Report

# A 12-core processor implementation on FPGA

**Author(s):**
Liu, Ling

**Publication Date:**
2009

**Permanent Link:**

https://doi.org/10.3929/ethz-a-006819824 →

# A 12-Core Processor implementation on FPGA

*Computer systems institute, ETH Zürich, Switzerland*

*ling.liu@inf.ethz.ch*

*18 September, 2009, rev. 5 October, 2009*

## Introduction

The design of this 12-core processor stems from Chuck Thacker, redesigned by Prof. Niklaus Wirth and partly implemented by Ling Liu on Xilinx ML505 evaluation platform[1]. In this processor design, 12 simple RISC processor cores are organized as 4 groups. Each group includes three cores and communicates with the other cores via 4 buses. Figure 1 shows the schematic of the 12-core processor.

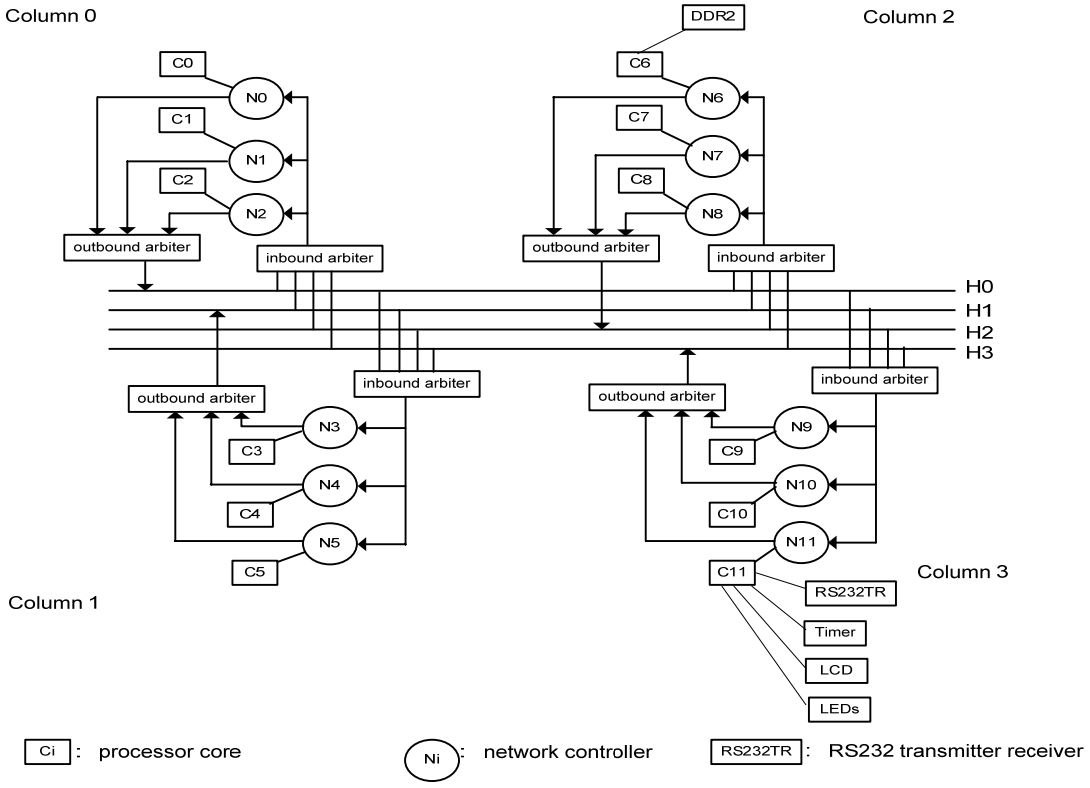


Figure 1: The schematic of 12-core processor

[1] Xilinx ML505 evaluation platform uses Virtex-5 LX50T-1 FPGA chip.

