TMS9900 Central Processing Unit

Introduction

The TMS9900 is the CPU, i.e. the brain of the TI-99/4A. This microprocessor executes a machine language program located in memory and controls all the other chips in the computer. It's a real 16 bits microprocessor, which means it has 16 data lines and an address space of 2^^16 bytes, i.e. 64K. To accomodate all these lines, TI had to create an extra-large 64-pins chip, that was quite a novelty by that time. They could even afford the luxury of having 5 non-connected pins!

Pinout

```
        +----+-----+----+
    Vbb |1 o          64| HOLD*
    Vcc |2            63| MEMEN*
   WAIT |3      T     62| READY
  LOAD* |4      M     61| WE*
  HOLDA |5      S     60| CRUCLK
 RESET* |6            59| Vcc
    IAQ |7      9     58| nc
   PHI1 |8      9     57| nc
   PHI2 |9      0     56| D15
    A14 |10     0     55| D14
    A13 |11           54| D13
    A12 |12           53| D12
    A11 |13           52| D11
    A10 |14           51| D10
     A9 |15           50| D9
     A8 |16           49| D8
     A7 |17           48| D7
     A6 |18           47| D6
     A5 |19           46| D5
     A4 |20           45| D4
     A3 |21           44| D3
     A2 |22           43| D2
     A1 |23           42| D1
     A0 |24           41| D0
   PHI4 |25           40| Vss
    Vss |26           39| nc
```

```
    Vdd |27           38| nc
   PHI3 |28           37| nc
   DBIN |29           36| IC0
 CRUOUT |30           35| IC1
  CRUIN |31           34| IC2
INTREQ* |32           33| IC3
        +---------------+
```

Power supply
Vbb: -5V Vcc: +5V Both pins (2 and 59) must be connected.
Vdd: +12V Vss: Ground. Both pins (26 and 40) must be connected.

Clock
Phi1-Phi3 are 4 input pins that receive the same signal from the TMS9904 clock generator, with one exception: each signal is shifted by 1/4 of a phase with respect the the previous one.

Data bus
D0-D15: These 16 pins are used to read or write data. Note that contrarily to almost anybody else, TI made D0 the most significant bit (weight >8000) and D15 the least significant bit (weight >0001).

Address bus
A0-A14: These 15 output pins are used the specify the address of 32Kwords in memory. Each word is two bytes long (since we have 16 data lines), thus we are effectively addressing 64Kbytes. On the TI-99/4A, the data bus is multiplexed as 2 x 8 bits for almost every purpose except accessing the console ROMs and the scratch-pad RAM. The multiplexing circuitery controls a pseudo-address line A15 that indicated whether the 8-bit data bus contains the most significant byte of the 16-bit bus (A15=0) ot the least significant byte (A15=1).

Bus control
MEMEN* Memory enable. When active low, this pin indicated that the TMS9900 wants to access memory and has placed a valid address on the address bus.

DBIN Data bus in. When active (high) this pin indicates that the TMS9900 is ready to accept data.

WE* Write enable. When active (low) this pin indicates that the TMS9900 has placed valid data on the bus.

CRU control
CRUCLK CRU clock. When active (high) this pin indicates the the TMS9900 is performing a CRU operation (or an external instruction).

CRUOUT This output pin contains the data that the TMS9900 sends out during CRU operations.

CRUIN This pin is used by the TMS9900 to input data during CRU operations.

Interrupt control
INTREQ* When active (low) this input pin signals the TMS9900 that an interrupt is pending. If it accepts the interrupt, the TMS9900 will perform `BLWP @>0000` through `BLWP @>003C` depending on the interrupt level.

IC0-IC3 These 4 input pins indicate the level of the interrupt (0-15). The LIMI instruction can be used to define the "cutoff" level, above which the TMS9900 will ignore interrupts. On the TI-99/4A those pins are hardwired as low,low,low,high which means all interrupts are level 1.

LOAD* Non-maskable interrupt. When active low, this pin forces the TMS9900 to perform a `BLWP @>FFFC` interrupt. If it remains low, interrupts continue to be issued, thus is should not remain low for more than one instruction.

IAQ Instruction acquisition. This output pin indicates that the TMS9900 is acquiring an instruction. It can be used to detect illegal opcodes or to prevent LOAD* to last longer than one instruction. On the TI-99/4A, IAQ and HOLDA are combined via an OR gate and presented to the peripheral port. However, the flex cable connector does not carry that signal to the PE-Box.

RESET* When active (low) this input pin resets the TMS9900 and inhibits WE* and CRUCLK. RESET* must remain low for at least 3 clock cycles. As soon as it becomes high again, the TMS9900 performs a BLWP @>0000 (note that it is the same vector as interrupt level 0).

Memory control
HOLD* When active (low) this pins tells the TMS9900 that a DMA controller wants to perform Direct Memory Access. The TMS9900 set D0-D15, A0-A14, MEMEN*, DBIN and WE* in high impedance state (isolated) then activate HOLDA and waits until HOLD* becomes high again. On the TI-99/4A this pin is hardwired high, which means we cannot perform DMA operations.

HOLDA Hold acknowledge. This output pin is used to tell the DMA controller that the it can perform direct memory access.

