



Laboratory manual for TSEA44

Olle Seger, Per Karlström, Andreas Ehliar, Kent Palmkvist
Computer Engineering
Department of Electrical Engineering
Linköping University, S-581 83 Linköping, Sweden
Email: Kent.Palmkvist@liu.se

October 31, 2022

Contents

1	The system	7
1.1	Introduction	7
1.2	Hardware	7
1.2.1	Virtex-II Development board	7
1.2.2	Communication/Memory Module	7
1.2.3	Virtex-II 4000 FPGA	9
1.3	Open RISC	10
1.3.1	Top Design	10
1.3.2	Structure of the Verilog code	11
1.3.3	OR1200 CPU	13
1.3.4	The Wishbone Interconnect Bus	14
1.3.5	Memory Controller	15
1.3.6	Ethernet Controller	15
1.3.7	VGA Controller	16
1.3.8	UART	16
1.4	Software	16
1.4.1	Memory map	16
1.4.2	A simple boot monitor	17
1.4.3	The simulator <code>or32-uclinux-sim</code>	20
1.4.4	<code>μ</code> Clinux	21
2	Lab task 0 - Build a UART in Verilog	25
2.1	Introduction	25
2.2	A simple UART	25
2.2.1	General	25
2.2.2	The RS232 protocol	26
2.2.3	The zedboard hardware	26
2.2.4	A simple testbench	27
2.3	Exercises	29
2.3.1	Commands	29
2.3.2	A User Constraint File	30
2.4	<code>gtkterm</code> usage	31
3	Lab task 1 - Interfacing to the Wishbone bus	33
3.1	Introduction	33
3.2	Some Basic Facts on the Wishbone Bus	34

3.2.1	A Wishbone Interconnect	35
3.3	A Simple Computer	36
3.3.1	General	36
3.3.2	A Wishbone Interface for the UART	36
3.3.3	The Monitor	38
3.3.4	Test Your Design	39
3.4	A Benchmark Program	40
3.4.1	JPEG Compression	40
3.4.2	Integer DCT	40
3.4.3	The Test Program <code>dct_sw</code>	42
3.4.4	A Test Example	42
3.5	Design a Performance Counter Module	42
3.6	Useful Commands	44
3.6.1	Synthesis Reports	44
3.7	How to get Started Writing/Executing C Programs	45
3.7.1	A Note on Volatile	45
3.7.2	What to Include in the Lab Report	46
4	Lab task 2 - Design a JPEG accelerator	47
4.1	The lab system	47
4.2	Proposed architecture	47
4.2.1	Block RAMs in VirtexII	48
4.2.2	Distributed RAMs	49
4.2.3	The transpose memory	50
4.2.4	WB memory map	50
4.3	Introduction to μ Clinux	51
4.3.1	Compiling an application to μ Clinux	51
4.3.2	Starting the TFTP server	51
4.3.3	Downloading applications via TFTP	52
4.4	Introduction to jpegfiles	52
4.4.1	Important files in the lab skeleton	53
4.4.2	The jpegtest application	54
4.4.3	The webcam application	54
4.5	Timestamps	54
4.6	Quantization	55
4.6.1	General	55
4.6.2	Design of a hardware accelerator for quantization	56
4.7	Tips and tricks	57
4.8	What to include in the lab report	57
5	Lab task 3	59
5.1	DMA in the DCT Accelerator	59
5.1.1	Proposed architecture	59
5.1.2	<code>jpeg_dma.sv</code>	60
5.1.3	How to use DMA in jpegfiles	62
5.1.4	Cache coherency issue	62
5.2	What to Include in the Lab Report	63

6	Lab task 4 - Custom Instructions	65
6.1	Introduction	65
6.1.1	Huffman Coding	65
6.1.2	The Problem	66
6.2	Adding a New Instruction	66
6.2.1	Making the Processor Understand	66
6.2.2	Adding Special Purpose Registers	66
6.2.3	Adding the Required Hardware	67
6.3	Proposed Architecture	67
6.3.1	Control Unit	67
6.3.2	Data Path	68
6.3.3	Store Unit	68
6.3.4	Multi Cycle Instructions	68
6.3.5	Instruction Details	68
6.4	Hardware Implementation	68
6.4.1	Constructing the Hardware	69
6.5	Software Implementation	69
6.5.1	Running the Instruction	69
6.5.2	Integration into jpegfiles	71
6.5.3	JPEG Markers	71
6.6	Important Files For this lab task	72
6.7	Tips and tricks	72
6.8	What to Include in the Lab Report	73
6.9	Beyond tsea44	74
A	Open RISC Reference Platform	77
A.1	Address map	77
A.2	Interrupts	78
B	The Wishbone specification	79
B.1	Introduction	79
B.2	Interface signals	80
B.2.1	adr	80
B.2.2	dat_o and dat_i	80
B.2.3	we	80
B.2.4	sel	80
B.2.5	stb	80
B.2.6	cyc	81
B.2.7	ack	81
B.2.8	cti	81
B.2.9	bte	81
B.2.10	err	81
B.3	Wishbone classical cycles	81
B.4	Wishbone incrementing burst cycles	82
B.5	System Verilog Interface	83
C	Tips & Trix	85

Chapter 1

The system

1.1 Introduction

This text is intended as a laboratory compendium for the course TSEA44 Computer Hardware - a System On a Chip. We begin with a presentation of the hardware and software used for the laboratory exercises. If you wonder, the name *dafk* seen in many places in this course, comes from the name *DAtorteknik FortsättningsKurs* which is the Swedish name of the first version of this course. Roughly translated it means advanced course in computer technology.

1.2 Hardware

The lab tasks are all tested on real hardware. A suitable technology that is flexible enough to allow changes on both software and hardware is FPGA-based development systems. This course is therefore using such a board as the hardware platform on which the computer system is implemented.

1.2.1 Virtex-II Development board

In the course we will use a development FPGA board from Avnet Corporation. A block diagram of this board is shown in Figure 1.1. More details are given in the User's Guide, [1].

1.2.2 Communication/Memory Module

The main FPGA board is extended with a Communication and Memory Module, shown in Figure 1.2. More details are given in the User's Guide, [2].

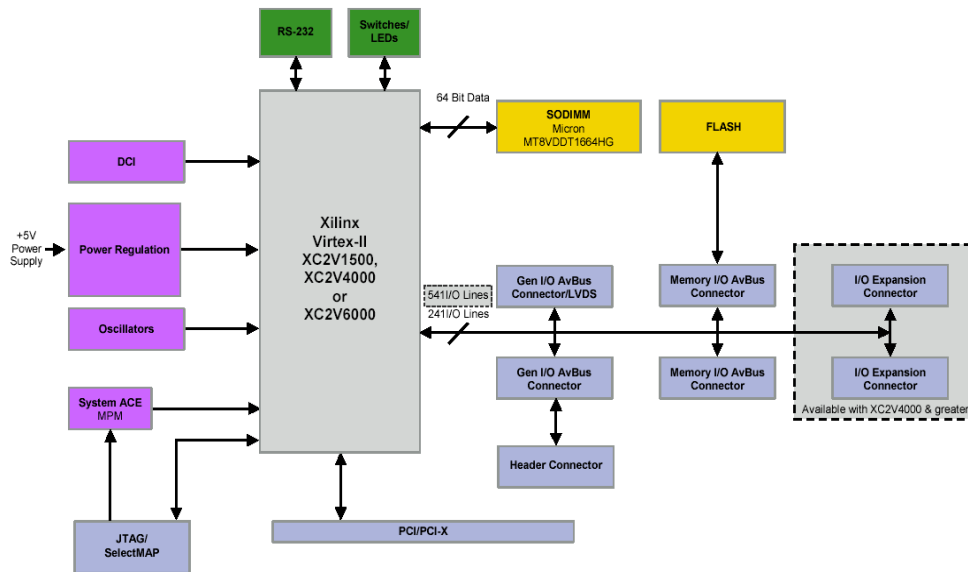


Figure 1.1: Block diagram of the Avnet main board.

Features of the Communication/Memory module are:

- 1 MB SRAM.
- 16 MB Flash memory.
- 64 MB SDRAM.
- 10/(100/1000) Mb/s Ethernet PHY.
- (IrDA for infrared communication).
- (USB2.0 PHY).
- (PC card connector).

Features within parentheses will not be used in this course.

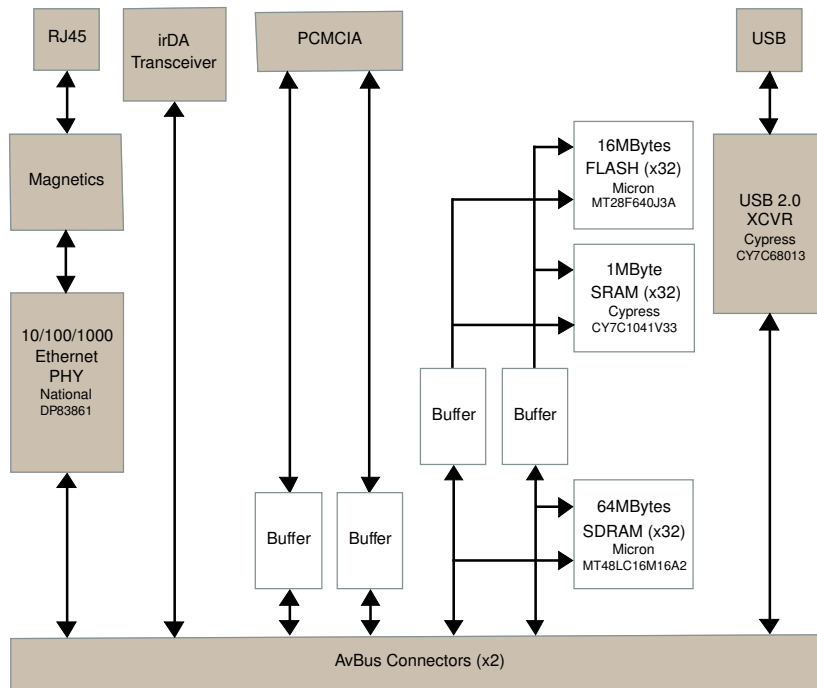


Figure 1.2: Block diagram of the Avnet communication and memory module.

1.2.3 Virtex-II 4000 FPGA

The most important circuit on the board is of course the mighty Virtex-II 4000 FPGA, [5]. The table 1.1 gives some details of this impressive circuit, which is shipped in an 1152 pin BGA (ball grid array).

The internal configurable logic in the FPGA includes four major elements organized in a regular array:

- **CLBs (Configurable Logic Blocks):**
This is the programmable logic used to build combinatorial and sequential logic. The FPGA contains $80 \times 72 = 5760$ CLBs. Each CLB is made up of 4 slices, see Figure 1.4.
- **Multipliers:**
The FPGA contains 120 18×18 -bit multipliers. These are used for the ALU in the OR1200 CPU.
- **Block RAMs:**
The FPGA contains 120 18 kbit RAMs. These are typically used for cache memories inside the CPU, FIFOs in the UART and Ethernet controller.
- **DCMs (Digital Clock Managers):**
The FPGA contains 12 DCMs. The DCMs can divide/multiply the input clock frequency. We use a DCM to transform the input 40 MHz to 25 MHz.

A floorplan of the Virtex-II is shown in Figure 1.3.

The CLBs are organized in an array and connected to a switching matrix. Each CLB comprises 4 slices, which are connected locally. Each slice includes two 4-input

Device	System Gates	CLB (1 CLB = 4 slices = Max 128 bits)			Multiplier Blocks	SelectRAM Blocks		DCMs	Max I/O Pads ⁽¹⁾
		Array Row x Col.	Slices	Maximum Distributed RAM Kbits		18 Kbit Blocks	Max RAM (Kbits)		
XC2V40	40K	8 x 8	256	8	4	4	72	4	88
XC2V80	80K	16 x 8	512	16	8	8	144	4	120
XC2V250	250K	24 x 16	1,536	48	24	24	432	8	200
XC2V500	500K	32 x 24	3,072	96	32	32	576	8	264
XC2V1000	1M	40 x 32	5,120	160	40	40	720	8	432
XC2V1500	1.5M	48 x 40	7,680	240	48	48	864	8	528
XC2V2000	2M	56 x 48	10,752	336	56	56	1,008	8	624
XC2V3000	3M	64 x 56	14,336	448	96	96	1,728	12	720
XC2V4000	4M	80 x 72	23,040	720	120	120	2,160	12	912
XC2V6000	6M	96 x 88	33,792	1,056	144	144	2,592	12	1,104
XC2V8000	8M	112 x 104	46,592	1,456	168	168	3,024	12	1,108

Table 1.1: Virtex-II table. XC2V4000 is our FPGA.

function generators, carry logic, multiplexers and two storage elements, see Figure 1.4. The function generator can be programmed as a 4-input lookup table (LUT), 16-bit RAM or 16-bit variable-tap shift register.

1.3 Open RISC

The computer architecture used in this lab course is based on an open-source processor design called Open RISC. This processor together with its bus system and some I/O units are defined and implemented in the FPGA by synthesis of Verilog code.

1.3.1 Top Design

The computer used in this lab course is designed with Verilog modules, which can be downloaded free from Open Cores (www.opencores.org) and some modules designed by us.

This section describes the main system defined in `dafk.sv`, which you will use in lab task 2–4

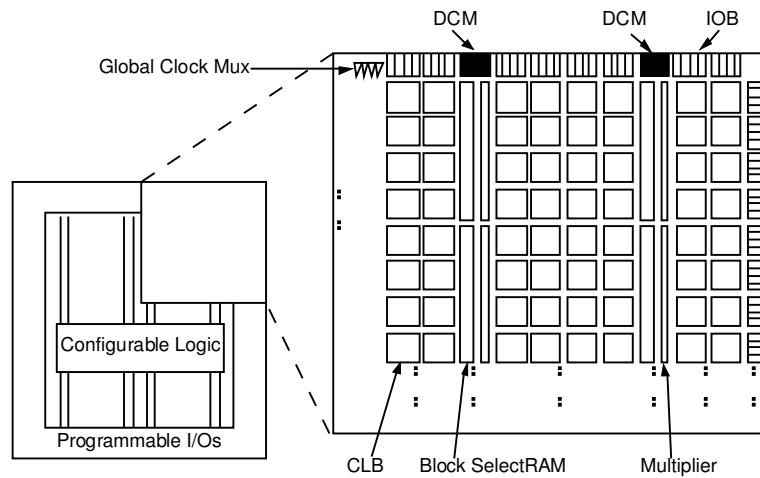


Figure 1.3: Virtex-II architectural overview.

The computer in Figure 1.5 consists of the following modules:

- **OR1200 CPU:**
RISC CPU with a 5-stage pipeline.
- **Wishbone:**
An interconnect bus with 16 ports, 8 master ports and 8 slave ports.
- **Memory Controller:**
A memory controller for SRAM, SDRAM and Flash memories.
- **UART:**
A 16550 UART with baudrates up to 115200 b/s.
- **Ethernet Controller:**
An implementation of the MAC layer, which requires an external PHY circuit for a complete solution.
- **Parallel Port**
- **VGA Controller**
- **Camera Controller**
- **DCT Accelerator:**
It will be your task to finish the implementation of this module.