At the center of the columns, there are four horizontal busses, each of which is 8-bit wide and driven from a single point. This point is the outbound wiring from each column. The output of each core is connected to the outbound arbiter via a network controller, and in the end to the horizontal bus owned by its column. The message packet is driven onto the bus and captured by the inbound arbiter, and stored into the message buffer of a network controller. Each processor core can access the message buffer in the connected network controller (called netNode). Each netNode has one 32 * 4 byte input message buffer and one 32*4 byte output message buffer. The details of the message network design are illustrated in [1]. This document records the implementation of the processor core, the connection between the processor core and the network controller , DDR2 controller and I/O device controllers (RS232 Transmitter/Receiver, Timer, 2-line LCD, and 8 LEDs).

## The Tiny Register Machine (TRM)

Each processor core is a simple register processor, which is called "Tiny Register Machine (TRM) [2]. Each TRM contains an arithmetic-logic unit (ALU) and a shifter. The 32-bit operands and results are stored in a bank of 16 registers. The local data memory consists of 2048 words of 32 bits. The local program memory consists of 4096 instructions with 18 bits. Each TRM also has a register called H for storing the high 32 bits of the product, and 4 conditional registers C, N, V, Z.

### Instruction Summary

Registers: R0 .. R15, PC, H, Cond (Z, N, C, V).

Instruction fields op, d, b, a. Literals with zero extension.
R.i denotes register i. x stands for “don’t care”

| | | | | | |
|---|---|---|---|---|---|
| AND | 0000 | dddd | bbbb | 0nnnnn | R.d := R.b & n |
| AND | 0000 | dddd | bbbb | 1xaaaa | R.d := R.b & R.a |
| BIC | 0001 | dddd | bbbb | 0nnnnn | R.d := R.b & ~n |
| BIC | 0001 | dddd | bbbb | 1xaaaa | R.d := R.b & ~R.a |
| OR | 0010 | dddd | bbbb | 0nnnnn | R.d := R.b OR n |
| OR | 0010 | dddd | bbbb | 1xaaaa | R.d := R.b OR R.a |
| XOR | 0011 | dddd | bbbb | 0nnnnn | R.d := R.b XOR n |
| XOR | 0011 | dddd | bbbb | 1xaaaa | R.d := R.b XOR R.a |
| | | | | | |
| ADD | 0100 | dddd | bbbb | 0nnnnn | R.d := R.b + n |
| ADD | 0100 | dddd | bbbb | 1xaaaa | R.d := R.b + R.a |
| SUB | 0101 | dddd | bbbb | 0nnnnn | R.d := R.b - n |
| SUB | 0101 | dddd | bbbb | 1xaaaa | R.d := R.b - R.a |
| MUL | 0110 | dddd | bbbb | 0nnnnn | R.d := R.b * n |
| MUL | 0110 | dddd | bbbb | 1xaaaa | R.d := R.b * R.a |

```
NOT   0111  dddd  xxxx   0nnnnn    R.d := -n
NOT   0111  dddd  xxxx   10aaaa    R.d := -R.a
LDH   0111  dddd  xxxx   110000    R.d := H
LDC   0111  dddd  xxxx   110001    R.d := C

LIT   1000  dddd  nnnn   nnnnnn    R.d := n  (10-bit zero extended literal)
ROR   1001  dddd  bbbb   0nnnnn    R.d := R.b ROR n
ROR   1001  dddd  bbbb   1xaaaa    R.d := R.b ROR R.a
BLR   1010  xxxx  xxxx   1xaaaa    R.15 := PC+1; PC := R.a
BR    1011  xxxx  xxxx   10aaaa    PC := R.a (R.a in register bank 0)
BR    1011  xxxx  xxxx   0xxxxx    RegBank = IR[1]; GlobIntEnb = IR[0]
BR    1011  xxxx  xxxx   11aaaa    PC := R.a (R.a in register bank 1); Return from interrupt
handler


LD    1100  dddd  bbbb   0nnnnn    R.d := M[R.b + n]
LD    1100  dddd  bbbb   1xaaaa    R.d := M[R.b + R.a]
ST    1101  dddd  bbbb   0nnnnn    M[R.b + n] := R.d
ST    1101  dddd  bbbb   1xaaaa    M[R.b + R.a] := R.d
Bc    1110  cccc  nnnn   nnnnnn    PC := PC + 1 + n, on condition c
BL    1111  nnnn  nnnn   nnnnnn    R.15 := PC+1; PC := PC + 1 + n
```

