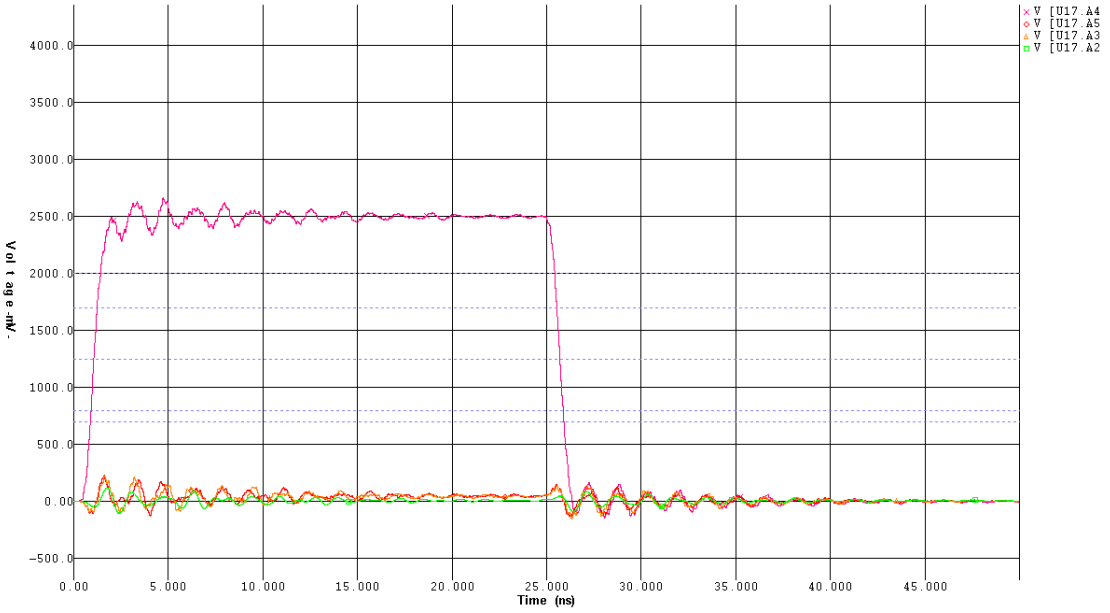

Figure 42. Data bus cross talk

Figure 42 shows the cross talk on the data bus. The FPGA is driving A4 at fast slew rate 8 mA. The pink wave is the applied signal at the RAM chip. The other signals are from the adjacent tracks measured at the RAM chip. The cross talk is low as can be seen from this figure.

The HyperLynx simulation shows no signal integrity problems or any over voltage conditions. It is important to ensure that the over and under shoot of the signals are within the values in Table 17. Violating the specifications will not only damage the component, but will also cause higher power consumption due to the protection diodes conducting. If the maximum current is not exceeded, the device will not be damaged but power consumption and the core temperature of the FPGA will increase thereby increasing the leakage current.

Section 4.1.7 Software

Section 4.1.7.1 Debugger

Debugging software running on an embedded processor is difficult without development tools. Most commercial micro controllers come with some form of a debugging environment. In some cases the environment is free but often one has to buy the environment. When

choosing an embedded processor one of the main contributing factors is the debug environment.

In this research the soft-core implemented on the FPGA came without a debugger thus it was necessary to develop a debugger. Instead of reinventing the wheel the author looked at one of the most popular environments in both the PC and embedded market namely the GNU tools. The soft-core used already makes use of the GCC compiler thus it makes sense looking at GDB.

This section describes the implementation of a minimal GDB server that runs on the soft-core. The debugging session is split into two halves: a server, also called a GDB stub, that runs on the target being debugged and the debugger that runs on a host machine. The goal is to run as little code as possible on the chip. Anything that the remote machine can take care of gets isolated and runs on a remote machine. Symbol information, for example, is non-essential to debugging an application, so the remote machine instead of the stub can manage it. User interface is also a non-essential concern. The remote machine has the luxury of developing a grand user interface, but it makes less sense to have to run this code on the chip.

The LM32 core has an exception base address (EBA) and a debug exception base address (DEBA). When an exception occurs the CPU branches off to an address that is an offset from, either the EBA or the DEBA register. The DEBA register is used for the GDB stub to redirect all debug exceptions to the GDB stub exception handler. In debug mode the break point, instruction bus error, data bus error and watch point exceptions are handled by the GDB stub. All other exceptions are handled by the application being debugged.

