written from the bus into memory.

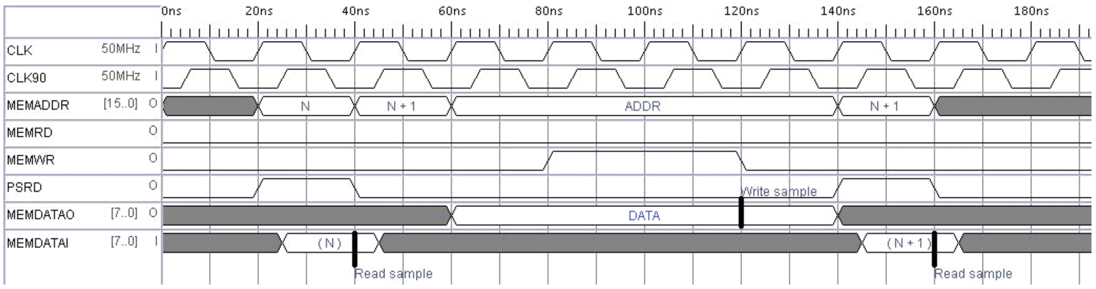


Figure 20. External Data memory Write cycle with stretch 2

Note:		
	CLK	- system clock signal (CLK)
	CLK90	- system clock signal (CLK90)
	N	- address of current instruction to be executed
	(N)	- instruction fetched from address N
	N+1	- address of next instruction to be executed
	ADDR	- address of data memory cell
	DATA	- data to be written into address ADDR
	Write sample	- point at which data is written from the bus into memory.

Revision History

Date	Version No.	Revision
31-Dec-2003	1.0	New product release
01-Oct-2004	1.1	Modifications to Program memory, external Data memory, Interrupts, Priority structure, On-Chip debugging. Addition of XP register and DA A instruction. Removal of DPS, DPL1 and DPH1 registers. Addition of Wishbone versions – TSK52B_W and TSK52B_WD.
03-Nov-2004	1.2	Change to use of bit 0 for the WBT0 register. Also swapped the High/Low descriptions for bit 1 of this register.
08-Feb-2005	1.3	Addition of Wishbone Peripheral memory mapping information, showing alignment with SFR space for the TSK52B_W and TSK52B_WD. Modifications to debug panel information in On-Chip Debugging section.
09-May-2005	1.4	Updated for SP4
12-Dec-2005	1.5	Path references updated for Altium Designer 6