1.3.2 Structure of the Verilog code

The structure of the Verilog code closely resembles the block diagram shown in Figure 1.5. Components outside the FPGA are simulated. Here is a list of the hierarchy instantiated in `dafk_tb`:

- `phy0`: Simulation model of the Ethernet physical to logic level chip
- `videomem`: Simulation model of the video memory

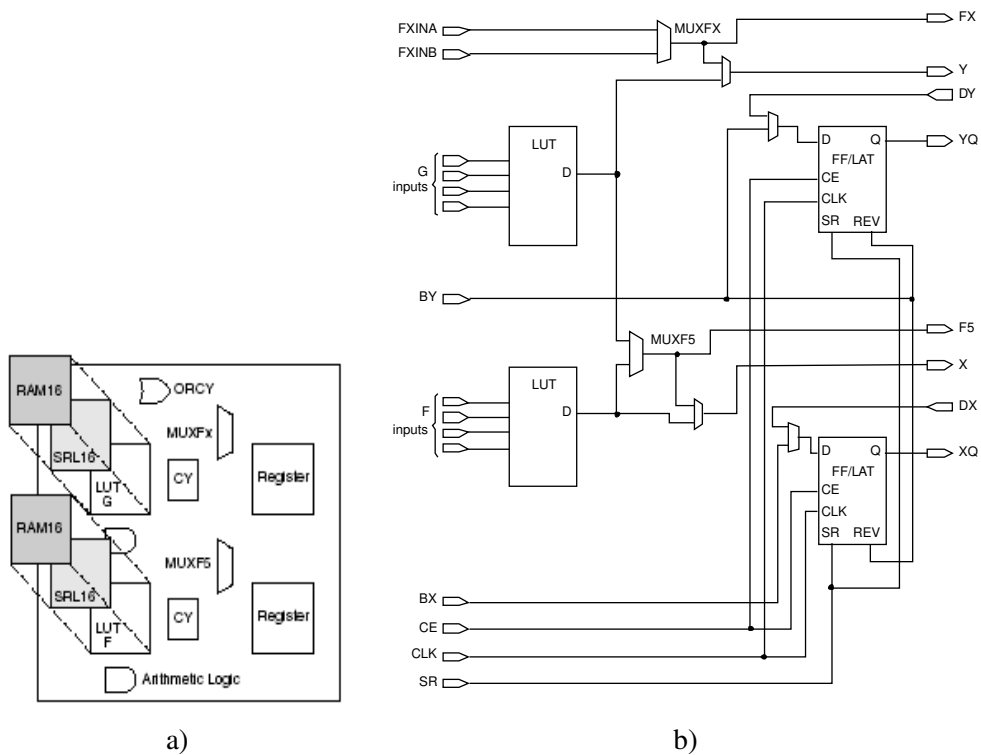


Figure 1.4: a) Virtex-II slice configuration b) Detail of slice (top half).

- mysram: Simulation model of the SRAM
- sdram0: Simulation model of the SDRAM
- dafk_top: The code to be synthesized in the FPGA
 - sys_sig_gen: Generates clock and reset signals
 - or1200_top: The OR1200 CPU
 - pkmc_top: Memory controller
 - rom0: The boot code and vector table resides here
 - uart2: UART 16550
 - eth3: Ethernet controller
 - dvga: VGA controller
 - pia: Simple parallel port
 - jpg0: DCT accelerator
 - perf: Performance counters
 - leela: Camera module
 - wb_conbus: The wishbone bus

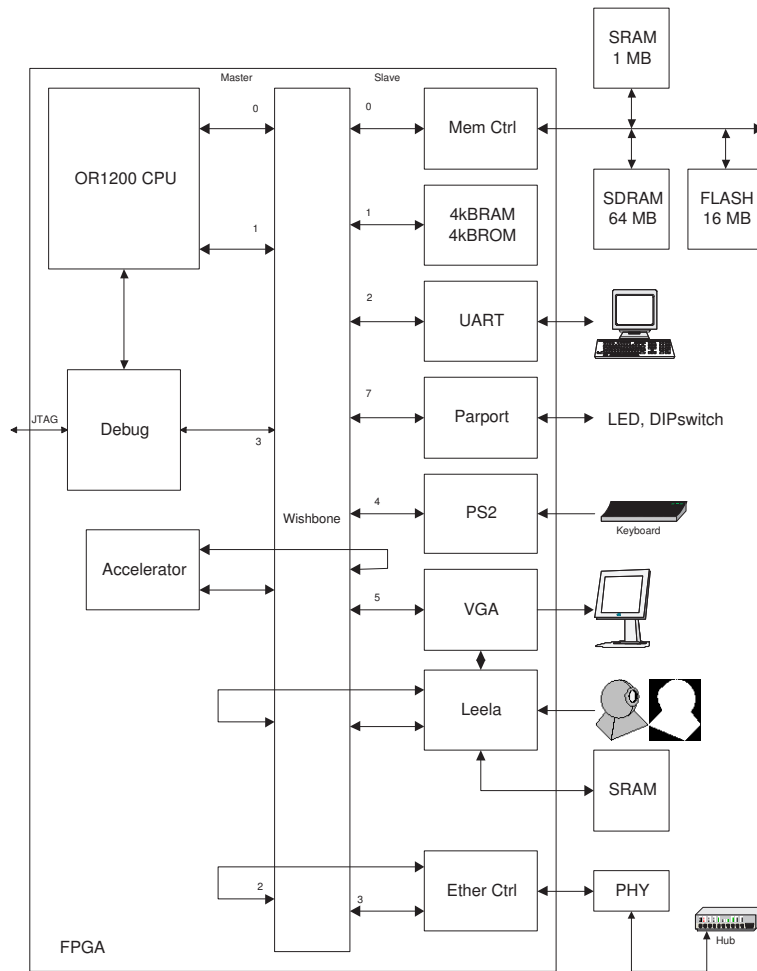


Figure 1.5: An Open RISC computer.

1.3.3 OR1200 CPU

A block diagram of the OR1200 CPU is shown in Figure 1.6. More information about the CPU can be found in [6, 7].

The OR1200 CPU, [8], consists of several blocks:

- **High Performance 32-Bit CPU/DSP**
 - 32-bit architecture implementing ORBIS32 instruction set
 - Scalar, single-issue 5-stage pipeline delivering sustained throughput
 - Single-cycle instruction execution on most instructions
 - Can be run at 250MHz in an ASIC
 - Thirty-two, 32-bit general-purpose registers
 - Custom user instructions
- **L1 Caches**
 - Harvard model with split instruction and data cache

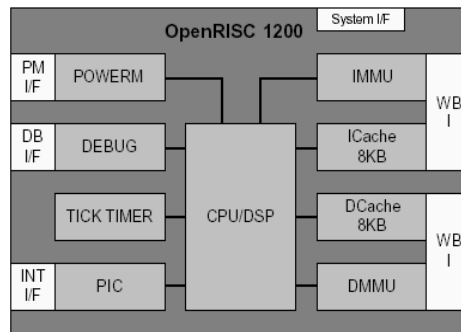


Figure 1.6: Block diagram of the OR1200 CPU.

- Instruction/data cache size scalable from 1KB to 64KB
- **Memory Management Unit**
 - Harvard model with split instruction and data MMU
 - Instruction/data TLB size scalable from 16 to 256 entries
 - Direct-mapped hash-based TLB
 - Linear address space with 32-bit virtual address and physical address from 24 to 32 bits
 - Page size 8KB with per-page attributes
- **Advanced Debug Unit**
 - Conventional target-debug agent with a debug exception handler
 - Non-intrusive debug/trace for both RISC and system
 - Access and control of debug unit from RISC or via development interface
- **Integrated Tick Timer**
 - Task scheduling and precise time measuring
 - Maximum timer range of 2^{32} clock cycles
 - Maskable tick-timer interrupt
- **Programmable Interrupt Controller**
 - 2 non-maskable interrupt sources
 - 30 maskable interrupt sources
 - two interrupt priorities

In this lab course the Power Management module is disabled.

1.3.4 The Wishbone Interconnect Bus

The Wishbone Interconnect is a standard way of connecting IP (Intellectual Property) blocks in System-on-Chip designs, see for instance [9]. It can be implemented in different ways ranging from a fully connected crossbar to an ordinary shared bus. In this course the shared bus variant is used. It is important to understand that there is no

parallelism in this implementation. It is just a connection between one master and one slave. Furthermore tristate is not used, instead there are two databuses, one in each direction. The address bus and the data busses are 32 bits wide.

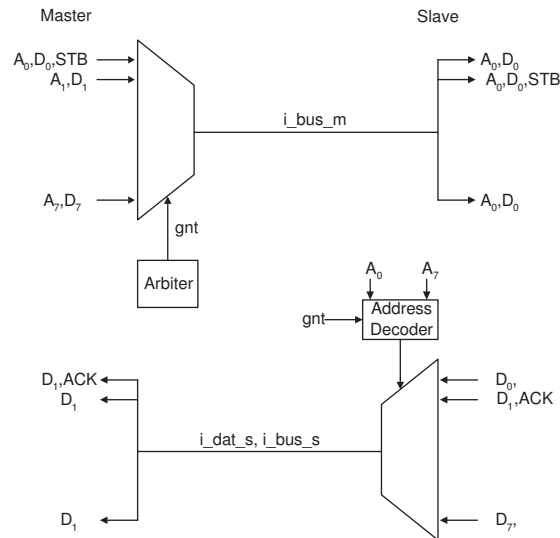


Figure 1.7: The Wishbone interconnect bus. In this example Master 0 is addressing Slave 1. Master 0 has won the arbitration.

We will briefly explain how the Wishbone bus works with a simple example. We assume a computer system like in Figure 1.5 and that the CPU executes a program in the memory, that is connected to slave port 1, see Figure 1.7. The CPU places an address A_0 at the address lines at master port 0 and asserts the signal STB . The arbiter inside the Wishbone grants the bus to master 0. The address A_0 will now show up on all slave ports. Address decoding logic routes the asserted STB -signal only to slave port 1. The memory at slave port 1 places D_1 on the data bus and asserts the signal ACK . D_1 will now show up on all master ports, but ACK will only be asserted at master port 0.

1.3.5 Memory Controller

In this lab course we will use a simple memory controller, designated PKMC, designed by us. PKMC is implemented for this particular system and thus needs no configuration. PKMC handles all communications with the SRAM, SDRAM and FLASH memory. Especially ensuring that the SDRAM is refreshed correctly.

1.3.6 Ethernet Controller

The Ethernet IP Core, [10], consists of five modules:

- The MAC (Media Access Control) module, formed by transmit, receive, and control module
- The MII (Media Independent Interface) Management module

Address	Type	Content
0x0000_0000 - 0x03ff_ffff	SDRAM	Programs can be loaded and run here 64MB
0x2000_0000 - 0x200f_ffff	SRAM	Data area 1MB
0x4000_0000 - 0x4000_5fff	ROM	Boot monitor, 24kB
0x4001_1000 - 0x4001_1fff	RAM	Data area for monitor and stack 8kB
0x9000_0000 - 0x90ff_ffff	UART	
0x9100_0000 - 0x91ff_ffff	Parallel port	
0x9200_0000 - 0x92ff_ffff	Ethernet	
0x9600_0000 - 0x96ff_ffff	Accelerator	Your JPEG accelerator
0xf000_0000 - 0xf0ff_ffff	FLASH	Bender, μ Clinux and Linux

Table 1.2: Memory map of the computer system.

- The Host Interface

The Ethernet IP Core is capable of operating at 10 or 100 Mbps for Ethernet and Fast Ethernet applications. An external PHY is needed for a complete Ethernet solution.

In short the ethernet controller works as follows. There are 64 transmit buffers and 64 receive buffers. These buffers are typically located in the SRAM. To each such buffer there is a pair of registers (a buffer descriptor) inside the Ethernet Controller, one register holds the address of the buffer and one register is a control/status-register. The ethernet controller transmits/receives packets from/to the SRAM buffers with DMA.

1.3.7 VGA Controller

The VGA controller used is designed by us and is a simple single-video-mode controller for use in FPGA or ASIC environments. The VGA controller supports a single resolution/refresh rate in grey scale or 8-bit pseudocolor with 15-bit color sprites. For further details see [12].

1.3.8 UART

The UART (Universal Asynchronous Receiver Transmitter) is an implementation of the industry standard 16550 device. Details can be found in [13].

1.4 Software

An important part of the computer system is the software running on it. How to access hardware units, memories etc. is described in this section.

1.4.1 Memory map

The I/O is memory mapped. The memory addresses for the various I/O units and memory types are show in table 1.2.

Command	Explanation
d <addr>	display memory content
m <addr> <data>	modify memory content
g <addr>	go (execute)
l	load Intel hex file
u	boot uClinux (copy from FLASH)

Table 1.3: Some useful commands in the monitor.

1.4.2 A simple boot monitor

A simple monitor runs in the memory on slave port 1. The monitor will start at boot. Use for instance `gtkterm` and adjust the baud rate to 115200 b/s. Format should be 8N1 and no flow control. The port shall be `/dev/ttyUSB0` or similar. Type `h` for help. Some available commands are explained in table 1.3.

Type `l` and then use the command `File -> Send raw file` in `gtkterm` to load an Intel hex file into memory. The hex file itself contains address information.

A simple program.

In this section we will demonstrate how to compile, load and run a C-program in the monitor environment. We will use the program described in Listing 1.1 as an example.

Listing 1.1: simpleprog

```
#include "common.h"

int main(void)
{
    int Begin_Time, User_Time;
    int i;
    printf("Hello world!\n");

    Begin_Time = get_timer(0);

    for (i=0; i<10; i++) {
        led(i); /* Set the led display on the card */
        printf("%d\n", i);
        sleep(1); /* sleep 1 s */
    }

    User_Time = get_timer(Begin_Time);
    printf("Time=%d s\n", User_Time);

    return (0);
}
```

The program prints a string, counts on the LEDs and measures the elapsed time.

To build `simpleprog` we use a Makefile described in Listing 1.2, in the Makefile we observe the following:

- A cross-compiler, `or32-uclinux-gcc`, must be used.

- The functions `printf`, `get_timer`, `led` and `sleep` are library functions in `openrisclib`, which is included in the lab skeleton for lab 1.
- A link script `ram.ld` is used to determine where in memory our program should be located.

Listing 1.2: Makefile for simpleprog (Listing 1.1)

```

# The name of the program we want to compile
PROGRAM = simpleprog

# The directory containing the open risc supportdir
LIBDIR=../lib
INCLUDEDIR=../include

CFLAGS += -I$(INCLUDEDIR) -Wall -Wstrict-prototypes
CFLAGS += -Werror-implicit-function-declaration
CFLAGS += -Os -g -fno-builtin -fomit-frame-pointer -nostdlib

# Toolchain configuration
AS = or32-uclinux-as
CC = or32-uclinux-gcc
LD = or32-uclinux-ld
DUMP = or32-uclinux-objdump -S -D -EB
COPY = or32-uclinux-objcopy
SIM = or32-uclinux-sim

# Flags to LD, need to include a link script here
LDFLAGS = -Tram.ld

OBJFILES=$(PROGRAM).o

HEXFILE=$(PROGRAM).hex

SIMPROGRAM=$(PROGRAM)sim

all: $(PROGRAM) $(HEXFILE) $(SIMPROGRAM)

# The minimal supportlib containing printf/sleep/etc
openrisclib: $(LIBDIR)/openrisclib.a $(LIBDIR)/crt.o $(LIBDIR)/reset.o

# Commands to make the open risc support lib
$(LIBDIR)/openrisclib.a:
    cd $(LIBDIR) && $(MAKE)

$(LIBDIR)/crt.o:
    cd $(LIBDIR) && $(MAKE)

$(LIBDIR)/reset.o:
    cd $(LIBDIR) && $(MAKE)

.S.o:
    $(CC) $(CFLAGS) -c $<

.c.o:
    $(CC) -c $(CFLAGS) -o $@ $<

# Link the program together with the supportlib
# (And create a text file with the disassembled contents of the program)
$(PROGRAM): $(OBJFILES) ram.ld openrisclib
    $(LD) -Bstatic $(LIBDIR)/crt.o $(OBJFILES) $(LIBDIR)/openrisclib.a \
        $(LDFLAGS) -o $(PROGRAM)
    $(DUMP) $(PROGRAM) > $(PROGRAM).txt

# Create an intel hex dump of the program
$(HEXFILE): $(PROGRAM)
    $(COPY) -O ihex $(PROGRAM) $(HEXFILE)

# Create a binary we can simulate with the openrisc simulator
$(SIMPROGRAM): $(PROGRAM) ram.ld openrisclib
    $(LD) -Bstatic $(LIBDIR)/reset.o $(PROGRAM) $(LDFLAGS) -o $(SIMPROGRAM)
    $(DUMP) $(SIMPROGRAM) > $(SIMPROGRAM).txt

# Run the simulator on the program
sim: $(SIMPROGRAM)
    $(SIM) -i -f sim.cfg $(SIMPROGRAM)

clean:
    rm -f *.o *~ sim.profile $(PROGRAM) $(SIMPROGRAM) $(HEXFILE) *.txt uart0.tx uart0.rx

```

We place all the segments of `simpleprog` at the address `0x2000` with the link script shown in Listing 1.3.