## Condition Codes

Apart from multiplication instructions, N and Z are set by all register instructions that write some result to a register. The ROR instruction sets C equal to bit 0 of the shifted result.

```
N  =  bit 31 of result
Z  =  all 32 bits are zero
C  =  carry
V  =  overflow
```

**Bc** instructions contain 4 condition field. Its value determines which condition is tested. S stands for N xor V.

```
0000   EQ    Z          1000  HI  ~(~C | Z)
0001   NEQ  ~Z          1001  LS  ~C | Z

0010   C                1010  GEQ   ~S

0011   ~C               1011  LSS    S

0100   N                1100  GTR   ~(S | Z)

0101   ~N               1101  LEQ   S | Z
```

0110 V 1110 true (jump always)

0111 ~V 1111 false (no jump)

## Instruction execution

Figure 2 is a block diagram of the TRM. TRM is currently implemented as 2-stage pipelined machine, which is running at 116MHz (frequency of clk clock). At the rising edge of the clk clock, two instructions (36 bit pmout) are fetched from the instruction memory. Then the instruction (IR) under execution is selected from *pmout* based on bit 0 of *PC*.

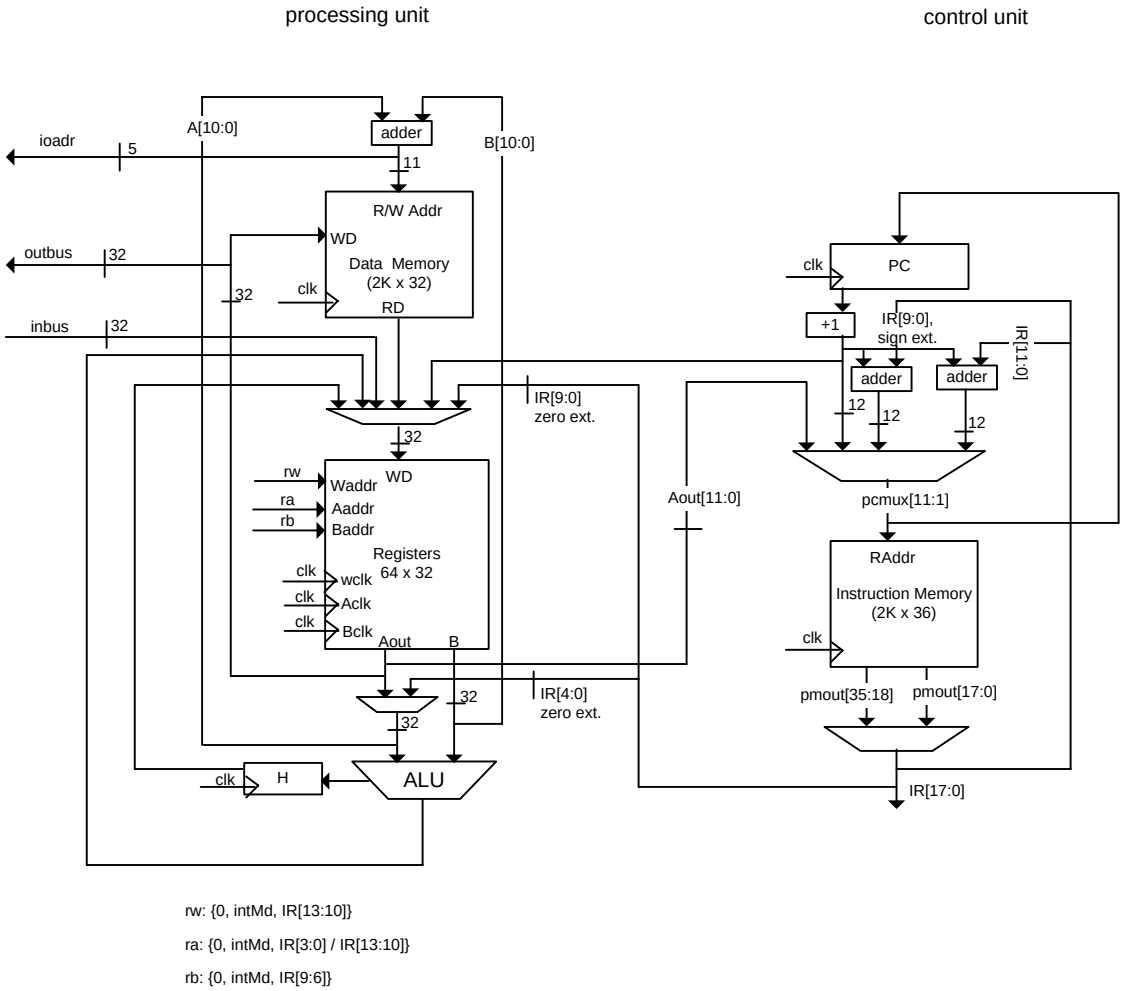


Figure 2: TRM Block Diagram

IR is decoded to get ra, rb and rw that are used to read the register file and get two 32 bit data items Aout and B. According to the *op* field in *IR*, Aout or 5-bit zero extended literal in IR is used as one input A to ALU. The ALU computes a result by combing A and B according to the *op* field of *IR*. The result of ALU is then written back into register *rw*, and H if the instruction is multiplication. The multiplication is implemented by a *MUL unit* that contains 4 DSPs. **The**