READY When active (high) this input pin tells the TMS9900 that the memory is ready to read or write data. If it's low, the TMS9900 enters a wait state and suspends all operations until READY becomes high again. On the TI-99/4A this line is used to multiplex the data bus, i.e. handle it as two 8-bit bytes, instead of one 16-but word. WAIT When active (high) this output pin indicates that the TMS9900 is now in wait state.

Execution speed

To execute an instruction, the TMS9900 must first fetch it from memory, which takes a few clock cycles (depending on the memory), then is must execute the instruction wich takes more clock cycles (depending on the instruction) and may require fetching one or two arguments from the memory (again, more clock cycles according to the addressing mode and memory type).

On the TI-99/4A, there are two kinds of memory: 16-bit and 8-bit. Console ROMs (address >0000-1FFF) and the RAM scratch-pad (address >8300-83FF) are the only 16-bit memories. All the rest, including peripheral cards, and memory-mapped devices (GROM, VDP, sound and speech chips) are accessed in a byte-wise manner. This requires multiplexing the 16-bit data bus in two 8-bits chunks. An electronic circuitery in the console takes care ot that burden and uses the READY line to halt the TMS9900 until the peripheral has received/sent the second data byte. Therefore, accessing such a memory results in 4 wait states for each memory access.

The table below can be used to calculate how long an instruction takes to be executed.

- The first column lists the instructions in alphabetical order.
- The second column indicates how many clock cycles are required to execute that instruction in its most primitive form (generally: using workspace registers).
- The third column indicates how many memory access operations are required to fetch the instruction from the program memory and the operands from their memory. In most cases, only one memory access is needed to retrieve the instruction, except for immeidiate instructions that need a second cycle to retrieve the immediate value embedded in the program. If the program is in slow memory, it will take 4 more clock cycles for each of these. All other memory accesses are used to read and write the arguments. If the memory in question is not the console ROMs nor the scratch-pad RAM, add 4 clock cycles per access operation.
- The last two columns indicate whether the instruction required fetching arguments. Some don't, some only need a source arguments, some need two arguments (source and destination).
- In case the arguments are not register, the instruction may require more clock cycles and/or memory access than what's listed in column 2 and 3. Table 2 allows to calculate the number of extra clock cycles required to access arguments.

Instruction	Clock cycles	Memory access	Source	Destination
A	14	4	Y	Y

AB	14	4	Y	Y
ABS(pos)	12	2	Y	-
(neg)	14	3	Y	-
AI	14	4	-	-
ANDI	14	4	-	-
B	8	2	Y	-
BL	12	3	Y	-
BLWP	26	6	Y	-
C	14	3	Y	Y
CB	14	3	Y	Y
CI	14	3	-	-
CKOF	12	1	-	-
CKON	12	1	-	-
CLR	10	3	Y	-
COC	14	3	Y	-
CZC	14	3	Y	-
DEC	10	3	Y	-
DECT	10	3	Y	-
DIV (ovf)	16	3	Y	-
(no ovf)	92-124 (1)	6	Y	-
IDLE	12	1	-	-
INC	10	3	Y	-
INCT	10	3	Y	-
INV	10	3	Y	-
Jump (taken)	10	1	-	-
(not taken)	8	1	-	-
LDCR	20 +2*bits	3	Y	-
LI	12	3	-	-
LIMI	16	2	-	-
LREX	12	1	-	-
LWPI	10	2	-	-
MOV	14	4	Y	Y
MOVB	14	4	Y	Y
MPY	52	5	Y	-
NEG	12	3	Y	-
ORI	14	4	-	-
RSET	12	1	-	-
RTWP	14	4	-	-
S	14	4	Y	Y
SB	14	4	Y	Y
SBO	12	2	-	-
SBZ	12	2	-	-
SETO	10	3	Y	-

Shift	12 +2*disp	3	-	-
(disp in R0)	20 +2*disp	4	-	-
SOC	14	4	Y	Y
SOCB	14	4	Y	Y
STCR (1-7)	42	4	Y	-
(8 bits)	44	4	Y	-
(9-15 bits)	58	4	Y	-
(16 bits)	60	4	Y	-
STST	8	2	-	-
STWP	8	2	-	-
SWPB	10	3	Y	-
SZC	14	4	Y	Y
SZCB	14	4	Y	Y
TB	12	2	-	-
X (note 2)	8	2	Y	-
XOP	36	8	Y	-
XOR	14	4	Y	-
Illegal	6	1	-	-
Interrupts	22	5	-	-
Reset	26	5	-	-

Notes
1) DIV execution time, when no overflow occurs, depends on the partial quotient after each clock cycle during execution.
2) For X, add this time to the execution time of the instruction found at the source address, minus 4 clock cycles and 1 memory access.

For each source and destination arguments (if any) add the following:

Address mode	**Clock cycles**	**Memory access**
Rx	0	0
*Rx	4	1
*Rx+ (byte)	6	2
(word)	8	2
@>xxxx	8	1
@>xxxx(Rx)	8	2

Note
For the *Rx+ addressing mode, the number of clock cycles depends on whether the register must be incremented by 1 or by 2. The byte-oriented operations increment it by 1 and use 6 clock cycles, these are: AB, CB, MOVB, SB, SOCB and SZCB. In addition, the LDCR and STCR are considered as byte operations if they transfer 1 to 8 bits (with 9 to 16 bits they are word operations and use 8 clock cycles).