Listing 1.3: Link script for `simpleprog` (Listing 1.1)

```
MEMORY
{
  vectors   : ORIGIN = 0x00000000, LENGTH = 0x00002000
  sdram     : ORIGIN = 0x00002000, LENGTH = 0x03ffe000
}

SECTIONS
{
  .vectors :
  {
    *(.vectors)
  } > vectors

  .text :
  {
    *(.text)
  } > sdram

  .rodata ALIGN(4) :
  {
    *(.rodata)
  } > sdram

  .rodata.str1.1 ALIGN(4) :
  {
    *(.rodata.str1.1)
  } > sdram

  .data ALIGN(4):
  {
    *(.data)
  } > sdram

  .bss ALIGN(4):
  {
    *(.bss)
  } > sdram
}

```

1.4.3 The simulator `or32-uclinux-sim`

The simulator is started with the command

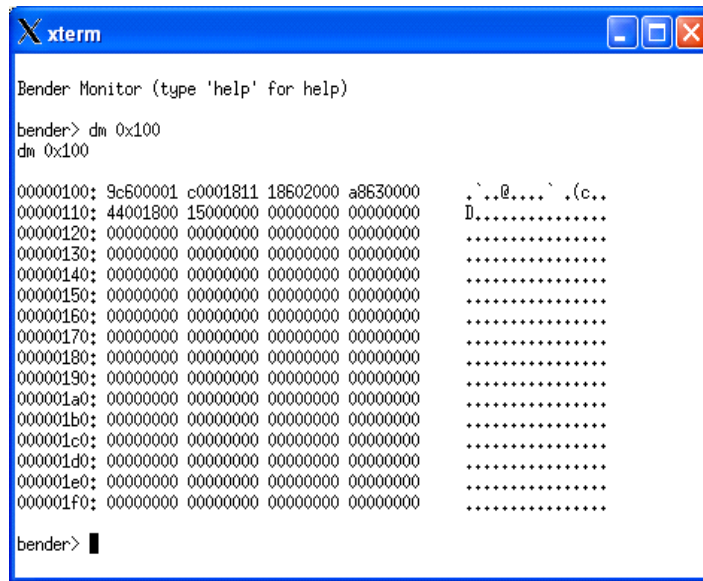
```
or32-uclinux-sim -f sim.cfg prog ,
```

where `sim.cfg` describes the hardware and `prog` is the program to run on the simulated hardware. Some help is printed out by the command

```
or32-uclinux-sim -h .
```

The simulator can also be started in an interactive mode by

```
or32-uclinux-sim -f sim.cfg -i prog .
```



```

xterm
Bender Monitor (type 'help' for help)
bender> dm 0x100
dm 0x100

00000100: 9c600001 c0001811 18602000 a8630000      ,`..@....`.(c..
00000110: 44001800 15000000 00000000 00000000      D.....
00000120: 00000000 00000000 00000000 00000000      .....
00000130: 00000000 00000000 00000000 00000000      .....
00000140: 00000000 00000000 00000000 00000000      .....
00000150: 00000000 00000000 00000000 00000000      .....
00000160: 00000000 00000000 00000000 00000000      .....
00000170: 00000000 00000000 00000000 00000000      .....
00000180: 00000000 00000000 00000000 00000000      .....
00000190: 00000000 00000000 00000000 00000000      .....
000001a0: 00000000 00000000 00000000 00000000      .....
000001b0: 00000000 00000000 00000000 00000000      .....
000001c0: 00000000 00000000 00000000 00000000      .....
000001d0: 00000000 00000000 00000000 00000000      .....
000001e0: 00000000 00000000 00000000 00000000      .....
000001f0: 00000000 00000000 00000000 00000000      .....

bender> █

```

Figure 1.8: Simulation of the bender monitor.

In Figure 1.8 we show as an example the simulation of a simple monitor in an xterm window.

The command `help` lists available commands, for instance `t` (trace):

```

>t
00000100: : 00000000 l.j 0x0 (executed) [time 40ns, #1]
00000104: : 00000000 l.j 0x0 (next insn) (delay insn)
GPR00: 00000000 GPR01: 00000000 GPR02: 00000000 GPR03: 00000000
GPR04: 00000000 GPR05: 00000000 GPR06: 00000000 GPR07: 00000000
GPR08: 00000000 GPR09: 00000000 GPR10: 00000000 GPR11: 00000000
GPR12: 00000000 GPR13: 00000000 GPR14: 00000000 GPR15: 00000000
GPR16: 00000000 GPR17: 00000000 GPR18: 00000000 GPR19: 00000000
GPR20: 00000000 GPR21: 00000000 GPR22: 00000000 GPR23: 00000000
GPR24: 00000000 GPR25: 00000000 GPR26: 00000000 GPR27: 00000000
GPR28: 00000000 GPR29: 00000000 GPR30: 00000000 GPR31: 00000000 flag: 0

```

1.4.4 μ Clinux

μ Clinux, which stands for microcontroller Linux, is a Linux variant intended for computers without a Memory Management Unit (MMU). This means that the kernel and the processes reside in the same address space.

You can start μ Clinux by giving the `u` command from the boot monitor. μ Clinux is now copied from FLASH (`0xf0100000`) to SDRAM (`0x0`) and the booting process starts.

The command `help` will list the built-in shell commands.

An important file is `/etc/rc`, the start-up file, which is shown in Listing 1.4. If you want to change the start-up behavior of μ Clinux this the file to change. In a running μ Clinux this file resides in a non-writable file system. A new system must be recompiled on a host computer, downloaded over the serial port and flashed to the flash memory. It is very unlikely that you have to do this in the course of this lab series.

Listing 1.4: μ Clinux configuration file `/etc/rc`

```
#!/bin/sh
#
setenv PATH /bin:/sbin:/usr/bin
hostname bender
#
mount -t proc none /proc
#
/bin/expand /ramfs512.img /dev/ram1
mount -t ext2 /dev/ram1 /var
mkdir /var/log /var/log/boa /var/lock /var/tmp /var/run
chmod 777 /var/tmp
#
/bin/expand /ramfs8192.img /dev/ram2
mount -t ext2 /dev/ram2 /mnt
mkdir /mnt/bin

# Set up the webserver stuff
mkdir /mnt/htdocs
cp /misc/* /mnt/htdocs
mkdir /mnt/htdocs/cgi-bin

# Bring up the local interface
/sbin/ifconfig lo 127.0.0.1
/sbin/route add -net 127.0.0.0

# Set IP address from configuration data in flash
/sbin/setip

# Start the web server
/sbin/boa -d &
```

Running programs under μ Clinux

We demonstrate how to run a program in the μ Clinux environment by an example. The program, shown in Listing 1.5, displays the contents of a Special Purpose Register (SPR). It uses inline assembler to read a register. We use the Makefile shown in Listing 1.6 to compile the program shown in Listing 1.5. The flags `-r` and `-d` to `$(CC)` are important, otherwise the program will not execute.

Listing 1.5: Program showing contents of a special purpose register.

```

#include <sys/types.h>
#include <sys/stat.h>
#include <asm/io.h>
#include <asm/spr_defs.h>
#include <asm/board.h>

int main (int argc, char *argv[])
{
    unsigned long val, addr;

    if (argc == 2) {
        addr = strtoul (argv[1], 0, 0);

        /* Read SPR */
        asm("l.mfspr,%0,%1,0": "=r" (val) : "r" (addr));
        printf ("\nSPR_%04lx:_%08lx\n", addr, val);
    } else return -1;
    return 0;
}

```

Listing 1.6: Makefile to compile the program showing an SPR (Listing 1.5).

```

CC = or32-uclibc-gcc
STRIP = or32-uclibc-strip

PRGS = mfspr

all: $(PRGS)

mfspr: mfspr.o
    $(CC) -r -d mfspr.o -o $@
    $(STRIP) -g $@

```

Finally the program can be downloaded with tftp. Change directory (cd) to a writable portion of the filesystem, like for instance /var/tmp. Then start the tftp client

```

> tftp IP_address_of_your_tftp_server
Retrieve the program:
> get mfspr

```

See section 4.3.2 for more information on how to start the TFTP server.

Chapter 2

Lab task 0 - Build a UART in Verilog

2.1 Introduction

In this introductory lab exercise you will learn the HDL Verilog. We require that you are familiar with another HDL, typically VHDL. In our opinion hardware design is done by drawing hardware diagrams, so that the programming in Verilog is just a final simple translation step!

You will also get (re)acquainted with the tools used in this course, ModelSim and make (or Xilinx Project Navigator).

The initial tests will use the Zedboard instead of the described VirtexII. Once the design works on the Zedboard, you will move the design to the VirtexII board.

2.2 A simple UART

A computer system is hard to use if you can not communicate with it. One of the simplest ways to send and receive characters is to use a serial communication protocol such as RS232.

2.2.1 General

For the lab you will have to download `tsea44.tgz`. Uncompress the downloaded file using `tar xzvf tsea44.tgz`. Make sure the path does not contains spaces or other strange characters. Inspect the directory `hw` and you will find:

- `lab0_zed.sv`, a skeleton for the top file when using zedboard.
- `lab0.sv`, a skeleton for the top file when using VirtexII.
- `lab0_zed.ucf`, `lab0.ucf`, User Constraints Files for use with zedboard and VirtexII respectively.
- `dafk_tb`, a folder containing the testbenches for both zedboard and VirtexII board.

- `simulator`, a folder containing tcl script files for use with the modelsim simulator.

2.2.2 The RS232 protocol

In this exercise you shall design a simple RS232 transceiver in Verilog. We assume that the serial port of the FPGA board is connected to a PC, where a terminal program is running. This is typically `gtkterm` if you are using Linux or `Teraterm` if you are running in Windows. The bit rate should be fixed 115200 bits/s. Your design shall use the parameters 8N1, that is 8 message bits, no parity bit and 1 stop bit, see Figure 2.1. Messages are sent and received with LSB first. Furthermore your UART shall support full duplex operation, that is be able to transmit and receive at the same time.

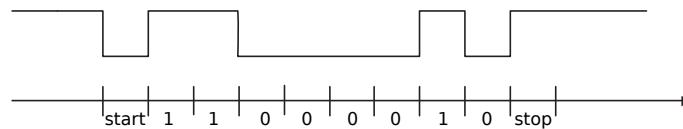


Figure 2.1: The letter C (0x43). Time per bit is 8.68 μ s.

2.2.3 The zedboard hardware

The system clock of the zedboard is running at 100 MHz. You will need a reset-signal and a send-signal, see Figure 2.2. Both these signals are active-high.

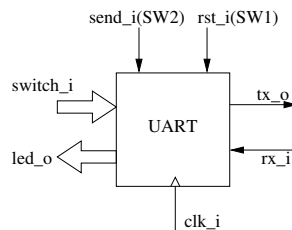


Figure 2.2: The UART.

Your task is twofold with an additional feature:

- send exactly one ASCII-coded character from the switches of the zedboard to the PC by pressing the button BTNU switch once, see Figure 2.2.
- catch the incoming characters from `rx_i` and present the ASCII code on the LED display, see Figure 2.2.
- when reset is pressed and released (BTND switch) output (and keep) on the LED the Binary Coded Decimal (BCD) value of the last two digits of your student-id. That is, the 4 leftmost bits show the binary value of the first middle digit in your student-id, and the 4 rightmost bits show the binary representation of the last digit in your student-id. Example: the student `linus123` should output the value 23 on the LED (00100011).

Some advice before you start:

- The signal `rx_i` is asynchronous. We strongly advice you to synchronize it!
- You will use your UART on the VirtexII board in the second part of lab0 with a slower system clock of 40 MHz, and finally in lab task 1 with an even slower system clock of 25 MHz. We suggest that you prepare the frequency change with an `'ifdef 'else 'endif` construct.
- Avoid continously sending the character while BTNU is pressed.

2.2.4 A simple testbench

You will also need a test bench. Since you are designing both a transmitter and a receiver you may choose to test them both at the same time, see Figure 2.3.

There are two different boards used, and there are therefore two different testbenches available. Use the `lab0_zed_tb` to test the design to run on the zedboard. The corresponding designfile is then named `lab0_zed.sv`.

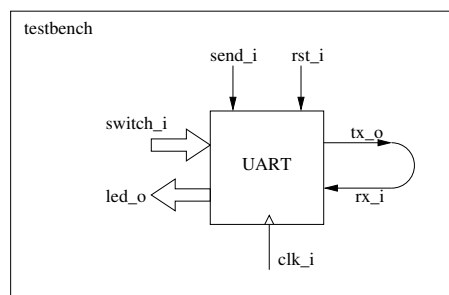


Figure 2.3: A testbench.

The code for the test bench shown in Figure 2.3, is listed in Listing 2.1.

Listing 2.1: Test bench for the UART.

```
'timescale 1ns / 10ps

module lab0_zed_tb();

    reg clk_i;
    reg rst_i;
    reg send_i;
    reg [7:0] switch_i;
    wire [7:0] led_o;
    wire jumper;

    // Instantiate a UART
    lab0_zed uart(.clk_i(clk_i), .rst_i(rst_i), .rx_i(jumper), .tx_o(jumper),
                 .led_o(led_o), .switch_i(switch_i), .send_i(send_i)) ;

    always #5 clk_i = ~clk_i; // 100 MHz clock

    initial
    begin
        clk_i = 1'b0;
        switch_i = 8'h43; // C
        rst_i = 1'b1;
        send_i = 1'b0;
        #200 rst_i = 1'b0;
        #1000 send_i = 1'b1;
        #1100 send_i = 1'b0;
    end

endmodule
```

2.3 Exercises

Preparation task 1

Individual! Draw a HW diagram of the UART. Use simple components like counters, registers, shift registers, and state machines. Show this to the lab supervisor at the start of the lab!

Laboration task 0.1

Individual!

- a) *Translate your HW diagram into Verilog code to be used on the zedboard. Use the lab0_zed.sv file*
- b) *Simulate your design in ModelSim, using the lab0_zed_tb*
- c) *Synthesize your design, program the FPGA and test run your design.*
- d) *Move the design to the VirtexII board. Do this by copying the code into lab0.sv, update the clock frequency, and if necessary modifying the testbench lab0_tb.sv*

2.3.1 Commands

Add definitions of tools to use by adding the course module using the command `module load courses/TSEA44`. To start the simulator with the zedboard design, use the command `make sim_lab0_zed`, and to start the simulator with the VirtexII design use the command `make sim_lab0`. To generate a bitfile to program the FPGA of the zedboard use `make lab0_zed.bit` and for the VirtexII use `make lab0.bit`.

To configure the Zedboard FPGA with a .BIT file, use `sh utils/download_zed.sh lab0_zed.bit`, and use `sh utils/download.sh lab0.bit` to program the VirtexII board. There are also a makefile target `prog_lab0_zed` and `prog_lab0` to simplify these tasks.