**multiplication will cause the processor to stall for 4 clk clocks. The LD instruction will cause the processor to stall one clk cycle**.

## Interface to network and I/O device controllers

Each TRM processor is connected to a network controller called *netNode*. TRM processor core 11 is also connected to a RS232 controller (RS232 transmitter receiver), a 2-line LCD controller, a Timer, 8 LEDs. TRM processor core 6 is also connected to 512MB DDR2 controller. Each netNode or the RS232 controller is treated as an I/O port to the TRM processor, and communicates with the corresponding TRM core through a 32-bit on-chip I/O bus.

Two I/O memory addresses are allocated for each port, one each for the Data Register and Command Register. Table 1 gives the I/O memory address for each port.

Table 1: The I/O memory addresses

| I/O memory address | Port |
|---|---|
| F02 | netNode data register |
| F03 | netNode command register |
| F04 | RS232 data register |
| F05 | RS232 command register |
| F06 | Timer data register |
| F07 | Timer command register |
| F08 | LCD data register |
| F09 | LCD command register |
| F0A | DDR2 data register |
| F0B | DDR2 command register |
| F0C | LED data register |
| F0D | reserved |
| F0E | Switch input (8-bit input, Switch 1 is MSB) |

The bit locations in the command registers are explained in the following paragraphs.