Through the GDB stub the PC based debugger can interrogate the CPU by reading and writing to CPU registers and memory locations on the data and instruction bus. Hardware and software breakpoints can be set. The hardware breakpoint implementation is straight forward. Once the CPU receives the command and address for the break point it checks if one of the four hardware breakpoint registers (BP0 – BP3) are available and populates it with the address. The CPU hardware constantly compares the program counter (PC) with the BPx registers and when the PC match an exception is generated in which case the GDB stub exception handler takes over. The software breakpoint works similar except that there are no BPx registers and the CPU places a BREAK instruction at the address where the breakpoint

should occur. A copy is made of the replaced instruction which is restored once the breakpoint is cleared.

```
insn = *(unsigned *)registers[PC];
opcode = insn & 0xfc000000;
if ((opcode == 0xe0000000) ||  // bi    - Unconditional branch instruction
    (opcode == 0xf8000000))    // calli - 4 cyles
{
        branch_step = 1;
        //PC = PC + sign_extend(imm16 << 2)  for bi
        //PC = PC + sign_extend(imm26 << 2)  for calli
        branch_target = registers[PC] + (((signed)insn << 6) >> 4);
}
else if ( (opcode == 0x44000000)  // be            Branch if equal
|| (opcode == 0x48000000)         // bg            Branch if greater
|| (opcode == 0x4c000000)         // bge           Branch if greater or equal
|| (opcode == 0x50000000)         // bgeu          Branch if greater or equal, unsigned
|| (opcode == 0x54000000)         // bgu           Branch if greater, unsigned
|| (opcode == 0x5c000000) )       // bne           Branch if not equal
{
        branch_step = 1;
       //PC = PC + sign_extend(imm16 << 2)
        branch_target = registers[PC] + (((signed)insn << 16) >> 14);       }
else if ( (opcode == 0xd8000000)  // call
|| (opcode == 0xc0000000) )       // b             Unconditional branch
{
    branch_step = 1;
    //Adds 4 to the PC, storing the result in ra, then unconditionally branches to the address in
    //rX (bits 25:21).
    branch_target = registers[(insn >> 21) & 0x1f];
}
else
    branch_step = 0;

/* Set breakpoint after instruction we're stepping */
seq_ptr = (unsigned *)registers[PC];
seq_ptr++;
seq_insn = *seq_ptr;
*seq_ptr = LM32_BREAK;
if (branch_step)
{
        /* Set breakpoint on branch target */
        branch_ptr = (unsigned *)branch_target;
        branch_insn = *branch_ptr;
        *branch_ptr = LM32_BREAK;
}
flush_i_cache ();
```

Table 18. GDB-stub code segment for step command

Implementing single step in the GDB-stub is more involved. Before stepping, the assembler instruction must be consider to determine what affect it will have on the program counter. Table 18 shows the implementation of the step command in C code.

Another useful debugging tool is the integration of the Xilinx JTAG programmer with the software development environment. This was accomplished through batch file processing. The Xilinx development environment contains a command line tool (data2mem) to map software binaries into the compiled bit stream without recompiling the bit stream. The FPGA user constraint file (UCF file) is used to label and specify the location of the affected BRAM regions (see Table 19). In Table 19 rom0, rom1, rom2 and rom3 combined forms a 2K by 32-bit emulated flash. Table 20 shows the configuration file for data2mem. Here we can see exactly how the binary address space maps to the BRAM inside the FPGA.

When the FPGA loads the modified bit stream at start-up the corresponding BRAM (flash emulation) regions are initialised with the binaries. To further speed up the process this new bit stream is immediately dumped through the Xilinx iMPACT load software (configuration file driven) into the SRAM of the FPGA after which the FPGA is reset. From compile time to code execution takes a few seconds and all of this is integrated into the make file of the Eclipse environment.

```
INST "rom0/ram_31_24"  LOC = RAMB16_X0Y15;
INST "rom0/ram_23_16"  LOC = RAMB16_X0Y14;
INST "rom0/ram_15_8"   LOC = RAMB16_X0Y13;
INST "rom0/ram_7_0"    LOC = RAMB16_X0Y12;

INST "rom1/ram_31_24"  LOC = RAMB16_X1Y15;
INST "rom1/ram_23_16"  LOC = RAMB16_X1Y14;
INST "rom1/ram_15_8"   LOC = RAMB16_X1Y13;
INST "rom1/ram_7_0"    LOC = RAMB16_X1Y12;

INST "rom2/ram_31_24"  LOC = RAMB16_X2Y15;
INST "rom2/ram_23_16"  LOC = RAMB16_X2Y14;
INST "rom2/ram_15_8"   LOC = RAMB16_X2Y13;
INST "rom2/ram_7_0"    LOC = RAMB16_X2Y12;

INST "rom_monitor/ram_31_24"  LOC = RAMB16_X0Y11;
INST "rom_monitor/ram_23_16"  LOC = RAMB16_X0Y10;
INST "rom_monitor/ram_15_8"   LOC = RAMB16_X0Y9;
INST "rom_monitor/ram_7_0"    LOC = RAMB16_X0Y8;
```