2.3.2 A User Constraint File

You will need the User Constraint File that describe the board signals. The list used for the zedboard is shown in Listing 2.3. The exact same signals and names mentioned in Listing 2.3 must be present in the interface declaration of you top module. Comment out the lines that you don't use. (This file is included in the lab skeleton as lab0_zed.ucf for the zedboard and lab0.ucf for the Virtex II board.)

Listing 2.2: User constraints file for your UART on zedboard

```

NET "clk_i"          LOC = "Y9" | IOSTANDARD=LVCMOS33; // 100 MHz on Zedboard
NET "rst_i"         LOC = "R16" | IOSTANDARD=LVCMOS18; // BTND (downward) on green flexo
NET "send_i"        LOC = "T18" | IOSTANDARD=LVCMOS18; // BTNU (up) on green flexo
// black switches
NET "switch_i<0>"   LOC = "F22" | IOSTANDARD=LVCMOS18; // SW0
NET "switch_i<1>"   LOC = "G22" | IOSTANDARD=LVCMOS18; // SW1
NET "switch_i<2>"   LOC = "H22" | IOSTANDARD=LVCMOS18; // SW2
NET "switch_i<3>"   LOC = "F21" | IOSTANDARD=LVCMOS18; // SW3
NET "switch_i<4>"   LOC = "H19" | IOSTANDARD=LVCMOS18; // SW4
NET "switch_i<5>"   LOC = "H18" | IOSTANDARD=LVCMOS18; // SW5
NET "switch_i<6>"   LOC = "H17" | IOSTANDARD=LVCMOS18; // SW6
NET "switch_i<7>"   LOC = "M15" | IOSTANDARD=LVCMOS18; // SW7
// row of LEDs above switches
NET "led_o<0>"      LOC = "T22" | IOSTANDARD=LVCMOS33; // LED LD0
NET "led_o<1>"      LOC = "T21" | IOSTANDARD=LVCMOS33; // LED LD1
NET "led_o<2>"      LOC = "U22" | IOSTANDARD=LVCMOS33; // LED LD2
NET "led_o<3>"      LOC = "U21" | IOSTANDARD=LVCMOS33; // LED LD3
NET "led_o<4>"      LOC = "V22" | IOSTANDARD=LVCMOS33; // LED LD4
NET "led_o<5>"      LOC = "W22" | IOSTANDARD=LVCMOS33; // LED LD5
NET "led_o<6>"      LOC = "U19" | IOSTANDARD=LVCMOS33; // LED LD6
NET "led_o<7>"      LOC = "U14" | IOSTANDARD=LVCMOS33; // LED LD7
// PMOD JB connected to USB-UART
NET "rx_i"          LOC = "V10" | IOSTANDARD=LVCMOS33; // PMOD B, JB3
NET "tx_o"          LOC = "W11" | IOSTANDARD=LVCMOS33; // PMOD B, JB2

```

Listing 2.3: User constraints file for your UART on VirtexII

```

NET "clk_i"          LOC = "AK19"; // 40 MHz in this lab
NET "rst_i"         LOC = "C2"; // SW1 (red) on green flexo
NET "send_i"        LOC = "B3"; // SW2 (black) on green flexo
// blue DIP switches
NET "switch_i<7>"   LOC = "AL3"; // SWITCH 1
NET "switch_i<6>"   LOC = "AK3"; // SWITCH 2
NET "switch_i<5>"   LOC = "AJ5"; // SWITCH 3
NET "switch_i<4>"   LOC = "AH6"; // SWITCH 4
NET "switch_i<3>"   LOC = "AG7"; // SWITCH 5
NET "switch_i<2>"   LOC = "AF7"; // SWITCH 6
NET "switch_i<1>"   LOC = "AF11"; // SWITCH 7
NET "switch_i<0>"   LOC = "AE11"; // SWITCH 8
// row of LEDs
NET "led_o<7>"      LOC = "N9"; // LED D4
NET "led_o<6>"      LOC = "P8"; // LED D5
NET "led_o<5>"      LOC = "N8"; // LED D6
NET "led_o<4>"      LOC = "N7"; // LED D7
NET "led_o<3>"      LOC = "M6"; // LED D8
NET "led_o<2>"      LOC = "M3"; // LED D9
NET "led_o<1>"      LOC = "L6"; // LED D10
NET "led_o<0>"      LOC = "L3"; // LED D11
// rainbow flat cable
NET "rx_i"          LOC = "M9";
NET "tx_o"          LOC = "K5";

```

2.4 gtkterm usage

Start `gtkterm` in a shell or from **Applications->Accessories->GTKTerm** if your computer have the board to use connected. All computers have a zedboard connected, but there are fewer VirtexII boards.

Communication parameters are set from **Configuration->Port** and should for the zedboard be `/dev/ttyUSB0` and for the VirtexII board be `/dev/ttyUSB2`. For both boards use speed 115200, no parity, 8 bits, 1 stop bit and no flow control.

Chapter 3

Lab task 1 - Interfacing to the Wishbone bus

3.1 Introduction

In this lab exercise you will get acquainted with the OR 1200 RISC processor and particularly the Wishbone bus. You will do this by designing and interfacing two modules, a UART and a performance counter module to the Wishbone bus.

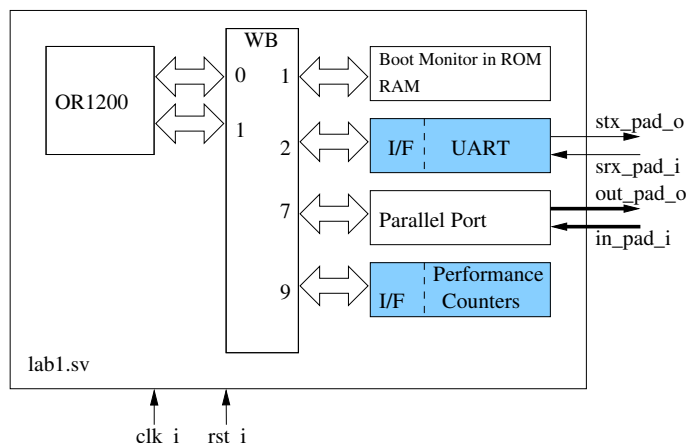


Figure 3.1: The computer. The two gray modules will be designed by you.

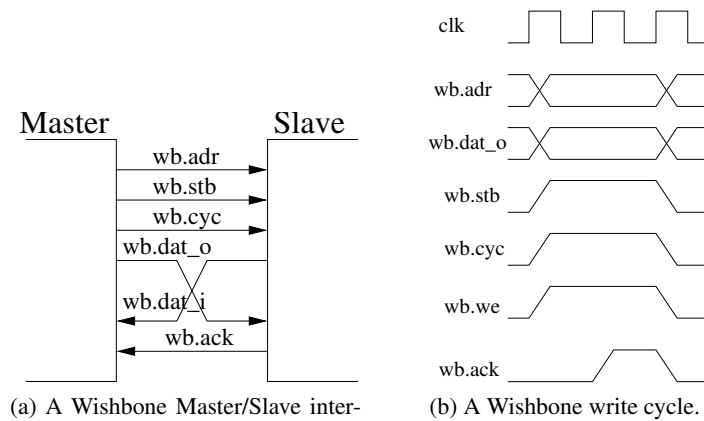
Figure 3.1 depicts the computer that you are going to work with in this laboratory exercise. You will have to:

1. modify your UART from the previous lab and interface it to the Wishbone bus. The wishbone interface should be inserted into `lab1/lab1_uart_top.sv`.
2. check the UART device drivers in the boot monitor. The driver is in this file `monitor/firmware/src/uartfun.c`.
3. download and execute a benchmark program, that performs the DCT part of JPEG compression on a small image in your RAM module.

4. simulate the computer running the benchmark program.
5. design a module containing hardware performance counters (`perf_top.sv` in the lab skeleton).

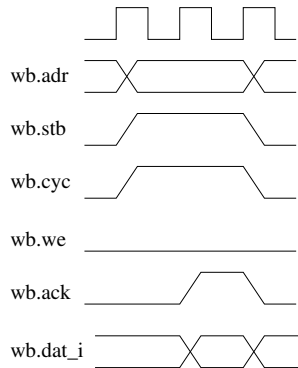
3.2 Some Basic Facts on the Wishbone Bus

The Wishbone bus is intended for implementation in FPGAs or ASICs. Typical for such a bus is that multiplexers are used instead of tristate buffers. Two data buses are used, one for each direction, see Figure 3.2a.



(a) A Wishbone Master/Slave interface.

(b) A Wishbone write cycle.



(c) A Wishbone read cycle.

Figure 3.2: The Wishbone bus protocol.

In this lab we will only need a subset of the Wishbone protocol, namely the basic write and read bus cycles.

For the write cycle, see Figure 3.2b, we have:

1. The master places address and data on the buses `wb.adr` and `wb.dat_o`, respectively. Finally the master asserts the `wb.stb`-signal, `wb.cyc`-signal, and `wb.we`-signal.
2. The slave, when ready, decodes the address bus, latches the data and asserts the `wb.ack`-signal.

3. The Master deasserts the `wb.stb`, `wb.cyc` and `wb.we`-signals.
4. The slave deasserts the `wb.ack`-signal.

For the read cycle, see Figure 3.2c, we have:

1. The master places the address on the bus `wb.adr` and asserts the `wb.stb`-signal, the `wb.cyc`-signal, and deasserts the `wb.we`-signal.
2. The slave, when ready, decodes the address bus, places the data on the data bus `wb.dat_i` and asserts the `wb.ack`-signal.
3. The Master deasserts the `wb.stb` and `wb.cyc`-signals.
4. The slave deasserts the `wb.ack`-signal.

In these basic write and read bus cycles the `wb.stb` and `wb.cyc`-signals are identical. The `wb.cyc`-signal is used for arbitration of the bus, so the master may assert it for many cycles, for instance during a cache line refill.

3.2.1 A Wishbone Interconnect

Before we begin with the actual integration of the computer we would like to give a short explanation of the Wishbone interconnect. In Figure 3.3 we show an example of 2 masters and 3 slaves connected to a Wishbone bus. The `m-bus` is all the signals going from the master to a slave, like the address bus, data bus and in particular the `stb`-signal. The `s-bus` is all the signals going from the slave to a master, like the data bus and the `ack`-signal.

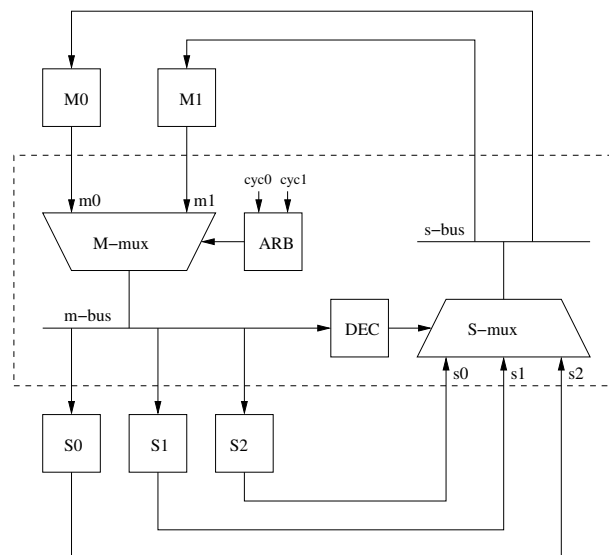


Figure 3.3: A Wishbone interconnect for 2 masters and 3 slaves.

An arbiter, a finite state machine, listens to the `cyc`-signals from the masters. The masters `M0` and `M1` are, in our implementation, granted the bus in a round-robin fashion. The `m-bus` is then connected to all the slaves. The `stb`-signal is, however, only asserted at the addressed slave port.

In the return path the addressed slave's s-bus is connected to all the masters. This is handled by the block DEC. The ack-signal is, however, only asserted at the master that won the arbitration.

3.3 A Simple Computer

3.3.1 General

For the lab you will have to download `tsea44.tgz` if you haven't done so already. Uncompress the zip-file to your home directory. Inspect the directory `hw` and you will find:

- the file `lab1/lab1_uart_top.sv`, a skeleton for the top file.
- the file `lab1.ucf`, a User Constraints File.
- the directory `or1200` containing the CPU. The top file is `or1200_top.sv`.
- the directory `monitor` containing both HW and SW for the boot monitor.
- the directory `wb` containing the Wishbone interconnect.
- the directory `include` containing some include files.
- the directory `firmware`, which contains the example program `dct_sw/dct_sw` that can be downloaded to your computer with the boot monitor.

3.3.2 A Wishbone Interface for the UART

Let's start our computer design with the UART. In the introductory lab you designed a simple UART. All that is needed now is to attach a Wishbone interface to your design, see Figure 3.4.

Since you will use a boot monitor that is written for the standard 16550 UART, you will want to make your design emulate that UART.

Luckily our device driver does not use much of the functionality in the 16550. The main enhancement in the 16550 are 16 character FIFOs in both directions. This is more or less mandatory when you run an OS, which always has some interrupt latency.

The driver routine expects three bytesized registers:

1. *transmit register*, `adr=0`, write-only
2. *receive register*, `adr=0`, read-only
3. *status register*, `adr=5`, read-only

In the *status register*, you will only need two F/Fs:

- `rx_full`, set when the stop-bit is received and reset when the receive register is read. Use signal `wb_sel[3]` to determine when the receive register is read.

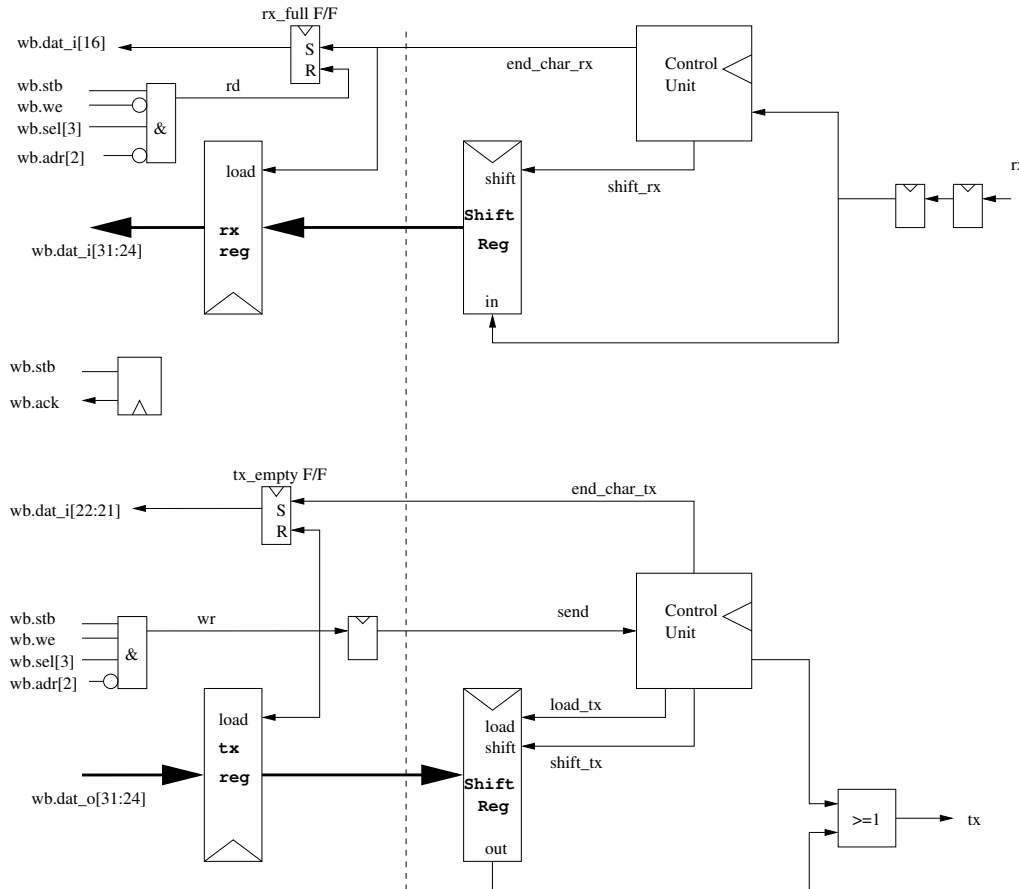


Figure 3.4: A sketch of the Wishbone interface for the UART. The signal `load_tx` is a single-pulsed version of `send`. The `tx_empty` F/F is connected to two wires.