**netNode command register**

**Bit 0**: Input message buf of the netNode is taken or a message is received.

**Bit 1:** output message buf of the netNode is taken or a message is ready for sending.

**Bit 2:** Clear output / input message buf address register.

**Bit 3:** reserved.

**Bit 4:** Enable input message buf taken / message received interrupt.

**Bit 5:** Enable output message buf empty interrupt.

**Bit 6 ~ 7:** Reserved.

**Bit 8 ~ Bit 15:** Id of the netNode.

**RS232 command register**

**Bit 0:** One byte data is received from RS232 port and ready to be read.

**Bit 1**: RS232 transmitter is ready.

**Bit 2:** Clear the data buf of RS232 controller.

**Bit 3:** Enable RS232 receiving interrupt.

**Timer command register**

**Bit 0:** Clear the clock counter

**LCD command register**

**Bit 0:** LCD_RW, selects read or write.

**Bit 1:** LCD_RS, selects registers.

**Bit 2:** LCD_E, starts data read / write

**DDR2 command register**

**Bit 0 ~ Bit 22:** Block (8 words) address in DDR2 memory.

**Bit 23:** Read (1) / Write (0).

**Bit 24:** reservered.

**Bit 25 ~ Bit 27:** reserved.

**Bit 28:** Command buffer is ready, that is, command buffer is not full.

**Bit 29:** Write data buffer is ready, that is write data buffer is not full.

**Bit 30:** A block of data (8 words) has be read out of DDR2 memory and ready in the read data buffer.

**Bit 31:** Calibration fail.

## Examples regarding the interface usage

### 1. Message passing example

**In this example, TRM core 11 sends two byte message 0809H to TRM core 0.**

**Oberon code on core 11:**

```
MODULE Sender;
   CONST MSGadr = 0F02H;
   VAR
       s: SET; header: INTEGER;
BEGIN
   REPEAT UNTIL ~BIT(MSGadr+1, 1); (* repeat until output msg buf is empty*)
   GET(MSGadr+1, s); (* read netNode command register*)
   s := s + {2};
   PUT(MSGadr +1, s); (*clear input/output msg buf address register in netNode*)
   header := ROR(ROR(ROR(ROR(0, 8) + 11, 8) + 2, 8) + 0, 8); (*construct 4 byte msg header: type(0),
                                                              len(2), source(11), dst(0)*)
   PUT(MSGadr, header); (*copy header to the msg output buf, buf address register increases by 1*)
   PUT(MSGadr, 0809H); (*write two byte message 0809H to msg output buf*)
   GET(MSGadr+1, s);
   s := s+ {1};
   PUT(MSGadr+1, s) (*set bit 1 of netNode command register to inform  netNode a message is
                     ready*)
END Sender.
```

**TRM instructions on core 11:**

```
(*instruction memory starts here*)
s    1  (* address of global variable s*)
header    2  (*address of global variable header*)
   0 0003C00F (*BL 15*)
  16 00027BD5   ROR    R14    R15  21
  17 00030340   LD     R0  R13   0
  18 00010001   ADD    R0  R0   1
  19 00030400   LD     R1  R0   0
  20 00024441   ROR    R1  R1   1
  21 00038BFB   BCS    -5
  22 00030340   LD     R0  R13   0
  23 00010001   ADD    R0  R0   1
  24 00030000   LD     R0  R0   0
  25 00034341   ST  R0 R13   1
  26 00030341   LD     R0  R13   1
  27 00020404   LIT    R1   4
  28 00008021   OR     R0  R0  R1
  29 00034341   ST  R0 R13   1
  30 00030340   LD     R0  R13   0
  31 00010001   ADD    R0  R0   1
```

```
  32 00030741   LD      R1  R13   1
  33 00034400   ST  R1  R0   0
  34 00020000   LIT     R0    0
  35 00024008   ROR     R0  R0   8
  36 0001000B   ADD     R0  R0  11
  37 00024008   ROR     R0  R0   8
  38 00010001   ADD     R0  R0   1
  39 00024008   ROR     R0  R0   8
  40 00010000   ADD     R0  R0   0
  41 00024008   ROR     R0  R0   8
  42 00034342   ST  R0  R13   2
  43 00030340   LD      R0  R13   0
  44 00030742   LD      R1  R13   2
  45 00034400   ST  R1  R0   0
  46 00030340   LD      R0  R13   0
  47 00030743   LD      R1  R13   3
  48 00034400   ST  R1  R0   0
  49 00030340   LD      R0  R13   0
  50 00010001   ADD     R0  R0   1
  51 00030000   LD      R0  R0   0
  52 00034341   ST  R0  R13   1
  53 00030341   LD      R0  R13   1
  54 00020402   LIT     R1    2
  55 00008021   OR      R0  R0  R1
  56 00034341   ST  R0  R13   1
  57 00030340   LD      R0  R13   0
  58 00010001   ADD     R0  R0   1
  59 00030741   LD      R1  R13   1
  60 00034400   ST  R1  R0   0
(*data memory starts here*)
data
   0 00000F02       3842
   3 00000809       2057
```

**2. RS232 input /output example**

In this example, core 11 reads a character from PC through RS232 port and transmits it back to PC through RS232 port.