Examples of calculations

LIMI 2
The LIMI instruction uses 16 clock cycles and 2 memory access operations: one to fetch the LIMI instruction, one to fetch the immediate value 2.

```
Execution: 16 cycles
1 mem access to read LIMI }
1 mem access to read 2    } add 8 cycles if program is in slow mem.
```

This adds up to:
16 cycles if the instruction is in the ROMs or the scratch-pad,
16+2*4=24 clock cycles otherwise (remember, there are 4 wait states per memory access).

With a 3 MHz clock that cycles every 333 nanoseconds, this boils down to 16*333 ns = 5.33 microseconds or 24*333 ns = 8 microseconds, depending on which memory the instruction is in. Tip: divide the number of clock cycles by 3 to find the execution time in microseconds (with a 3 MHz clock).

CLR R2
The CLR instruction uses 10 clock cycles and 3 memory access operations. Depending on which memory it is in, it may require from 10 to 22 clock cycles. But CLR also takes an argument that we must consider. In this case, the argument is a register and the clock cycles needed for its access can be found in table 1.

```
Execution: 10 cycles
1 mem access to read CLR: add 4 cycles if program is in slow mem.
1 mem access to read R2   }
1 mem access to write R2  } Add 8 cycles if workspace is in slow mem
```

CLR @TEST
The CLR instruction itself still uses 10 or 22 clock cycles to execute, but now dealing with the argument requires 8 extra clock cycles and 1 extra memory access operations (to read the address of TEST from the word following CLR in the program).

```
Execution: 10 cycles
Argument @xxx: 8 cycles
1 mem access to read CLR   }
1 mem access to read TEST  } Add 8 cycles if program is in slow mem
1 mem access to read the contents of @TEST }
1 mem access to write to @TEST             } Add 8 cycles if TEST is in slow mem
```

If the address TEST is not in the ROMs nor in the scratch-pad, it will add 8+2*4=16 clock cycles to the execution time. Otherwise it just adds 8 cycles. We thus have:

```
10+8   = 18 cycles if program and TEST are both in fast memory.
10+8+8 = 26 cycles if program in slow memory, but TEST is in fast memory.
10+8+8 = 26 cycles if program is in fast memory, but TEST is in slow memory.
10+8+8+8 = 34 cycles if both program and TEST are in slow memory.
```

CLR *R2
Fetching the source argument from the address found in register R2 requires 4 clock cycles and 1 memory access operation. Depending whether the workspace is in the scratch-pad or not (no workspace should ever be in ROM!), this may require 4 additional clock cycles. Depending whether the target address found in R2 (i.e. *R2) is in slow or fast memory, we may have to add 4 more cycles.

```
Execution: 10 cycles
Argument *Rx: 4 cycles
1 mem access to read CLR: Add 4 cycles if program is in slow mem
1 mem access to read R2: Add 4 cycles if workspace is in slow mem
1 mem access to read the contents of *R2 }
1 mem access to write to *R2             } Add 8 cycles if R2 points to slow mem
```

So the total could be:

```
10+4 = 14 cycles if CLR, workspace and *R2 are all in fast memory.
10+4+4 = 18 if either CLR or the workspace is in slow memory.
10+4+4+4 = 22 if both CLR and workspace are in slow memory
10+4+8   = 22 if R2 points to slow memory
10+4+8+4 = 26 if R2 points to slow memory and either CLR or WS is in slow memory
10+4+8+4+4 = 30 if everything is in slow memory.
```

LDCR R1,7

The LDCR instruction itself requires 20 clock cycles. There may be additonal cycles require to retrieve the instruction, or the register. Writing the CRU definitely implies 2 additional clock cycles per bit transfered: in this case we are transfering 7 bits, thus we'll eat 2*7=14 cycles.

```
Execution: 20 cycles
1 mem access to read LDCR: Add 4 cycles if program is in slow mem
1 mem access to read R1: Add 4 cycles if workspace is in slow mem
1 cru write operation: Add 2 cycles per bit transfered (here: 2*7=14)
```

LDCR *R1+,7

Now this one gets tricky: the LDCR instruction requires 20+2*7=14 clock cycles plus whatever may be needed for the memory access operations, just as above. But now we must allocate time to increment R1 after execution. Since we are transfering only one byte (that is, less than one byte: only 7 bits), R1 will be incremented by 1, which requires 6 clock cycles and 2 memory accesses. Depending on where the workspace is, this could add up to 6+2*4=14 cycles.

```
Execution: 20 cycles
1 mem access to read LDCR: Add 4 cycles if program is in slow mem
1 mem access to read R1: Add 4 cycles if workspace is in slow mem
1 mem access to read *R1: Add 4 cycles if source is in slow mem
Incrementing R1: 6 cycles (byte operation)
1 mem access to write back R1: Add 4 cycles if workspace is in slow mem
1 cru write operation: Add 2 cycles per bit transfered (here: 7*2=14)
```