- `tx_empty`, set when the stop-bit has been transmitted and reset when the transmit register is written. The 16550 has two slightly different flags for this case. The monitor will work if you connect `tx_empty` to both these flags.

Figure 3.5 shows address maps for the UART connected to an 8 bit bus and a 32 bit bus. The *transmit register* should be placed on `wb.dat_o[31:24]`, the *receive register* on `wb.dat_i[31:24]`. The *status register* should be placed on `wb.dat_i[23:16]`. What about address decoding? The 8 most significant bits are already decoded in the `wb.stb`-signal. Since we are now using a 32 bit data bus, we will not use the two least significant address bits. Instead the `wb.sel`-signal is used to access individual data bytes. For instance `wb.sel[3]` is asserted when a byte on address `0x9000_0000` or (for instance) `0x9000_0004` is accessed. To prevent an access to `0x9000_0004` to reset the status F/Fs, we connect `wb.adr[2]` to the AND gates.

Preparation task 2

Why must the `wb.sel[3]`-signal be included in the reset condition for the `rx_full` F/F?

The code for the lab skeleton `lab1_uart_top.sv` is given in the listing 3.1. The

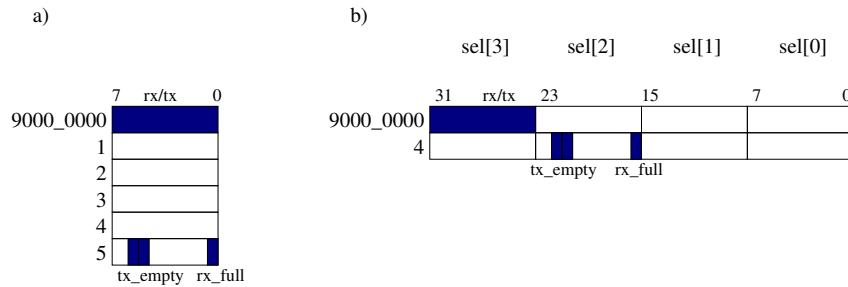


Figure 3.5: a) Address map for the UART connected to an 8 bit bus b) Address map for the UART connected to a 32 bit bus. The sel-signals are used to address individual bytes.

definition of the wishbone SystemVerilog interface can be found in the appendix section B.5.

Listing 3.1: Lab skeleton lab1_uart_top.sv.

```

module lab1_uart_top
  (wishbone.slave wb,
   output wire int_o ,
   input wire  srx_pad_i ,
   output wire stx_pad_o);

  assign int_o = 1'b0;      // Interrupt, not used in this lab
  assign wb.err = 1'b0;    // Error, not used in this lab
  assign wb.rty = 1'b0;    // Retry, not used in this lab
  assign wb.ack = wb.stb;  // change if needed

  // Here you must instantiate lab0_uart or cut and paste
  // You will also have to change the interface of lab0_uart to make this work.
  assign stx_pad_o = srx_pad_i; // Change this line.. :)
endmodule

```

Preparation task 3

Write Verilog code for the Wishbone interface of your UART.

Preparation task 4

Inspect the driver routines `getch` and `putch` in the file `monitor/firmware/src/uartfun.c`. You will also have to look in `uartfun.h`.

3.3.3 The Monitor

The monitor directory contains a couple of Verilog files that implements an 8 kB block RAM at base address 0x4001_0000. This RAM will contain the stack of the monitor. The monitor itself is implemented in an 24 kB block ROM at base address 0x4000_0000. The contents of the block ROM is in the Verilog file `mon_prog_bram_contents.v`.

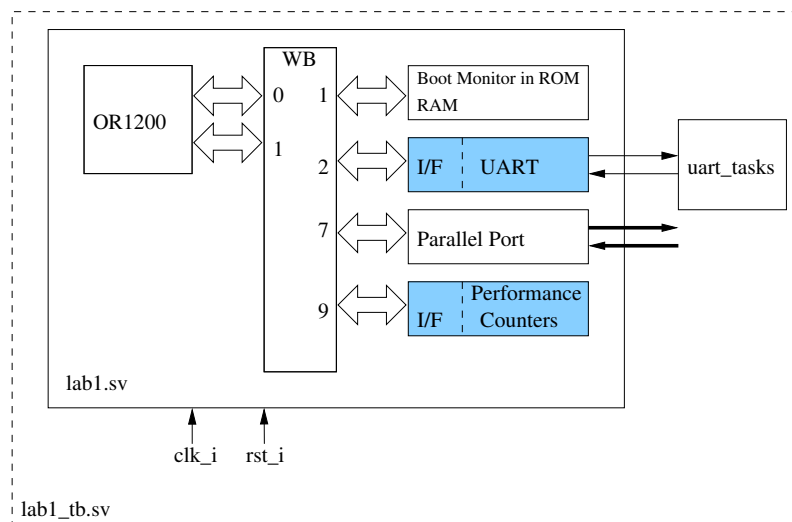
The software is in the sub directories `firmware/src` and `firmware/include`.

Check `mon2.c` to see what the monitor does at startup so that you can verify that the hardware does the correct thing.

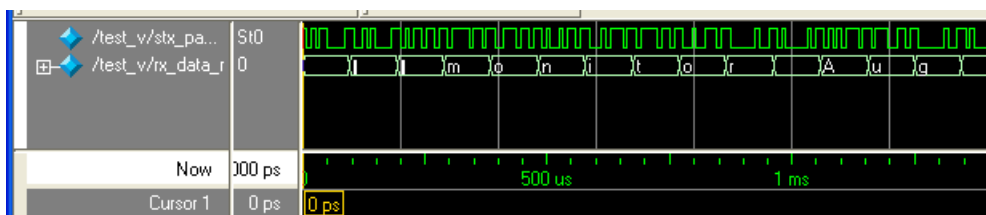
3.3.4 Test Your Design

In Figure 3.6a we show a test bench for the computer. The only signals that the test bench has to activate in this case are the `clk_i`- and `rst_i`-signals. We check the behavior of the computer by listening to `tx`-signal from the UART. Part of a testbench has already been written for you in `dafk_tb/lab1_tb.v`. This test bench can be started with `make sim_lab1`.

Remember that you may be required to change parts of the testbench also. Make sure to verify the testbench outputs to verify that both the input pattern and the output patterns are as expected.



(a) A test bench. The module `uart_tasks` gives a nice printout.



(b) A test run in ModelSim, showing the signals `tx` and `rx_data` in the test bench.

Figure 3.6: Simulation of your design

There is also a smaller test bench, `lab1/uart_tb.v`, that will test only your UART design. This test bench can be started with `make sim_uart`. Remember to verify that the generated testpattern is correct, and adjust the testbench if necessary.

Laboration task 1.1

Test your computer.

3.4 A Benchmark Program

3.4.1 JPEG Compression

We will use the first part, DCT, of the JPEG compression algorithm to test our computer. This section is inspired by [3]. We begin with a short discussion of how DCT works.

3.4.2 Integer DCT

The two dimensional discrete cosine transform (DCT) for an 8×8 array $a[x, y]$ is defined as

$$A[u, v] = c[u]c[v] \cdot \sum_{x=0}^7 \sum_{y=0}^7 a[x, y] \cos \frac{\pi u}{8} \left(x + \frac{1}{2}\right) \cos \frac{\pi v}{8} \left(y + \frac{1}{2}\right) \quad (3.1)$$

where $c[0] = 1/\sqrt{8}$ and $c[u] = 1/2$ when $u \neq 0$.

The transform in (3.1) can be separated. First compute a one-dimensional DCT on each row and then a one-dimensional DCT on each column.

$$A[u, v] = c[v] \cdot \sum_{y=0}^7 \left\{ c[u] \sum_{x=0}^7 a[x, y] \cos \frac{2\pi}{32} (2x + 1)u \right\} \cos \frac{2\pi}{32} (2v + 1)y \quad (3.2)$$

The innermost part of (3.1) is the 1-D DCT, which we repeat with slightly different notation:

$$A[u] = c[u] \cdot \sum_{x=0}^7 a[x] \cos \left(\frac{2\pi}{32} (2x + 1)u \right) \quad (3.3)$$

By using all possible symmetries of the cosine function it is not so difficult to figure out a fast DCT, see Figure 3.7. This computation scheme is usually referred to as Loeffler's algorithm. It computes the 1-D DCT as defined in (3.2) multiplied with $\sqrt{8}$.

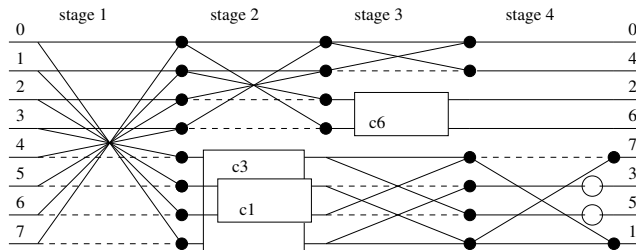


Figure 3.7: Loeffler's original algorithm for fast DCT. Black circles means addition, dashed lines multiplication with -1 and white circles multiplication with $\sqrt{2}$. White boxes marked cn denote rotation with $n\pi/16$.

The white boxes in Figure 3.7 denote rotation with $n\pi/16$:

$$\begin{cases} x_{out} &= x_{in} \cdot \cos n\pi/16 + y_{in} \cdot \sin n\pi/16 \\ y_{out} &= -x_{in} \cdot \sin n\pi/16 + y_{in} \cdot \cos n\pi/16 \end{cases} \quad (3.4)$$

Sofar we have presented three ways of computing the 2-D DCT. We compare the computation complexity of the algorithms:

Algorithm	MUL	ADD
Eq (3.1)	4096	4032
Eq (3.2)	1024	892
Loeffler original	224	416

The post multiplication with $c[u]$ has been left out of the table.

The OR1200 CPU has no floating point arithmetic, so the sin/cosine factors and $\sqrt{2}$ in Figure 3.7 must be mapped to integers. We have chosen to multiply with 2^{13} and rounding to the nearest integer. We have the following correspondences:

Real number	Integer
$\sqrt{2}$	11585
$\cos \pi/16$	8035
$\sin \pi/16$	1598
$\cos 3\pi/16$	6811
$\sin 3\pi/16$	4551
$\sqrt{2} \cos 6\pi/16$	4433
$\sqrt{2} \sin 6\pi/16$	10703

The scheme in Figure 3.7 has a serious drawback. The outputs 3 and 5 pass through two multipliers. There is, however, a modified version where this flaw has been removed at the price of 1 extra multiplier and 3 extra adders. The last 3 stages on the lower part of Figure 3.7 is replaced with the computation scheme in Figure 3.8. This modification is not, in our opinion, so easy to figure out. The interested reader is therefore directed to the original article, [4].

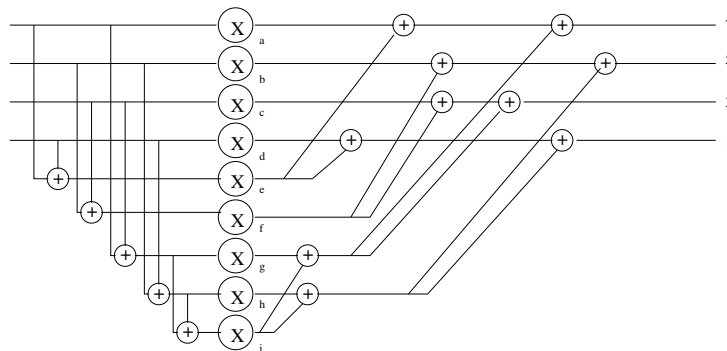


Figure 3.8: Loeffler's modified algorithm. Computation of the odd part with parallel multiplications.

We conclude that Loeffler's modified algorithm can be used with integer arithmetic. After each run all outputs, except 0 and 4, must be arithmetic right shifted 13 steps. The 2-D DCT will be $8 \cdot A[u, v]$ compared to (3.2), which can be compensated for in later stages in the JPEG compression algorithm.

Preparation task 5

Why do we go through all the trouble inserting the module in Figure 3.8? Why is it so bad having 2 multipliers in series?

3.4.3 The Test Program `dct_sw`

For this lab you will get a test program `dct_sw.c`, written by us. It is a straightforward implementation of Loeffler's algorithm and computes the 2-D DCT of an 8×8 image. You will actually find two copies:

- in the directory `hw/firmware/dct_sw` for downloading and running on the target computer. You can also run it on the host computer.
- in the directory `hw/monitor/firmware/src` for simulation. A call to the DCT program has been inserted in the beginning of the monitor program `mon2.c`.

3.4.4 A Test Example

$$a[x, y] = \begin{pmatrix} \mathbf{1} & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\ 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 \\ 57 & 58 & 59 & 60 & 61 & 62 & 63 & 64 \end{pmatrix} \quad (3.5)$$

Origo is shown in bold text.

$$8 \cdot DCT[a - 128] = \begin{pmatrix} -\mathbf{6112} & -152 & 0 & -16 & 0 & -8 & 0 & -8 \\ -1167 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -122 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -37 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (3.6)$$

3.5 Design a Performance Counter Module

As a last task in this lab you shall design a simple performance counter module, see Figure 3.1. The module shall:

1. be connected to slave port 9 of the Wishbone bus.
2. have the port definition shown in Listing 3.2.
3. contain four 32 bit counters that can be read and written on the addresses `0x9900_0000` to `0x9900_000c`.

4. The counter on address `0x9900_0000` shall count the number of clock cycles that `m0.cyc` and `m0.stb` are both asserted. The counter on address `0x9900_0004` shall count the number of clock cycles that `m0.ack` is asserted.

5. The counter on address `0x9900_0008` shall count the number of clock cycles that `m1.cyc` and `m1.stb` are both asserted. The counter on address `0x9900_000c` shall count the number of clock cycles that `m1.ack` is asserted.

6. Be aware that you will add extra signals (and counters) to this module in later labs to measure DMA activities.

7. You may optionally use the `m?.we` signals to gather even more statistics.

Listing 3.2: Performance counter module port definition. The definition of the wishbone SystemVerilog interface can be found in the appendix, section B.5.

```

module perf_top ( wishbone.slave wb, wishbone.monitor m0, m1 );

  reg    [31:0]  ctr0 , ctr1;  // your counters

  assign  wb.ack = wb.stb && wb.cyc; // how to fix the ack-signal

  // your code goes here

endmodule // perf_top

```

Laboration task 1.2

Design the performance counter module and use these counters to measure the performance of the `dct_sw.c` program. There is also a free running timer present in the processor. You can access it on SPR register `0x5002`. In this lab you may also use the regular timer register in the processor since no operating system will modify it.

3.6 Useful Commands

We have prepared a makefile based build system that is responsible for both building the monitor firmware and synthesizing the hardware from the RTL source code. You can use it on the Linux computers in Muxen 1. The following targets will be useful for you:

- `make lab1` Creates a bit file of the computer in this lab task.
- `make sim_lab1` Launches Modelsim on the “lab1” system.
- `make sim_uart` Launches Modelsim on your UART.
- `make dafk` Creates a bit file of the complete system.
- `make sim` Launches Modelsim on the complete system.
- `make simfiles` Recompiles all source files for use with Modelsim but does not launch Modelsim itself. This is mainly useful if you already have Modelsim running and want to try out some changes to your source code. This way you don’t need to close Modelsim, it is enough to issue a `restart` command in Modelsim.
- `make clean` Removes intermediate files and backup files.
- `make updatebit` Compiles the monitor and updates `dafk.bit`, and `lab1.bit` with the new monitor. This way you don’t need to resynthesize the design to test changes in your monitor. The updated bit file is named `updated_dafk.bit` for `dafk.bit` and so on.