**Oberon code on core 11:**

```
MODULE RS232;
   CONST RSadr= 0F04H;
   VAR
    s: SET; ch: CHAR;
BEGIN
   REPEAT  UNTIL BIT(RSadr+1, 0);  (*repeat until one byte data is ready on RS232 port*)
   GET(RSadr, ch); (*copy received data to ch*)
   GET(RSadr+1, s);
   s := s + {2};
```

```
    PUT(RSadr+1, s); (*set bit 2 of RS232 command register to clear data buf*)
    REPEAT UNTIL BIT(RSadr+1, 1); (*wait until RS232 transmitter is ready*)
    PUT(RSadr, ch) (*Write data to RS232 port*)
END RS232.
```

**TRM instructions on core 11:**

```
(*instruction memory starts here*)
s    1 (*address of global variable s*)
ch   2 (*address of global variable ch*)
   0 0003C00F (*BL 15*)
  16 00027BD5   ROR    R14    R15  21
  17 00030340   LD     R0  R13   0
  18 00010001   ADD    R0  R0    1
  19 00030400   LD     R1  R0    0
  20 00024440   ROR    R1  R1    0
  21 00038FFB   BCC    -5
  22 00030340   LD     R0  R13   0
  23 00030000   LD     R0  R0    0
  24 00034342   ST  R0 R13   2
  25 00030340   LD     R0  R13   0
  26 00010001   ADD    R0  R0    1
  27 00030000   LD     R0  R0    0
  28 00034341   ST  R0 R13   1
  29 00030341   LD     R0  R13   1
  30 00020404   LIT    R1    4
  31 00008021   OR     R0  R0  R1
  32 00034341   ST  R0 R13   1
  33 00030340   LD     R0  R13   0
  34 00010001   ADD    R0  R0    1
  35 00030741   LD     R1  R13   1
  36 00034400   ST  R1 R0    0
  37 00030340   LD     R0  R13   0
  38 00010001   ADD    R0  R0    1
  39 00030400   LD     R1  R0    0
  40 00024441   ROR    R1  R1    1
  41 00038FFB   BCC    -5
  42 00030340   LD     R0  R13   0
  43 00030742   LD     R1  R13   2
  44 00034400   ST  R1 R0    0
(*data memory starts here*)
data
   0 00000F04       3844
```

## Implementation of the interface between TRM and netNode and I/O Controllers

Bit 11 to Bit 8 in memory address signal are used to enable/disable I/O port access. When these four bits are equal to 1111, then current instruction is accessing I/O port, which is either

netNode, DDR2 or I/O devices (RS232 port, LCD, Timer or LEDs). If the instruction is LD then it will read the data from inbus; if the instruction is ST then it will write the data onto outbus (see Figure 2 for the connection of inbus, outbus and ioadr). Signal ioadr comes from the least five bits of the memory address signal. Figure 3 gives the interface between core 6 and netNode and DDR2 controller. Core 11 has similar interface with more I/O devices. Other cores only connect to netNodes.