LDCR *R1+,14

Here, we are transfering 14 bits, thus the LCDR instruction takes 20+2*14=48 cycles, plus what's needed for the memory access. In addition, since we are transfering more that 1 byte, R1 will be incremented by two, which requires 8 clock cycles instead of the 6 mentionned above.

```
Execution: 20 cycles
1 mem access to read LDCR: Add 4 cycles if program is in slow mem
1 mem access to read R1: Add 4 cycles if workspace is in slow mem
1 mem access to read *R1: Add 4 cycles if source is in slow mem
Incrementing R1 by two: 8 cycles (word operation)
1 mem access to write R1: Add 4 cycles if workspace is in slow mem
1 cru write operation: Add 2 cycles per bit transfered (here: 14*2=28 cycles)
```

SRL R2,4

The SRL shift operation requires 12 clock cycles, plus 2 cycles for each position shifted. Since the displacement is 4 in this example, it will require 12+2*4=20 cycles. Plus the number of cycles required for 3 memory access operations: 0 to12 depending on the position of the program and the workspace.

```
Execution: 12 cycles
Shifting: 2 cycles per position
1 mem access to read SRL: Add 4 cycles if program is in slow mem
1 mem access to read R2   }
1 mem access to write R2  } Add 8 cycles if workspace is in slow mem
```

SRL R2,0

Here, we are fetching the displacement from R0 (let's say it contains 5). This indirect shift operation requires 20+2*5=30 cycles, and 4 memory access operations instead of 3, as we must now fetch R0.

```
Execution: 12 cycles
Shifting: 2 cycles per position
1 mem access to read SRL: Add 4 cycles if program is in slow mem
1 mem access to read R2  }
1 mem access to read R0  } Add 12 cycles if workspace is in slow mem
1 mem access to write R2 }
```

These calculations are a pain to perform, aren't they? I'm playing with the idea to write an optimisation helper, i.e. a program that would read an assembly source file, and produce a corresponding output file with the execution times listed as comments. But I don't know when I will have time for that. Anyone aware of such a program around here?

Optimizing for speed

Now we can see what are the cycle-hungry operations: `DIV, MPY, LDCR, STCR, XOP` and `BLWP`.

That's why it is often wise to perform a multiplication using shifts and additions rather than `MPY`:

```
       MPY  R0,R8
```

Requires 72 cycles to execute (52 in 16-bits memory). And that's the fastest `MPY`.

Now if R0 contains 8, we could have written:

```
       SLL  R8,3
```

Which does the same, but only uses 30 cycles (18 in fast memory).

To multiply by ten, we could do:

```
       SLL  R8,1               Multiply by two
       MOV  R8,R9
       SLL  R8,2               And then by 4 (which makes 8)
       A    R9,R8              Add it up: 2+8=10.
```

This requires 58 cycles in fast memory and 114 in slow memory. True, this is slower than the initial MPY, but we may have a use for the intermediary result in R9 (that is, R8 times two).

For the same reason, many programers avoid calling subroutines with `BLWP-RTWP` and favor `BL-B *R11`, at least in critical regions of their programs.

We can't do much about `LDCR` and `STCR`, but this is less of a problem: these instructions are rarely used anyhow, and the limiting factor may well be the hardware they are addressing (although not very likely: TTLs are fast).

Test program

All this theory is impressive, but we'd like to verify whether it is true in the real word. Let's write a little test program and time its execution with a stop watch (you may want to time it automatically, with the TMS9901 timer, but that's another story).

```
START  LWPI >A800            Load our workspace      ** 1 **
       LI   R1,DELAY         The subroutine we want to time
       LI   R2,>B000         Where it will run       ** 2 **
       MOV  R2,R3
       LI   R0,EOPG-DELAY    Subroutine size
LP0    MOV  *R1+,*R2+        Copy subroutine in target memory
       DECT R0
```

```
       JNE  LP0
*      B    @RET              To return immediately   ** 3 **
       B    *R3               Call subroutine

DELAY  LI   R1,100            You can change this value
LP1    CLR  R0
LP2    DEC  R0                Inside loop, executes 65536 times
       JNE  LP2
       DEC  R1                Outside loop
       JNE  LP1

RET    LWPI >20BA             Assuming editor/assembler workspace
       B   *R11               Done, return to editor/assembler module
EOPG   END
```

The test program first copies the timed routine in memory. This could be the scratch pad memory or the memory expansion. Then it executes the delay loop. Once it is done, it returns to the caller. I have assumed that this is the Editor/Assembler cartridge (oe Funnelweb). If it's not, modify the return instrucutions accordingly.

Now let's do some measurements. First of all, assemble the program with the `B @RET` in line ** 3 **. The delay loops will be skipped and the program will return immediately. This allows us to account for the time it takes the Editor/Assembler module to enter our program, and to display the <press any key> message when returning from it. As you'll see, this is so fast that we cannot time it..