We also have some utilities that you might be interested in. The first of these is `download.sh` which you can use to download your design. Invoke it as in the following example:

```
utils/download.sh dafk.bit
```

Another utility is designed to highlight the error and warning messages in the various reports that the Xilinx flow will output. Use it on (for example) the synthesis report with the following command:

```
make dafk.bit | utils/checklogs.pl
```

3.6.1 Synthesis Reports

If you use `make`, the following files will be of special interest to you: (look in `/tmp/<your name>/synthdir`)

- `precision.log`: Synthesis report
- `foo_map.mrp`: Map report

- `foo.par`: Place and Route report
- `foo_timing.rep`: Timing analyzer report

(Where `foo` is the name of the top level file you compiled, as in `dafk` or `lab1`).

3.7 How to get Started Writing/Executing C Programs

A good starting point is the program `simpleprog` situated in the directory `firmware`. It can be compiled with `make` in Linux.

The executable file is `simpleprog.hex` which can be downloaded with the command `l` in the monitor and `File->Send Raw File` in `gtkterm`. You run the program with `g 2000` or just `g`.

The size of the DRAM is 64 MB, so there is plenty of room for your program. You can check the length of the program by looking inside the file `simpleprog.txt`, which is a disassembled version of `simpleprog`. The length of `simpleprog` is 2800 bytes.

3.7.1 A Note on Volatile

Normally, the compiler assumes that memory locations will not change unless the program itself changes it. This assumption does not hold when the program tries to access I/O memory. For example, in the following code shown in Listing 3.3, the programmer wants the program to wait until pin 1 of the parallel port is set to 1. The problem is that an optimizing compiler will generate assembler code doing approximately what is shown in Listing 3.4.

Listing 3.3: Volatile is not used for memory mapped I/O.

```
unsigned int *parport = 0x91000000;
while ((*parport & 0x1) != 1); /* Busy wait */
```

Listing 3.4: Resulting assembler from code in Listing 3.3.

```
LOAD R0,[0x91000000]    ; Load value from memory
AND  R0,R0,0x1         ; And R0 with 1
CMP  R0,0x1           ; Compare R0 with 1
loop :
  BNEQ loop          ; Jump to loop if R0 was not equal to 1
```

This is certainly not what the programmer had in mind. This kind of error is even more insidious because in some cases it might work ok and in some cases it will fail sporadically and in some cases it might not work at all. It will also depend on the optimization level of the compiler. The correct way to deal with this situation is to tell the C compiler that the memory location can change at any time. This will force the compiler to generate code that reloads the memory location every time it is referenced. This can be done using the `volatile` keyword. We recommend that you use the macros shown in Listing 3.5 to access memory mapped I/O. These macros are defined in both the monitor (`mon2.h`) and in `jpeglib.h` but if you write a small test program you might have to include them in your own source code as well. Using these macros¹

¹The macros assume that a long is 32 bits, a short is 16 bits and a char is 8 bits.

the program from Listing 3.3 would look like what's shown in Listing 3.6.

Listing 3.5: Recommended macros for memory mapped I/O access.

```
#define REG32(addr) *((volatile unsigned long*)(addr))
#define REG16(addr) *((volatile unsigned short*)(addr))
#define REG8(addr) *((volatile unsigned char*)(addr))
```

Listing 3.6: Correct program, using volatile.

```
while((REG32(0x91000000) & 0x1) != 1); /* Busy wait */
```

3.7.2 What to Include in the Lab Report

The lab report should contain all source code that you have written. (The source code should of course be commented.) We would also like you to include a block diagram of your hardware. If you have written any FSM you should include a state diagram graph of the FSM.

We would also like you to discuss the following questions:

- How did you verify that your computer hardware worked?
- What is the performance of the 2D DCT software? (Try it with and without caches.)
- How much of the FPGA is used by our design?

And of course, the normal parts of a lab report such as a table of contents, an introduction, a conclusion, etc. The source code that you have written should be included in appendices and referred to from the main document.

Chapter 4

Lab task 2 - Design a JPEG accelerator

4.1 The lab system

In this lab task you will learn how to build a hardware accelerator for the JPEG image compression algorithm. In this lab you will use the build target `dafk.bit`. This is a complete system with the following components:

- OR1200 CPU
- Boot monitor
- UART
- VGA controller
- Camera controller
- Ethernet controller
- SDRAM, SRAM, and flash memory controller

μ Clinux is programmed into the flash memory on the FPGA board and we will use this operating system for the remainder of this course. Examples of how to compile for Linux are included in the lab skeleton in the `hello` directory.

4.2 Proposed architecture

We propose the general architecture shown in Figure 4.1. It works in the following way:

1. An 8×8 bytes image is written from the Wishbone bus by the application program to the `in` RAM in 16 write cycles. Pixels are 8-bit positive numbers and packed in one 32-bit word. We recommend that you subtract 128 from each pixel before it is written to the `in` RAM. The accelerator is then started by setting the `START` bit in `csr` (Control/Status Register).

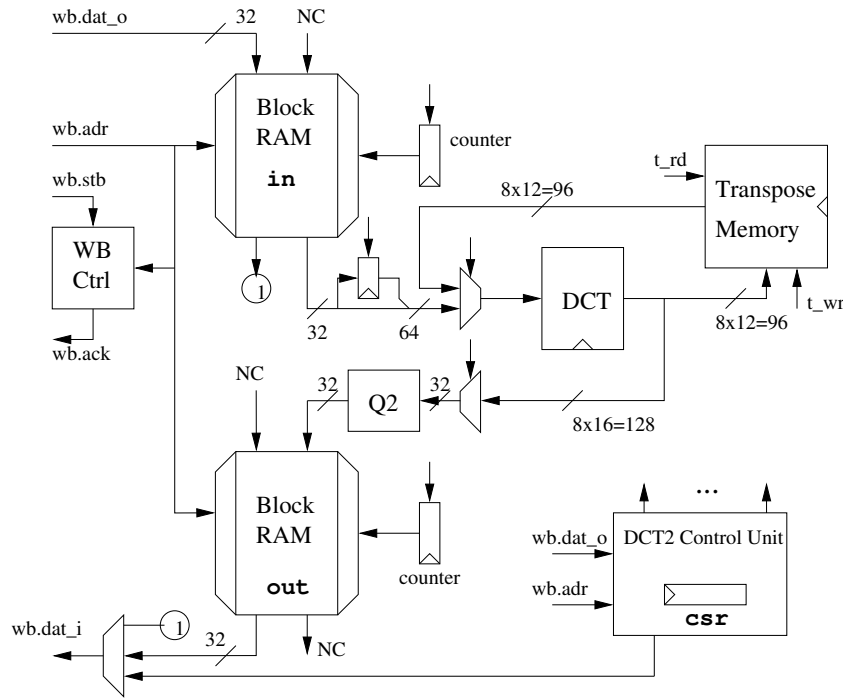


Figure 4.1: Proposed architecture for the 2-D DCT-accelerator. `csr` is a Control/Status register. Not all wires are shown.

2. A row of the image is read from the `in` RAM in 2 clock cycles. This is repeated 8 times.
3. The rows are transformed in DCT (12-bit signed numbers) and written to the transpose memory in the same tempo
4. When all rows have been written to `T`, columns, 8×12 bits, can be read from `T` and fed into the DCT again. A complete column can be read per clock cycle. After the second DCT the values are 16-bit signed numbers.
5. Finally 2×16 bits per clock cycle are quantized in `Q2` and written to the `out` RAM. When all columns have been written, the `RDY` bit in `csr` is set.

You will receive a Verilog module, `dct.v`, that computes a 1-D DCT multiplied with $\sqrt{8}$. This file is a straightforward implementation in Verilog of the computation schemes (modified Loeffler) in Figures 3.7 and 3.8 in Chapter 3.

Preparation task 6

Open the file `dct.v` and have a look at what it does. What are the inputs, what are the outputs? How many clock cycles does a computation take? Is it pipelined?

4.2.1 Block RAMs in VirtexII

The VirtexII-4000 FPGA contains 120 18 kbit block RAMs. They are dual ported with two completely independent sets of synchronous read and write ports. The easiest way

to use a block RAM is, in our opinion, to instantiate a library primitive. The code in Listing 4.1 instantiates a block RAM shown in Figure 4.2.

SSR is a set/reset signal, that only affects the output latches, not the RAM memory cells. DIP and DOP can be used for additional data such as parity bits but we do not use them in this lab. It is important to understand that both reads and writes are synchronous as opposed to an ordinary RAM that you might have used in one of our earlier courses such as *Digital Konstruktion*.

Listing 4.1: Instantiation of a block RAM as shown in Figure 4.2

```

wire [31:0] doa , dia , dob , dib ;
wire [8:0]  addra , addrb ;
wire  clk , cea , wea , ceb , web ;

// dual port 512x32 RAM
RAMB16_S36_S36 memory (
// port A
.DOA(doa) , .DOPA() , .ADDRA( addra ) , .CLKA( clk ) ,
.DIA( dia ) , .DIPA(4'h0) , .ENA( cea ) , .SSRA(1'b0) , .WEA( wea ) ,
// port B
.DOB(dob) , .DOPB() , .ADDRB( addrb ) , .CLKB( clk ) , .DIB( dib ) ,
.DIPB(4'h0) , .ENB( ceb ) , .SSRB(1'b0) , .WEB( web ) ) ;

```

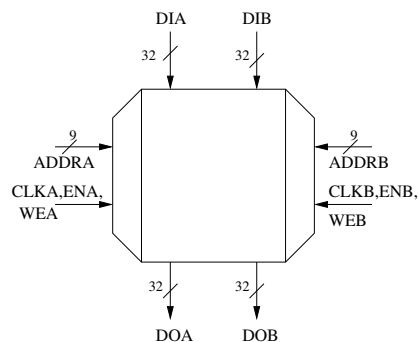


Figure 4.2: Dualported 512×32 bit block RAM.

4.2.2 Distributed RAMs

Small RAMs can be designed using the LUTs in the FPGA. A LUT is a 16×1 RAM. Distributed RAM memory supports the following:

- Single-port RAM with one synchronous write and one combinatorial read port
- Dual-port RAM with one synchronous write port and two asynchronous read ports

For instance a 16×8 RAM can be designed in Verilog as shown in Listing 4.2.

Listing 4.2: Distributed RAM instantiation in Verilog.

```

reg [7:0] mem[15:0];

```

```

wire [7:0] data_i , data_o ;
wire [3:0] addr_a , addr_b ;

// 1 combinatorial read port
assign data_o = mem[ addr_a ] ;

// 1 synchronous write port
always @(posedge clk) begin
  if (we)
    mem[ addr_b ] <= data_i ;
end

```

4.2.3 The transpose memory

The transpose memory shall:

- be designed as a Verilog module, with the interface shown in Listing 4.3.
- hold an $8 \times 8 \times 12$ bit image.
- allow writes of rows and reads of columns in one clock cycle.
- have no address inputs.

Listing 4.3: Transpose memory interface

```

module transpose(input clk , rst , wr , rd ,
  input [95:0] in , output [95:0] ut );

```

We leave it to the designer to decide how the signals `wr`, `rd` exactly work.

4.2.4 WB memory map

The accelerator shall have the memory map shown in Table 4.1. `csr` is the Control/Status register. The accelerator is started by setting (from WB) bit 0, the START bit. When the accelerator is ready bit 7, the RDY bit, is set. The START bit is cleared by the DCT2 Control Unit and the RDY bit by software.

mem	Base Address	Size	Read/Write
in	0x9600_0000	512 × 32	WO 32 bit
out	0x9600_0800	512 × 32	RO
csr	0x9600_1000	1 × 8	RW 8 bits
dma	0x9600_1800		For use in lab 3

Table 4.1: Accelerator memory map

Preparation task 7

Draw a block diagram of the transpose memory. Rows can be written and columns can be read in 1 clock cycle.

Laboration task 2.1

Design and implement the DCT accelerator with a WB interface.

Laboration task 2.2

Write a testbench for your DCT accelerator.

4.3 Introduction to μ Clinux

In the remaining labs we are going to run μ Clinux on the openrisc system. The most important difference between μ Clinux and Linux is that μ Clinux works without an MMU. This means that there is no memory protection for programs running on μ Clinux. Therefore, extra care must be taken during development since a bug in a program may cause the entire operating system to crash.

You can start μ Clinux on the openrisc system by using the `u` command in the monitor. This will copy a μ Clinux image from the flash memory to the SDRAM and boot μ Clinux. If everything worked you will get a prompt and you should also be able to browse a web page on the μ Clinux machine. The IP address of the μ Clinux machine is printed by the boot script.

On the μ Clinux machine, most directories are read only but `/mnt` and `/var` is writable. `/mnt` is a good directory to download programs to. The base directory for the web server documents is in `/mnt/htdocs`.

Laboration task 2.3

Boot μ Clinux and familiarize yourself with it.

4.3.1 Compiling an application to μ Clinux

In the `hello` directory of the lab skeleton there is a sample hello world application. This has to be cross compiled on one of the Linux machines in the lab. The cross compiler has access to a C library so you can use all standard functions like `printf`, `fopen`, `fread`, etc. If you are interested in how the cross compiler is invoked, you can take a look at the `Makefile`. Just type `make` in the `hello` directory to compile it.

4.3.2 Starting the TFTP server

In order to download applications via tftp we first need to start a TFTP server on one of the Linux computers in the lab. In Linux, this can be started with the following command:

```
/usr/sbin/uftpd -n -o ftp=5051,tftp=5050 ~/tftp
```

This will start a TFTP server listening on UDP port 5050¹. Files will be served from the `tftp` directory in your home directory. The options `--daemon` `--no-fork` are used so that the tftp client can be interrupted with `ctrl c`. (We don't want any TFTP

¹Port 69 is actually the standardized TFTP port but non privileged users in Linux are not permitted to open ports below 1024 so we decided to use port 5050 instead.

servers to be left after you log out since this would prohibit other lab groups from starting a TFTP server.) The ip-number of the tftp server is found by giving the linux command:

```
/sbin/ifconfig | grep 192
```

This asks for information about all connected networks, and only prints the ip-number of the network starting with 192. The local network between the μ Clinux and the Linux computer are all use a 192.168.0.xx number.

4.3.3 Downloading applications via TFTP

In order to download and run the hello application we must use tftp. First, hello has to be copied to the tftp directory in your home directory. After that you can write the following commands in μ Clinux (remember to replace 192.168.0.62 with the ip-number of the current Linux computer):

```
/> cd /mnt
/mnt> tftp 192.168.0.62
tftp> get hello
Received 28664 bytes in 0.8 seconds
tftp> quit
/mnt> chmod 755 hello
/mnt> hello
3
2
1
Hello uClinux!
/mnt>
```

Laboration task 2.4

Download and test hello.

It can be noted that tftp sometimes says “Not a typewriter” and aborts the transfer. This has not been fully debugged yet unfortunately. If it happens to you, just try again, it rarely happens twice in a row.

4.4 Introduction to jpegfiles

In this lab series we will be using and enhancing a library written originally by the Independent JPEG Group. (IJG)

The software package has been somewhat modified by us for the TSEA44 course. First of all, we have removed a lot of files that are not needed for a μ Clinux target (configuration files and Makefiles for other platforms, etc). Some of the more interesting functions have been instrumented with performance counters in order to measure how much of the CPU time is spent in these functions. Finally, we have modified the DCT handling code to correspond to the verilog source code for the 1D DCT which is used in the lab skeleton.

4.4.1 Important files in the lab skeleton

In this section we describe a number of important files that you will need to look at in this lab.