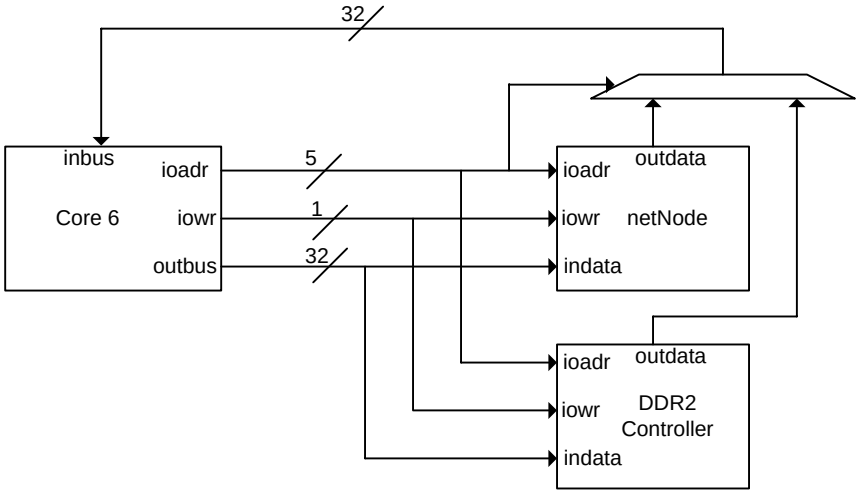


Figure 3: The interface between TRM core 6 and netNode and DDR2 controller

## Interrupt handling

Currently, TRM provides two interrupt sources. One is message received interrupt. Once the input message buf of the connected netNode is taken, that is, **a message is received**, this interrupt will be raised. The other is output message buf ready interrupt. Once **a message output buf of the connected netNode is empty**, this interrupt will be raised. All interrupts are assigned individual enable bits in the netNode command register. To enable interrupts, these bits and the Global Interrupt Enable register must be written logic one.

The lowest 16 addresses in the program memory space are by default defined as the interrupt vectors. The list of the interrupt vectors is shown in Table 2. The list also determines the priority levels of the different interrupts. The lower the address the higher is the priority level.

When an interrupt occurs while the corresponding interrupt enable bit is set, the Global Interrupt Enable register is cleared and all interrupts are disabled. Nested interrupts are not allowed. The Global Interrupt Enable register is automatically set when TRM processor returns from the interrupt handling routine. When the TRM returns from an interrupt handler, it always returns to the instruction that is pended when the interrupt is served.

Table 2: Interrupt vectors

| Vector | Programm Address | Source | Interrupt Definition |
|---|---|---|---|

| No. | | | |
|---|---|---|---|
| 0 | 0000H | undefined | |
| 1 | 0001H | undefined | |
| 2 | 0002H | Msg Input buf | a message is received |
| 3 | 0003H | Msg output buf | input message buf is empty |
| 4 | 0004H | RS232 receiver | one character is received via RS232 port |
| ... | ... | undefined | |
| 15 | 000FH | undefined | |

## *Implementation details of interrupt handling*

The register file in TRM is implemented by 64 deep LUT RAM, which is divided into 4 banks. Each bank has 16 registers. The main process uses the registers in bank 0, the interrupt handling routine uses the registers in bank 1. The bank number (0 or 1) is stored into 1 bit **intMd** register that is used with ra, rb, and rw to address a register. When an interrupt is raised, TRM processor takes one clk cycle to set the intMd register. **Thus the minimal interrupt response time (from the time an interrupt is raised to the time that the interrupt handler is executed) is one clk cycle**. A return from the interrupt handling routine takes one clk cycle.

Because the interrupt handling routine uses different bank of registers from the main routine, thus program does not need to store the working registers. In addition, when an interrupt handling routine is entered, the conditional registers (N, Z, C, V) are also automatically stored into the most significant 4 bits of register 15 in bank 1. But register H is not saved. Therefore, the programmer or the compiler has to save the content of register H before entering interrupt routine if necessary.

Following oberon code and TRM instructions are used in interrupt handling.

1. Enable / disable global interrupt

```
(*Oberon code*)
    SETPSR(0); (*set current register bank number as 0 and disable global interrupt*)
    SETPSR(1); (*enable global interrupt*)

(*TRM instructions*)
    0002C000    BR      R0  R0  0
    0002C001    BR      R0  R0  1 (*op = BR & ~IR[5], bit 0: enable global interrupt, bit1: register bank
    number *)
```

2. Return from interrupt handling routine

```
(*TRM instructions*)
    0002C03F BR      R0  R0  R15 (*op = BR & IR[5] & IR[4]*)
```