Then let's comment out line ** 3 ** and time our program in four different situations: Modify line ** 1 ** to use a workspace in the memory expansion (>A800) or in the scratch-pad (>83E0). For each of those, modify line ** 2 ** to copy the delay loop in the memory expansion (>B000) or in the scratch-pad (>8300). Write the resulting times in this table:

Program / Worskpace	**Memory expansion**	**Scratch-pad**
Memory expansion	89 sec (100%)	62 sec (70%)
Scratch-pad	62 sec (70%)	44 sec (49%)

Of course, the faster way is to have both the program and the workspace in the scratch-pad: in our case, it's twice as fast as the slowest solution. Unfortunately, this is not practical as the scratch-pad is only 256 bytes long. And many of those bytes have special meanings. Most of the time however, it is possible for you to place your workspace in the scratch-pad: `LWPI >8300` for instance. This will substantially speed up your program (in our case, by 30%), especially if you are carefull to reserve your registers for frequently used variables.

Now, if there are some speed-critical routines in your program (such a scrolling the screen left/right in an arcade game), and if they are small enough, you could copy them in the scratch-pad as we did above and execute them there. Say at >8320, not to overwrite your own workspace.

But anyhow, there is more to gain with the workspace than with the program. Consider the instruction `A R1,R2` for instance: it requires one memory operation to ftech the instruction from the program memory, and three accesses to the workspace (to read R1, to read R2 and to write back R2). So by placing the workspace in fast memory you gain three times more than by plancing the instruction itself in fast memory.

To speed up the program itself, you could optimize your code to avoid using those instructions that require a lot of time to execute, as discussed above. And often an improved algorithm will do better than nay optimization trick!

Instructions issues

It is not the purpose of these pages to teach assembly language. Thus I won't discuss in detail the meanings of each and every instructions (see my assembly language primer). However, there are a few that are worth noticing.

External instructions

There are five so-called external instructions: **`LREX`** (Load and Restart EXecution), **`CKOF`** (clock off), **`CKON`** (clock on), **`RSET`** (reset) and **`IDLE`**. The first four were used in the 990 microcomputer and have no special meaning on the TMS9900. They just place a special code on address lines A0-A2 and send a pulse on the CRUCLK pin. `RSET` also set the interrupt mask to zero, just like a `LIMI 0`.

`IDLE` puts the TMS9900 in an idle state in which it remains until an interrupt, a RESET* or a LOAD* signal occurs. During that time, the processor repeatedly places the special code for `IDLE` on lines A0-A2 and pluses the CRUCLK pin.

The special codes are the following:

Instruction	A0	A1	A2	A3-A14
LREX	H	H	H	n/a
CKOF	H	H	L	n/a
CKON	H	L	H	n/a
RSET	L	H	H	n/a
IDLE	L	H	L	n/a
CRU operations	L	L	L	From R12

These instructions should not be used on the TI-99/4A because neither the console nor any peripheral card I know of bothers with decoding lines A0-A2 to distinguish a CRU operations from an external instructions. In other words, the external instructions would be mistaken for CRU operations and could cause havoc.

We could however make use of them if we were to make a slight modification to the console: use a 74LS138 decoder to intercept the CRUCLK line and let the signal through only if the CRU code is present on lines A0-A2. The same decoder would activate five different lines, one for each external instruction, that would allow us to trigger five external devices.

```
         +---------+
A0-------|C     Y0*|-------|>o---CRUCLK
A1-------|B     Y1*|
A2-------|A     Y2*|-----IDLE*
         |      Y3*|-----RSET*
CRUCLK---|G1    Y4*|
         |      Y5*|-----CKON*
      +--|G2A*  Y6*|-----CKOF*
      +--|G2B*  Y7*|-----LREX*
      |  |         |
      |  +---------+
     Gnd
```

Note that we'll also need an inverter to make CRUCLK active high after the decoder. Very conveniently, there is a 74LS04 in the console with 3 non-connected inverters. It is located just above the TMS9900, right were we need it.!

Why would we need such a circuit?
- We could use the `IDLE` instruction to enter an idle state and wait for an interrupt.
- We could use a line (RSET* jumps to mind) the send a reset signal the the TMS9904 clock driver. This will perform a hardware reset programmatically, as opposed to `BLWP @>0000` that only performs a software reset (i.e. does not physically reset the peripheral chips).

- We could use the CKON* and CKOF* lines to switch the clock speed, by feeding the appropriate signals to the TMS9904.
- More generally, we could use those line to activate any kind of hardware we want.

For instance, I was told (by Anders Persson) that the "Cortex" computer, which was sold as a kit in the 80s, made use of these instructions for the following:
- RSET caused a hardware reset
- IDLE lit an external LED. The system would issues lots of IDLE when it had nothing to do, so you could see that.
- CKON and CKOF enabled and disabled a memory mapper system, in the style of the one used by the SuperAMS card.
- LREX triggered a delayed interrupt after two instructions (via a series of flip-flops clocked by the IAQ line). This was used by a debugger to execute single instructions, by setting up a pseudo-RTWP pointing at a given instruction, and then doing LREX RTWP. The RTWP would go back to the desired instruction, which was executed, and then the interrupt would fire and the debugger was back in control. Nifty, isn't it?

XOP instruction

The `XOP` (eXtended OPeration) instruction is kind of a special `BLWP`. It takes a source argument and an operation number from 0 to 15. The `XOP` instructions uses that number to perform a `BLWP` to one among 16 vectors located in memory addresses >0040-007F. In addition, it places the content of the source argument in the R11 register of the new workspace. Finally, bit 6 (weight >0200) is set in the status register while a `XOP` instruction is executed. Note that the TMS9900 does not test the INTREQ* interrupt request pin after a `XOP` operation.