- `Makefile` Contains the build instructions. If you need to modify the compilation flags, this is the file to look inside.
- `jpegtest.c` contains the test program we will use
- `testbild.raw` is a grayscale image in raw format.
- `perfctr.c,perfctr.h` This is the place to look if you want to add a new performance counter
- `jdctmgr.c` Contains the main computation loop and definitions of static variables. Also contains the *forward_DCT* function which calls the 2D DCT kernel and does the quantization.
- `jdct.c` Contains the 2D DCT kernel
- `jchuff.c` Contains the Huffman and RLE encoder.
- `webcam.c` Another test application we will use

Below is a call graph of the important functions called by *jpegtest*:

```

main()                (jpegtest.c)
  +-- draw_image()    (jpegtest.c)
  +-- init_encoder()  (jdctmgr.c)
  +-- encode_image()  (jdctmgr.c)
    |  +-- forward_DCT() (jdctmgr.c)
    |  |  +-- jpeg_fdct_islow() (jdct.c)
    |  +-- encode_mcu_huff() (jchuff.c)
    |  +-- emit_bits() (jchuff.c)
  +-- finish_pass_huff (jchuff.c)

```

- *encode_image()* - Creates a buffer for an 8×8 block and calls *forward_dct(8x8 buffer)*, the returned buffer is sent to *encode_mcu_huff(8x8 buffer)*. This will encode the first block of the image and save it in memory. The procedure is then repeated until every block is encoded and then *finish_pass_huff()* is called to write the memory buffer to file.
- *forward_dct()* - Extracts the first 8×8 block from the image and then runs the DCT on this block. The result is returned to *encode_image()*.
- *encode_mcu_huff()* - Uses a predefined Huffman code to compress the data returned from *forward_dct()* and sends the Huffman codes to *emit_bits()*.
- *emit_bits()* - Recieves Huffman codes and save them until enough bits to write a byte are received, then a byte is written to *buffer[]*.

- *buffer[]* - Storage for the encoded image memory during operation.
- *finish_pass_huff()* - Calls *emit_bits()* to write leftover bits to *buffer[]* and then calls *write_data()*.
- *write_data()* - Writes the contents of *buffer[]* to file.

Preparation task 8

Take a look at the file containing the 2D DCT kernel and figure out how to change it to use your 2D DCT hardware.

4.4.2 The jpegtest application

This is the main test application we are going to use in the lab series. It will first read a raw picture from a file named `testbild.raw`, encode it to JPEG format and write it to an output file which you specify on the command line. It will also output performance data on how many clock cycles some important functions consumed. In order to see the encoded image you can place it in the `/mnt/htdocs` directory and download it to your computer via the web server on the μ Clinux machine.

Laboration task 2.5

Download and test the `jpegtest` application. Both with and without the `testbild.raw` program.

4.4.3 The webcam application

The lab skeleton also includes a simple webcam application. You can download `webcam.cgi` to `/mnt/htdocs/cgi-bin` and look at the webcam via the web browser.

Laboration task 2.6

Modify `jpegfiles` to use your 2D DCT hardware and test it by using `jpegtest` and `webcam.cgi`. The results should be exactly the same as if you were using the software only version.

4.5 Timestamps

The size of the test image `wunderbart.jpg` is 512×400 pixels, or $64 \times 50 \times 8 \times 8$ -blocks. The Figure 4.3 shows the result of collecting timestamps in the beginning of block row 50.

The order of the operations is

1. read a block from DRAM (w/o DMA)
2. calculate DCT on the block (w/o HW DCT)
3. quantize the block (w/o HW Quantization)
4. readout of the block

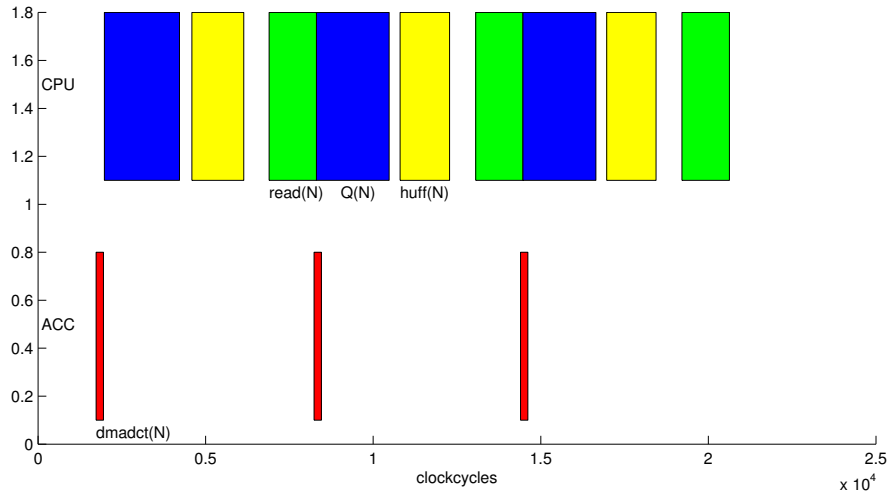


Figure 4.3: Timestamps for JPEG compression pipeline. Each color coded patch represents the processing of an 8×8 -block. Colorcodes: DMA+DCT red, readout green, quantization blue and Huffman encoding yellow.

5. Huffman encoding of the block (w/o special instruction)

In Figure 4.3 we have implemented all the HW accelerators except the quantization step. It is evident that the design will gain a lot from the acceleration of the quantization step. Putting more work into the DCT-accelerator is wasted hardware. The DCT-accelerator will always be overlapped by (typically) the Huffman encoding.

Timestamps can be collected by using

```
timestamp = gettimer();
```

`gettimer` is a macro defined in `perfctr.h` in `jpegfiles`.

The completion of the HW DCT operation can not be determined by software. Instead you can include a 10-bit counter (for instance) in the accelerator. The counter can be mapped into the free bits of the control register.

4.6 Quantization

4.6.1 General

The standard JPEG quantization table for the luminance channel is given by

$$Q_L[u, v] = \begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix}. \quad (4.1)$$

To get higher image quality this table is divided by 2. Reciprocals are then computed by the formula

$$R[u, v] = \frac{2^{14}}{Q_L[u, v]/2}, \quad (4.2)$$

which is the table in the file `jpegdctmgr.c`.

As an example we use $A[u, v]$ from (3.6). Before quantizing $A[u, v]$ we subtract $64 \times 128 = 8192$ from the DC value $A[0, 0]$, which corresponds to taking the DCT of $a[x, y] - 128$. This procedure is meant to give a smaller value on the average, which however is false in this particular case.

Finally we compensate for the scale factor 8 introduced in the DCT2 step by right-shifting 17 steps:

$$Y[u, v] = \text{round} [(A[u, v] - 8192 \cdot \delta[u, v]) \cdot R[u, v] \cdot 2^{-17}]$$

$$= \begin{pmatrix} -96 & -3 & 0 & 0 & 0 & 0 & 0 & 0 \\ -24 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad (4.3)$$

which left only four non zero coefficients.

4.6.2 Design of a hardware accelerator for quantization

The following piece of code in `jpegdctmgr.c` in `jpegfiles` calculates the quantization of one block. Instead of division multiplication with the reciprocal is used.

Listing 4.4: C code for quantization in `jpegdctmgr.c`.

```

int reciprocals [] = {2048, 2979, 3277, 2048, 1365, 819, 643, 537,
                      2731, 2731, 2341, 1725, 1260, 565, 546, 596,
                      2341, 2521, 2048, 1365, 819, 575, 475, 585,
                      2341, 1928, 1489, 1130, 643, 377, 410, 529,
                      1820, 1489, 886, 585, 482, 301, 318, 426,
                      1365, 936, 596, 512, 405, 315, 290, 356,
                      669, 512, 420, 377, 318, 271, 273, 324,
                      455, 356, 345, 334, 293, 328, 318, 331};

...

for (i = 0; i < DCTSIZE2; i++) {
    rval = reciprocals[i]; // 16 bit
    signed temp = workspace[i]; // 16 bit signed

    temp = temp*rval;

```



```
    if (temp & 0x10000) {
        temp = temp >> 17;
        temp += 1; }
    else
        temp = temp >> 17;

    coef_block[i] = (short) temp; // 16 bit signed }
...

```

Preparation task 9

Design a HW Quantization unit (Q2 in Figure 4.1), that calculates exactly the same values as the code in Listing 4.4. Instantiate your Q2 unit and modify the code in listing 4.4. We propose that Q2 should be able to quantize 2 numbers per clock cycle.

4.7 Tips and tricks

In this section we have collected some notes that you might find useful.

- If you want to simulate the 2D DCT accelerator together with the rest of the system, you will have to modify the monitor to run your testcode right after the system has started. See the directions in section 3.3.3 on how to modify the monitor.
- In order to improve the performance of the code you can remove some of the performance counters. But the performance counters in jpegtest has to remain!
- If you encounter some weird problems with the hardware you can try to turn the power off to the FPGA system before configuring it. We have had some problems with the FPGA board which can be solved in this manner.

4.8 What to include in the lab report

The lab report should contain all source code that you have written. (The source code should of course be commented.) We would also like you to include a block diagram of your hardware. If you have written any FSM you should include a state diagram graph of the FSM.

We would also like you to discuss the following questions in detail somewhere in your lab report.

- How does your 2D DCT hardware work?
- How did you verify that your 2D DCT hardware works correctly?
- What is the performance with and without the 2D DCT hardware? This should include measurements of both the 2D DCT kernel and the entire application.
- A timestamp diagram.

- How much of the FPGA is used by the 2D DCT hardware?
- How much is the 2D DCT hardware used while encoding an image in `jpegtest`?
- Is the size of the 2D DCT hardware justified by the performance improvements?
- What would be required in order to implement more functionality like zigzag addressing in the 2D DCT hardware module? Would it be difficult to modify `jpegfiles` to take advantage of such optimizations?

And of course, the normal parts of a lab report such as a table of contents, an introduction, a conclusion, etc. The source code that you have written should be included in appendices and referred to from the main document.

Chapter 5

Lab task 3

5.1 DMA in the DCT Accelerator

In this lab we will improve the DCT accelerator by using DMA. You can find a specification on how to do DMA in wishbone in appendix B.

5.1.1 Proposed architecture

In this lab we will modify the DCT accelerator created in lab 2 to use DMA. In this case the idea is that the DMA module will feed the DCT accelerator with data from the system memory but the CPU is still responsible for reading the data from the DCT accelerator. This means that the changes in `jpegfiles` will be kept to a minimum. The only changes will be to initialize the DMA as early as possible and to change `jpegfiles` to not write data to the DCT accelerator.

There are of course a wide variety of ways to do this but we propose that the accelerator should use the following interface:

0x9600_1800: *SRCADDR*, the address of the grayscale image we want to convert.

0x9600_1804: *PITCH*, the width of the image in bytes.

0x9600_1808: *ENDBLOCK_X*, the width of the image in macroblocks minus one.

0x9600_180c: *ENDBLOCK_Y*, the height of the image in macroblocks minus one.

0x9600_1810: *CONTROL* (When reading) Bit 0 indicates that the DMA is not idling, bit 1 indicates that a DCT operation for one block has been finished.

0x9600_1810: *CONTROL* (When writing) Writing a 1 to bit 0 starts the DMA FSM whereas writing a 1 to bit 1 tells the accelerator that the processor has read the result of one block and the DMA accelerator may proceed with the next block.

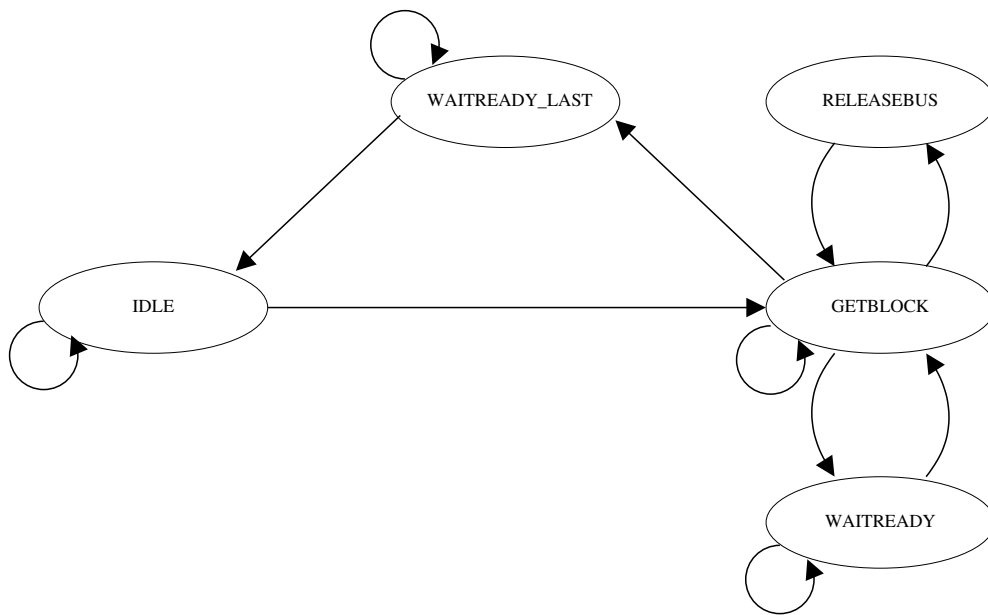


Figure 5.1: The proposed state diagram for the DMA accelerator.

In Figure 5.1 there is a state diagram which is suitable for the DMA accelerator. The states are described below:

- **IDLE:**
The DMA module is not doing anything.
- **GETBLOCK:**
The DMA module is fetching an 8x8 block. Once the block is fetched we go to the **WAITREADY** state and starts the DCT transform.
- **RELEASEBUS:**
The DMA accelerator has to release the bus regularly so that other components can access it. You should do this for every line of a macroblock that you read.
- **WAITREADY:**
In this state we wait until the program tells us that it has read the result of the transform by writing to the control register.
- **WAITREADY_LAST:**
Same as **WAITREADY** except that we go to the **IDLE** state when done.

5.1.2 jpeg_dma.sv

The interface to the jpeg_dma module consists of a wishbone master interface, a simplified wishbone slave port and some control signals that are connected to the DCT accelerator in the jpeg_top module. A simplified view of the proposed architecture is shown in figure 5.2. Some of the most important signals are described below:

- **dma_bram_data:**
The data we want to write to *inmem* in *jpeg_top*.
- **dma_bram_addr:**
The address we want to write the data to.
- **dma_bram_we:**
The write enable signal for *inmem*.
- **dma_start_dct:**
When this clock signal is high for one clock cycle, the DMA accelerator will start to transform the current block in *inmem*.
- **dct_busy:**
This input signal is high from one clock cycle after *dma_start_dct* has been activated to the moment all results have been written to *outmem*. (You have to make sure that your DCT accelerator busy signal actually works like this.)

The interface to the address generator is as follows:

- **resetaddr_i:**
The address generator sets the address to the start address specified in *dma_srcaddr*.
- **incaddr_i:**
The address generator will increase the address to the next word we need to read.
- **address_o:**
The current address of the image.
- **endframe_o:**
This signal is active when the current address is the last address of the image.
- **endblock_o:**
This signal is active when the current address is the last address of an 8x8 block.
- **endline_o:**
This signal is active when the current address is the last address of a line of an 8x8 block.
- **dma_srcaddr, dma_pitch, dma_endblock_x, dma_endblock_y:**
As described above.

There are also some other important signals that are available such as:

- **startfsm:**
This signal is active when the CPU has written a 1 to bit 0 of the *CONTROL* register.
- **startnextblock:**
This signal is one when the CPU writes a 1 to bit 1 of the *CONTROL* register.

It is important to note that the DCT accelerator should still work as before if DMA is not in use. But you are (of course) allowed to assume that the accelerator will be used either in DMA mode or regular mode at one time. This means that the results when both the DMA module and a wishbone master tries to write to *inmem* in *jpeg_top* at the same time are allowed to be undefined.