3. Eanble / disable single interrupt source

```
(*Oberon code*)
    GET(MSGadr+1, msgState);
    msgState := msgState + {4, 5};
    PUT(MSGadr+1, msgState); (*enable output msg buf empty /message received  interrupts*)
    msgState := msgState * {0, 1, 2};
    PUT(MSGadr+1, msgState); (*disable output msg buf empty / message received  interrupts*)


(*TRM instructions*)
  00030340   LD      R0  R13   0
  00010001   ADD     R0  R0   1
  00030000   LD      R0  R0   0
  00034341   ST      R0  R13   1
  00030341   LD      R0  R13   1
  00020430   LIT     R1   48
  00008021   OR      R0  R0  R1
  00034341   ST      R0  R13   1
  00030340   LD      R0  R13   0
  00010001   ADD     R0  R0   1
  00030741   LD      R1  R13   1
  00034400   ST      R1  R0   0
  00030341   LD      R0  R13   1
  00020407   LIT     R1    7
  00000021   AND     R0  R0  R1
  00034341   ST      R0  R13   1
  00030340   LD      R0  R13   0
  00010001   ADD     R0  R0   1
  00030741   LD      R1  R13   1
  00034400   ST      R1  R0   0

(*data memory starts here*)
data
   0 00000F02       3842
```

## Implementation of the interface between DDR2 controller and TRM core

The DDR2 controller is adapted from Chuck's BEE3 DDR2 controller [3, 4] to initialize, calibrate, refresh the 256MB SODIMM on the ML505 board and carry out the read, write requests for the SODIMM. The original DDR2 controller contains three even triggers used to generate timed events such as the transmission of an RS232 character one bit at a time, or refreshing the DDR2 memory, and simple event such as the receipt of an RS232 character.  Now the receiving and transmitting a character is done in polling loops in TRM via the on-chip I/O bus between Core11 and RS232 controller. Therefore, only one event trigger, periodic refreshing the DDR2 memory, is kept in the DDR2 controller.

The DDR2 memory is block wise access. Given an address of a block and a read/write command, the data in the block will be retrieved. One block contains 8 *32-bit words. The DDR2 controller provides three FIFOs - AF, WB and RB, as the interface to user logic. AF is used to store the block address (8 32-bit words) and the command (read/write) to the DDR2 memory. WB is used to store the input data to the memory, RB is used to store the data read out from the memory. Like the interface between TRM core and other I/O device controllers, the interface between the DDR2 controller and the TRM core also consists of one bit signal iowr, 5-bit ioadr, 32-bit input data from the TRM core and 32-bit output data to the TRM core. In the interface, there are two 256 bit buffers used to store the block of data (32 bits) read from or being written to the DDR2 memory. A 3-bit register is used to identify the index of the word in current block.
The TRM core 6 can issue a command to read/write a block of data and reset the word index register and read or write word by word from the block continuously.

## Reference


[1] Lisa (Ling) Liu, A Bus-Based On-Chip Message Passing Network, ETH technical report 645, https://www.inf.ethz.ch/research/disstechreps/techreports/show?serial=645&lang=en, October, 2009

[2] Niklaus Wirth, The Tiny Register Machine (TRM), ETH technical report 643, https://www.inf.ethz.ch/research/disstechreps/techreports/show?serial=643&lang=en, October, 2009.

[3] Chuck Thacker, DDR2 DRAM Controller for BEE3, http://research.microsoft.com/research/downloads/Details/12e67e9a-f130-4fd3-9bbd-f9e448cd6775/Details.aspx.

[4] Lisa (Ling) Liu, A DDR2 SDRAM Interface for Xilinx ML505 Evaluation Platform, ETH technical report 644, https://www.inf.ethz.ch/research/disstechreps/techreports/show?serial=644&lang=en


## Other oberon code examples working on this 12-core processor:

Svend Erik Knudsen wrote a N-Queen solution.

Alessandro Licata Caruso wrote a solution for computing the shortest distance between any pair of nodes in a 11 node graph.