The main advantage of `XOP` is that it only requires one word. Therefore, we could use it to replace any word in a program and interrupt its execution. That's how my debugger RIP v.2 works: to set a breakpoint is saves the content of a memory address and replaces it with an `XOP 1`. Execution of this `XOP` results in activating RIP that can then execute the saved instruction and/or ask the used what to do. True, `BLWP *Rx` is only one word long, and so is `BLWP Rx`, but both expect special values in the registers. The first one want a pointer to the WR and PC vectors in Rx, the second want the WR vector in Rx and the PC vector in the next register. `XOP` is much more convenient.

The drag is that all vectors for XOPs are in ROM memory, and only three of them (two with some consoles) have usefull values. That's because the GPL interpreter code begins right there. However, there are some eight empty words at the end of the console ROM, TI could just have shifted up the whole stuff and provide us with 4 more XOPs. Oh well, we can do with the first three. Not to mention that some of the following vectors happen to contain usefull values.

Address	XOP	WR	PC	Comments
>0040	0	>280A	>0C1C	Enters the extended GPL card
>0044	1	>FFD8	>FFF8	Very usefull for us
>0048	2	>83A0	>8300	Very usefull for us, but not always present
>004C	3	>1100	>06A0	
>0050	4	>0864	>06A0	
>0054	5	>0864	>C90D	
>0058	6	>8300	>C342	Could be used, although not meant to be so
>005C	7	>D11D	>C180	Dangerous (odd WR)
>0060	8	>DB46	>0402	Pops inside the keyscan routine with wrong WS
>0064	9	>0B60	>83ED	
>0068	10	>0402	>5802	

>006C	11	>011B	>837C	
>0070	12	>0300	>0002	
>0074	13	>0300	>0000	
>0078	14	>D25D	>1105	Pops inside XML >0E with wrong WS
>007c	15	>D109	>09C4	Pops intp the ISR (sprite motion) with wrong WS

Now, any WR value that maps to the console ROM (below >2000) is useless as it will result in loosing the return address. I suspect that odd workspace addresses may also cause havoc. The same is true for PC values: we don't want to branch to ROM routines. Odd PC values have less importance since the TMS9900 will ignore the least significant bit anyway.

XOP 0 is hardwired to switch on a peripheral card whose CRU base address should be >1B00, then enters its ROM at address >4028, after having changed the worskpace to >2800. My guess is that this card was meant to implement extra GPL opcodes, but I don't think it has ever been released. Note that XOP 0 will not check whether the card is here or not before branching. If there is no such card, the TI-99/4A will crash. Now this is the ideal instruction to use to implement a debugger board...

XOP 1 is extremely usefull for us. All we need to do is to place a `B @MYPROG` at location >FFF8 and XOP 1 will enter our program. Note that this will preserve the LOAD interrupt vectors at locations >FFFC-FFFF.

XOP2 Be carefull about XOP 2: some consoles do not support it... If you decide to use it, you should probably place a `B @WHERE` instruction right at >8300 or soon after, since scratch-pad RAM is a precious resource. Also, having our workspace at >83A0 may disturb the data stack of the GPL interpreter...

XOP 6 The vectors are not guarantied to have these values on each and every console...

XOP 8, XOP 14, XOP 15 land in the middle of various routines in the console ROMs. These routines expect a workspace of >83E0 which won't be the case. What happens then depends on the contents of the workspace in use, of the >83E0 workspace and possibly of other bytes in the scratch-pad. It is not unconceivable that you can come up with a valid combination that would do something usefull, but why bother?

Another (admitedly not so usefull) trick with the XOP instructions is to use them to build a routine that can be called either with `BLWP` or with `BL`. This relies on the fact that `XOP` opcodes have a value of >2Cxx, which can serve as a valid workspace. See, like this:

```
MYSUB  XOP  R0,1                This is equivalent to DATA >2C40 (WS vector)
       DATA MYSB1               PC vector

MYSB1  ...                      Do something
       RTWP                     Return to caller (or to BLS)

*----------------------------------------------------
* This routine transforms BL calls into BLWP calls
*----------------------------------------------------
BLS    MOV  R13,@>2C5A          Put user's workspace pointer in future R13
       MOV  @22(R13),@>2C5C     Put return address in future R14
       MOV  R15,@>2C5E          Put user's status in future R15
       MOV  *R14+,@>2C56        Get address from PC vector, in future R11
       LWPI >2C40               Change workspace
       B    *R11                Branch to routine

*----------------------------------------------------
* Set up XOP 1
*----------------------------------------------------
       AORG >FFF8
       B    @BLS                Entered by XOP 1

       END
```

If the procedure is called with `BLWP @MYSUB` it will treat the word at MYSUB as a worskpace pointer (of value >2C40) and begin execution of the subroutine at MYSB1 with a workspace of >2C40. Return to the caller is performed by a plain vanilla `RTWP`

If it is called with `BL @MYSUB` it executes XOP 1 which immediately branches to BLS with workspace >FFD8. BLS is a routine that converts `BLs` into `BLWPs`: it gets the PC vector from the data word following the XOP 1 instruction (in this case MYSB1) and puts it in R11 of workspace >2C40 (that receives the source argument R0 upon XOP execution). Then it copies the user's worskpace pointer and status (saved by XOP 1) into R13 and R15 of workspace >2C40. It gets the return address from the R11 in the user's workspace and places it in R14 of workspace >2C40. Finally it branches to the called subroutine, which will never be aware of all the above: it can just assumed it was called with `BLWP`, access parameters accordingly and return with `RTWP`.