Table 19. FPGA soft-core emulated flash storage description

```
ADDRESS_SPACE lm32_ROM RAMB16 INDEX_ADDRESSING[0x00000000:0x00007fff]

        BUS_BLOCK
                rom0/ram_31_24 [31:24] PLACED = X0Y15;
                rom0/ram_23_16 [23:16] PLACED = X0Y14;
                rom0/ram_15_8  [15:8]  PLACED = X0Y13;
                rom0/ram_7_0   [7:0]   PLACED = X0Y12;
        END_BUS_BLOCK;

        BUS_BLOCK
                rom1/ram_31_24 [31:24] PLACED = X1Y15;
                rom1/ram_23_16 [23:16] PLACED = X1Y14;
                rom1/ram_15_8  [15:8]  PLACED = X1Y13;
                rom1/ram_7_0   [7:0]   PLACED = X1Y12;
        END_BUS_BLOCK;

        BUS_BLOCK
                rom2/ram_31_24 [31:24] PLACED = X2Y15;
                rom2/ram_23_16 [23:16] PLACED = X2Y14;
                rom2/ram_15_8  [15:8]  PLACED = X2Y13;
                rom2/ram_7_0   [7:0]   PLACED = X2Y12;
        END_BUS_BLOCK;

        BUS_BLOCK
                rom_monitor/ram_31_24 [31:24] PLACED = X0Y11;
                rom_monitor/ram_23_16 [23:16] PLACED = X0Y10;
                rom_monitor/ram_15_8  [15:8]  PLACED = X0Y9;
                rom_monitor/ram_7_0   [7:0]   PLACED = X0Y8;
        END_BUS_BLOCK;

END_ADDRESS_SPACE;
```

Table 20. Configuration file for data2mem application

Section 4.1.7.2 OS support

Subsumption (see Section 2.2) relies heavily on parallel processing. To support this, the system makes use of a real-time multitasking operating system. After looking at various operating systems with a small footprint it was decided to use FreeRTOS. FreeRTOS is a lightweight pre-emptive operating system written in C. The core OS consists of only three C files. The porting to a new platform, like the LM32, is also fairly simple. In essence one has to write the code required for the OS timer tick, interrupt service routine, peripheral initialisation and processor start-up code. For this research, FreeRTOS V5.4.2 was ported to the LM32 processor with the timer tick rate set at 1kHz.

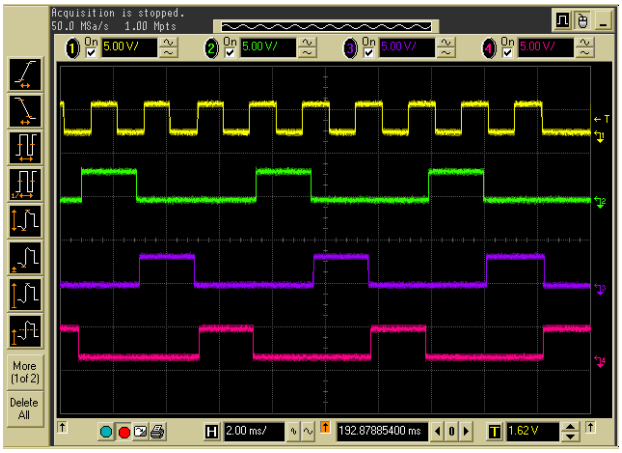

Figure 43. FreeRTOS tasks

To test the ported RTOS a small program was written which contained three tasks. Each task waits for a semaphore and once it is set the task toggles a I/O pin and then waits for the semaphore. When task1 completed one loop, it sets the semaphore for task 2. Task 2 and 3 executes in the same manner. Thus by using semaphores each task is depended on the others. All tasks have the same priority and will executed as follow: task1 -> task2 -> task3 -> task1 …….. Figure 43 shows the process. The yellow wave is generated at 1kHz by the operating system timer tick. The green wave is task1, purple is task 2 and red is task 3. From this we can see that the timer tick is always present and stable and the tasks executes correctly in sequence.

Section 4.2 Power supply

One of the major problems when designing with a FPGA is inrush current. Xilinx claims that the Virtex 4 has no inrush current. Table 21 shows the current consumption at start-up. The values in Table 21 was measured with a power on sequence of V_{CCINT} , V_{CCAUX} and lastly V_{CCO} . If power sequencing is implemented, this is the Xilinx recommended sequence.

It is very difficult estimating the power requirements for a FPGA design as it depends heavily on the HDL design on the FPGA. To overcome this challenge the PSU is designed with reserve capacity and adequate reservoir capacitors are placed on the board. At start-up the PSU is driving the loads as shown in Table 21 and a capacitive load consisting of all the decoupling capacitors on the board. To ensure a stable start-up the power supply contains a