That's a helluva slow way to call a subroutine, but it might be usefull in cases...

X instruction

This instruction can be used to simulate another instruction: just place the corresponding code in the source argument of the X (eXecute) instruction.

Example:

```
  LI   R0,>37C3     >37C3 means STCR R3,15
  X    R0
```

If the executed instruction has operands, they will be fetched from the words following `X`, which somewhat limitates the usefullness of `X`. If it were to fetch the operands from the words following the operand of the `X` instruction, it would be a wonderfull way to write a debugger: place a memory pointer in R1 and then do: `X *R1+` Ok,ok it's not that simple: we must trap the jumps and branches, and account for instructions that use R1, but you get the idea. Unfortunately, that's not the way `X` works...

Nevertheless, `X` may be usefull to replace a test in a frequently executed loop. For instance:

```
       MOV  @CLEAR,R2
       MOV  R5,R5         Performs some test
       JEQ  SK1           Decide whether to clear or set to one
       MOV  @SET,R2       Set to one

SK1    LI   R0,>2000      This loop is executed >2000 times
       LI   R1,BUFFER
LP1    X    R2            Equals CLR *R1+ if R5 was null, SETO *R1+ otherwise
       DEC  R0
       JNE  LP1
       ...

CLEAR  CLR  *R1+         Simple alternative to calculating the values
SET    SETO *R1+         of these instructions to put them in R2
```

We could have written:

```
       LI   R0,>2000
       LI   R1,BUFFER
LP1    MOV  R5,R5        Perform the test inside the loop
       JEQ  SK1
       SETO *R1+         Set
       JMP  SK2
SK1    CLR  *R1+         Clear
SK2    DEC  R0
       JNE  LP1
       ...
```

But the first way is much faster, since we don't have to repeat the test at each execution of the loop.

C instruction

Appart for comparison, this instruction can also be used to increment a register by four:

```
  C   *Rx+,*Rx+
```

This uses only one word of memory as opposed to the equivalent :

```
  INCT Rx
  INCT Rx
```

Note that the corresponding CB instruction would increment the register by two, but there is no advantage over a plain vanilla `INCT` in this case.

Timing diagrams

Clock signals

The TMS9900 is meant to be fed a 3 MHz clock signal (that's right, 3 not 30) by the TMS9904 clock generator. This signal comes on 4 different lines, each one being shifted by a quarter of a phase with respect to the previous one. A graphical representation of these signals looks like this:

```
   -|-____      333 ns            -|-____
__/ 45  \___________________/     \______ Phi 1
         _____                      _____
________/     \____________________/      Phi 2
   | 83  |        _____
_______________/     \___________________ Phi 3
              12     12_____
______________________/     \____________ Phi 4
                     5
```

The period of the clock, i.e the time between two pulses in a given phase is 333 nanoseconds for a 3 Megaherz clock (a nanosecond is a billionth of a second). Each pulse is high for about 45nanoseconds, with a rising time of 12 ns and a falling time of 12 ns (note that my graph is not drawn to scale).

Pulses in the next phase are 83 ns behind those in the first phase and there is a 5 ns lag time between the end of one pulse and the start of the corresponding pulse on the next phase. At least, that's what the data manual says, but if you add up durations: 12+45+12+5 you get 74 ns, not 83 ns! So were are the missing 9 nanoseconds? Your guess is mine...

Note that it is possible to crank up clock speed, upto 4 Mhz at least, without risking to fry the TMS9900. Such modifications the the TI-99/4A have been described (including in those pages), and are known to work. Of course any process that relies on execution speed to time an external device (such as disk access) will be messed up...

Internal memory bus timing

Below are some timing diagrams for the memory bus of the TMS9900. Note that these will be different in the PE-Box, due to the multiplexing of the data bus.

Read cycle (no wait state)

```
_|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____ Phi 1
___|¯|_____|a|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|___ Phi 2
_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_ Phi 3
_______|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯ Phi 4
____________                 ____________________________
            \_______________/                            MEMEN*
             _______________
____________/               \____________________________ DBIN
            d              d
_________________________________________________________ WE*

XXXXXXXXXXXX/ Valid address \XXXXXXXXXXXXXXXXXXXXXXXXXXXX A0-A14

                ______
XXXXXXXXXXXXXXX/ ^b   \XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX READY
               e      e
_________________________________________________________ WAIT

CPU driven  | Input mode |r| Input ^c  | CPU driven      D0-D15
             ________________
____________/ if instruction \___________________________ IAQ
```

Notes

a) The cycle begins and ends on the rising edge of Phi 2 pulses.
b) Inputs should be ready at least 30 ns before the rising edge of the next Phi 1 pulse.
c) Inputs should remain valid for at least 10 ns after the falling edge of the Phi 1 pulse.
d) Propagation delays are at most 30 ns for MEMEN*, DBIN, WE* and WAIT.
e) Propagation delays for all other outputs are at most 40 ns.
r) Read data

Write cycle (1 wait state)

```
_|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____ Phi 1
___|¯|_____|a|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|___ Phi 2
_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_ Phi 3
_______|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯ Phi 4
____________                         ____________________
            \_______________________/                    MEMEN*
_________________________________________________________ DBIN
___________________           ___________________________
                   \_________/                           WE*
XXXXXXXXXXXXXX/  Valid  address     \XXXXXXXXXXXXXXXXXXXX A0-A14
                          ___
XXXXXXXXXXXXXXXXX\^b___/ d^\XXXXXXXXXXXXXXXXXXXXXXXXXXXXX READY
                        _______
______________________/c       \_________________________ WAIT

CPU driven | CPU write data | CPU driven D0-D15
_________________________________________________________ IAQ
```

Notes

a) The cycle begins with the rising edge of Phi2.

b) The READY line is tested on the rising edge of the next Phi 1. It should be high at least 40 ns before that time..
c) If it's low, the TMS9900 enters a wait state and
d) retest the READY line at each Phi 1 pulse, until it is high again.

CRU output (2 bits)

```
_|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____ Phi 1

___|¯|_____|a|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|___ Phi 2

_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_ Phi 3

_______|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯ Phi 4

XXXXXXXXXXXX/ Bit 1 address | Bit 2 address \XXXXXXXXXXXX A0-A14
                    ¯¯¯             ¯¯¯
___________________|b  |___________|c  |_________________ CRUCLK
                   d   d           d   d
XXXXXXXXXXXX/ Bit 1 value   | Bit 2 value   \XXXXXXXXXXXX CRUOUT

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX CRUIN
```

<u>Notes</u>
a) The cycle begins on the rising edge of a Phi 2 pulse.
b) A CRUCLK pulse is issued at the next Phi 2 pulse, until the end of the Phi 3 pulse.
c) A similar CRUCLK pulses is issued for each following bit.
d) Propagation delays are at most 30 ns for CRUCLK (40 ns for other outputs).

CRU input (2 bits)

```
_|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____ Phi 1

___|¯|_____|a|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|___ Phi 2

_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_ Phi 3

_______|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯|_____|¯ Phi 4

XXXXXXXXXXXX/ Bit 1 address | Bit 2 address \XXXXXXXXXXXX A0-A14

_________________________________________________________ CRUCLK

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX CRUOUT
                        ¯¯¯¯            ¯¯¯¯
XXXXXXXXXXXXXXXXXXXXXXX/ ^b \XXXXXXXXXX/ ^c \XXXXXXXXXXXX CRUIN
```

<u>Notes</u>
a) The cycle begins on the rising edge of a Phi 2 pulse.
b) The CRUIN line is sampled at the rising edge of the second Phi 1 pulse following the Phi 2 pulse.
c) Following bits are sampled on the rising edge of every second Phi 1 pulse.
d) No CRUCLK is generated during CRU input.

Electrical characteristics

Recommended operating conditions

Parameter	Min	Nom	Max	Unit
Vbb	5.25	-5	-4.75	Volts
Vcc	4.75	5	5.25	Volts
Vdd	11.4	12	12.6	Volts
Vss	-	0	-	Volts
High level input	2.2	2.4	Vcc+1	Volts
Ditto for clock	Vdd-2	-	Vdd	Volts
Low level input	-1.0	0.4	0.8	Volts
Ditto for clocks	-0.3	0.3	0.6	Volts
Free-air temperature	0	25	70	`C

Electrical characteristics under recommended conditions

Parameter	Test conditions	Min	Nom	Max	Unit
Data bus input current	Vss to Vcc	-	50	100	uAmp
Clock input current	-0.3V to 12.6V	-	25	75	uAmp
Other pins input current	Vss to Vcc	-	1	10	uAmp
High level output voltage	-0.4 mAmp	2.4	-	Vcc	Volts
Low level output voltage	3.2 mAmp	-	-	0.65	Volts
	2.0 mAmp	-	-	0.50	Volts
Supply current from Vbb	-	-	0.1	1	mAmp
Supply current from Vcc	-	-	50	75	mAmp
Supply current from Vdd	-	-	25	45	mAmp
Data bus capacitance	Vbb=-5 f=1Mhz	-	15	25	pF
Clock 1 capacitance	Vbb=5 f=1Mhz	-	100	150	pF
Clock 2 capacitance	Vbb=5 f=1Mhz	-	150	200	pF
Clock 3 capacitance	Vbb=5 f=1Mhz	-	100	150	pF
Clock 4 capacitance	Vbb=5 f=1Mhz	-	100	150	pF
Other input capacitance	Vbb=5 f=1 MHz	-	10	15	pF

Revision 1. 3/25/99 Preliminary, but ok to release
Revision 2. 5/31/99 Tested & debugged examples
Revision 3. 1/4/02 Modified speed calculation examples
Revision 4. 1/15/06 Added example of external instruction use (Cortex computer)

Back to the TI-99/4A Tech Pages