You should mainly modify *jpeg_dma.sv* in this lab but you will also have to modify *jpeg_top.sv* to account for the *dma_bram_* signals. In *jpeg_dma.sv* some code already exists, for example, an address generator capable of generating the addresses required to read in the frame in 8×8 blocks is already instantiated. The writing and reading of control registers and status registers are also complete. Your main task is to complete the finite state machine.

In order to simplify debugging of your DMA accelerator it is useful to make sure that you can change the DMA accelerator state to IDLE by writing to a control register in the accelerator. This means that you don't have to reset your system if *jpegtest* aborts in the middle of a frame.

Preparation task 10

Complete as much as possible of the FSM in Figure 5.1 by filling in the conditions for the state transitions. (Hint: The lab skeleton already has code for some of the state transitions.)

Laboration task 3.1

Modify the DCT accelerator in lab 2 to use DMA.

Laboration task 3.2

*Test your DCT accelerator in modelsim by modifying the monitor to use your accelerator. Some initial code is available in *dma_dct_hw()* in *hw/monitor/firmware/src/dct2.c*.*

5.1.3 How to use DMA in jpegfiles

You will have to make a few changes to *jpegfiles* in this lab. You should modify both *jdct.c* and *jdctmgr.c* so that no values are copied from memory to your DCT accelerator. You should also note that there is a level conversion of the indata in *jdctmgr.c*, that is the pixels are modified by subtracting 128 from them. You will have to take this into account by modifying your hardware.

5.1.4 Cache coherency issue

Normally, once DMA is involved in a system it is very important to take cache coherency issues into account. Especially if the processor has no bus snooping facility to update the caches automatically. However, since the OR1200 has write through caches and the DMA accelerator only reads from memory we will not have any cache coherency issues.

Laboration task 3.3

*Modify *jpegfiles* to use the DMA enabled DCT accelerator.*

5.2 What to Include in the Lab Report

The lab report should contain all source code that you have written. (The source code should of course be commented.) We would also like you to include a block diagram of your hardware. If you have written any FSM you should include a state diagram graph of the FSM.

We would also like you to discuss the following questions in detail somewhere in your lab report¹:

- How does your hardware work?
- How did you verify that your hardware worked?
- How did you modify the software?
- A timing diagram.
- What is the utilization of your accelerator?
- What is the performance of jpegtest with DMA enabled?
- How long does it take (on average) to read a macroblock into the DCT accelerator via DMA?
- How much is the wishbone bus used by the DMA unit and how much is the bus used by the CPU?

And of course, the normal parts of a lab report such as a table of contents, an introduction, a conclusion, etc. The source code that you have written should be included in appendices and referred to from the main document.

¹In order to answer some of these questions you will probably have to modify the performance counters you added in lab 1 and add support for (at least) the DMA bus master to it.

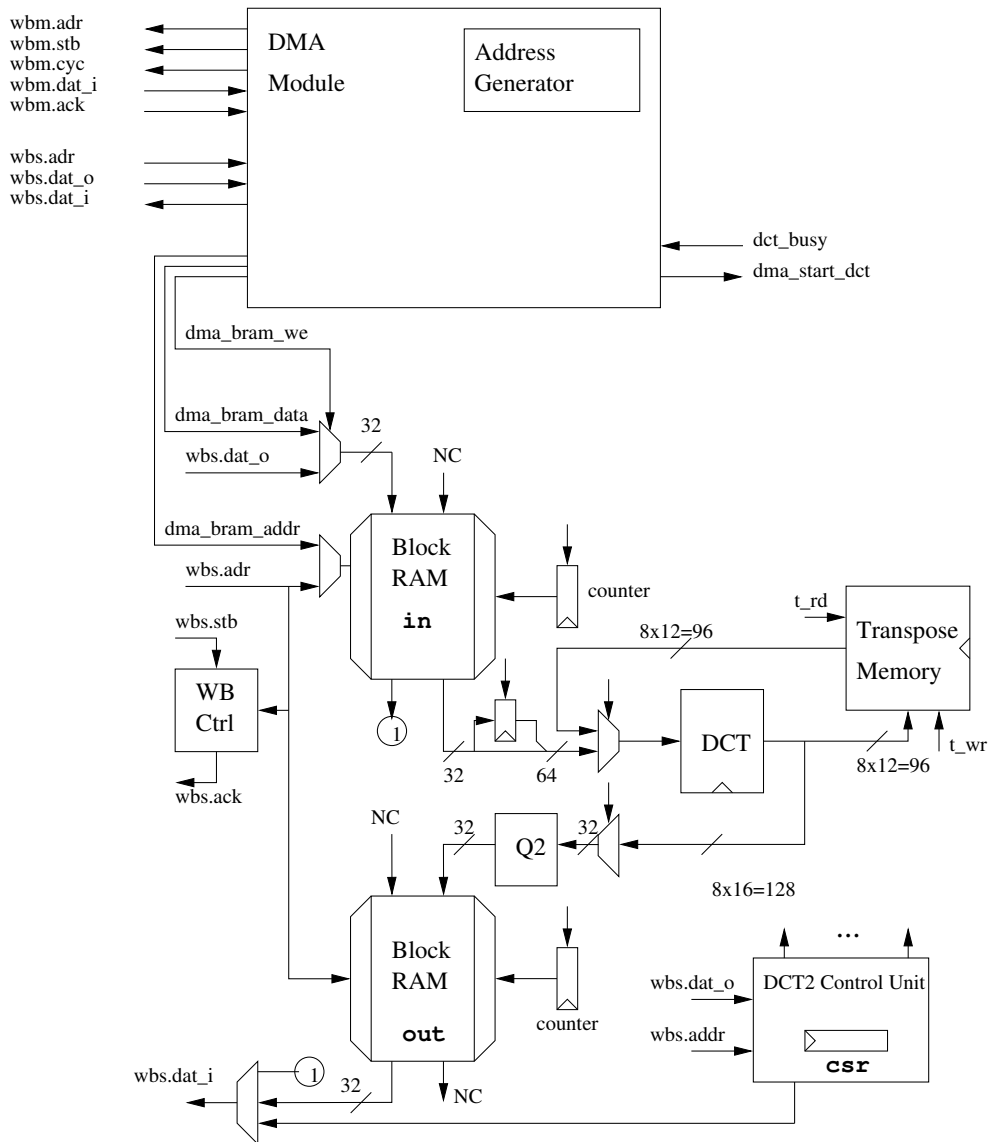


Figure 5.2: The proposed architecture for the DMA based DCT accelerator.

Chapter 6

Lab task 4 - Custom Instructions

6.1 Introduction

In this lab task you will learn how to design and integrate a new instruction into the processor. This part will target the bit alignment problem arising when trying to write bit streams to the memory. A bit stream is here defined as a stream of bits with no particular memory alignment. In accordance with jpegfiles the bit pattern to write to the stream will be called the *code* and the number of bits to write the *size* of the code. The extension you will make to the instruction set is called a Variable Length eXtension, or VLX for short.

6.1.1 Huffman Coding

First a short review of the principles of Huffman encoding. Huffman encoding works by encoding the most frequently occurring item with the fewest number of bits and then using more bits for items occurring less frequently. For instance take the sequence ABAACDAAAB, the letter's frequencies are calculated and listed in 6.1

Four different values imply that we at least need two bits per value thus we would need $10 \cdot 2 = 20$ bits to encode the string. If we were to use the Huffman codes in 6.1 we would need only $6 \cdot 1 + 2 \cdot 2 + 3 + 3 = 16$ bits, thus saving four bits. How the Huffman codes are calculated is not in the scope of this course, for further information about Huffman coding see for example [14].

When the Huffman tables are calculated the encoding process is simple, just do a lookup and replace, e.g. simply replace the data with its corresponding code.

In our case jpegfiles uses already pre calculated Huffman tables, the JPEG standard

Letter	Frequency	Huffman code
A	6	0
B	2	10
C	1	110
D	1	111

Table 6.1: Letter frequencies and Huffman codes

suggests four Huffman tables. Two for the luminance, one for the DC values and one for the AC values. And two for the chrominance channel one for the DC values and one for the AC values. the size of the Huffman codes in a JPEG stream varies from 1 to 16 bit.

6.1.2 The Problem

Although the lookup and replace is an easy operation the problem arises when we want to write the bit stream to the memory. A consecutive stream of codes varying in length from one 1 to 16 bits has to be written to memory.

In this lab your task is to design and add new hardware to the processor that can be used by an instruction to ease the bit stream writing, the idea is that it can be used as a normal store operation where the length of the data to be stored can be specified and the data stream will be packed in memory, i.e. each Huffman code will follow directly after the previous Huffman code.

Preparation task 11

Suggest a method to handle the bit field writing in software. Hint, look into the `jchuff.c` file in `jpegfiles`.

Preparation task 12

Suggest a method to handle the bit field writing in hardware.

6.2 Adding a New Instruction

In order to add a new instruction we first need to find a suitable instruction to use for our purpose. Looking through the open RISC instruction set we find that there are some instructions used for storing double words (64-bits) and we can thus “kidnap” one of these instructions, *Lsd* is a suitable instruction to use. Of course if you feel like recoding the assembler you can name your instruction whatever you feel like. How you use this instruction is up to you, remember that for one `vlx` operation you have to specify the code and the size of the code.

6.2.1 Making the Processor Understand

In order to add a new instruction we need to make the processor aware of the instruction we intend to add. The first stage is to make some modifications to the instruction decoder. In the file `or1200_ctrl.sv` you will find a number of *`ifdef OR1200_SBIT_IMPL`* preprocessor directives. Inside some of these you will find a comment asking you to write some code there. Find these places and add the missing code. Once this is done correctly you should be able to use the *Lsd* instruction without causing an illegal instruction exception but it will behave as a `nop` instruction.

6.2.2 Adding Special Purpose Registers

The hardware you are going to construct will use a number of special purpose registers to maintain observability and make it possible to handle some unusual situa-

tions in software. We will add a new group of special purpose registers to the processor, therefore we must make some changes in the *or1200_sprs* module. In the file *or1200_sprs.sv* you will find a `'ifdef OR1200_SBIT_IMPL` preprocessor directive. Add the missing code inside this directive. Once this is done you should be able to use the special purpose registers since they have already been implemented for you.

6.2.3 Adding the Required Hardware

Finally you must add the hardware using your instruction. The main part of the work is to extend the load and store unit described in the *or1200_lsu* module. In the file *or1200_lsu.sv* you will find a number of `'ifdef OR1200_SBIT_IMPL` preprocessor directives. Inside some of these you will find a comment asking you to write some code there. Find these places and add the missing code. In addition to this you will find the *or1200_vlx_top* module instantiated. This is the top module for the hardware handling the stream writing. The code for this module can be found in the *or1200_vlx_top.sv* file. Further instructions on how this module should be implemented can be found in section 6.3 and section 6.4.

6.3 Proposed Architecture

We propose the general architecture shown in figure 6.1. Gray arrows represent control signals and black arrows represent data.

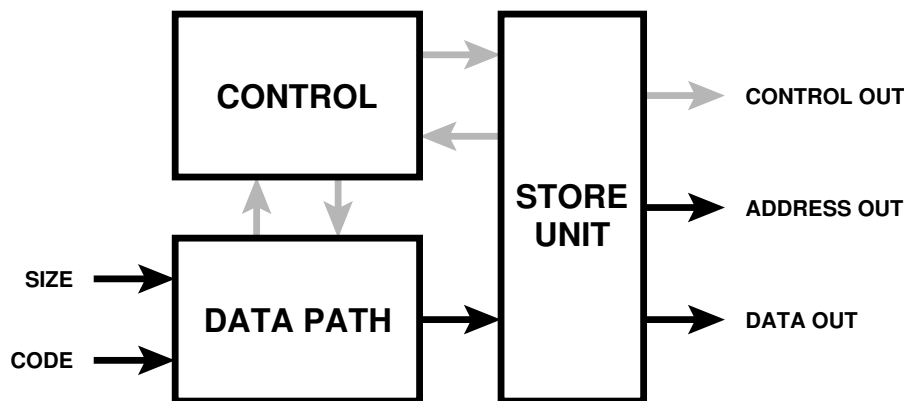


Figure 6.1: Proposed architecture for the set bit instruction.

6.3.1 Control Unit

The control unit should be defined in the *or1200_vlx_ctrl* module, a stub can be found in file *or1200_vlx_ctrl.sv*. The control unit's purpose is, as it sounds, to control the other units. This unit might include a simple FSM or this unit might not even be needed. All depending on your design.

6.3.2 Data Path

The data path unit should be defined in the *or1200_vlx_dp* module, a stub can be found in file *or1200_vlx_dp.sv*. The data path unit should contain all logic relevant for the data manipulation needed. This unit will be responsible for aligning and merging the incoming bits with the bits previously stored. Different solutions are possible, either taking one or several clock cycles. This is basically a trade off between speed and hardware. It is your task to implement this module of course in conjunction with how you implemented the control unit.

6.3.3 Store Unit

The store unit should be defined in the *or1200_vlx_su* module, a stub can be found in file *or1200_vlx_su.sv*. The store unit is responsible for storing a full bit vector to memory. A suggested solution is to take a 32 bit word and store it to main memory one byte at a time, doing the zero stuffing if necessary. Remember that a *0xFF* byte has to be stored as two bytes *0xFF 0x00*.

It is of course faster to store the data to the memory 32 bits at a time but keep in mind that nothing more than byte alignment is ensured by jpegfiles.

Important!

This is worth repeating, jpegfiles only ensures byte aligned data.

If you do not understand the implications of this, consult some one in the course staff before starting to implement your VLX unit.

6.3.4 Multi Cycle Instructions

Some instructions might be multicycle instructions, your solution will need to stall the processor at certain points, at least when the bit buffer is stored to memory. This is done by setting the *stall_cpu_o* signal high in the *or1200_vlx_top* module. When high it will stop the cpu from fetching new instructions. This is useful, and so used, when executing an instruction with indeterminable execution time, such as writing or reading from memory.

6.3.5 Instruction Details

Although a lot of information about how the instruction is supposed to work is given in this lab manual you have to work out the finer details yourself. A good place to start is to look into the *jchuff.c* file. Also all other place in the code surrounded by the *#ifdef HW_INST* (grep is a good tool) can be of interest. Also remember to design for test, the more status you can view of your hardware when it is running for real, the easier it will be for you to debug it if something fails.

6.4 Hardware Implementation

In this lab task you are supposed to construct much of the hardware yourself. You have to modify a small part of the *or1200_lsu* module (Load Store Unit). Three special purpose registers are already present in the VLX hardware, the *bit_reg* register, the *bit_reg_wr_pos* register, and the *vlx_add_o* register. They are all mapped to the special

purpose register space in group 24 (the group is selected with bit 15 - 11 of a special purpose register address), the three addresses for the three special purpose registers in the vlX unit is shown in table 6.2. If you feel like adding more special purpose registers you are free to do so, you are also free to remap the special purpose registers, e.g. if you want to use a 64 bit buffer instead of intended 32 bit version. Although if you remap the special purpose registers you have to specify the new mapping in your lab report. In case you do a remapping of the special purpose registers, you have to map your new registers to group 24 (or any other group free for custom use see [6], but this requires more extensive modifications in other modules of the processor).

Register Name	Register Function	SPR address
bit_reg_wr_pos	next free bit position in the bit buffer	0xc000
bit_reg	buffers 32 bit, written to memory when full	0xc001
vlx_addr_o	next address to write a full buffer to	0xc002

Table 6.2: VLX register specification

Write your RTL code inside the `'ifdef OR1200_SBIT_IMPL ... 'endif` regions, this makes it easy for you to identify your code and you can also easily remove your hardware from the design.

6.4.1 Constructing the Hardware

Some general advise, worth repeating, when building the hardware is to always know what you are building when writing the Verilog code. In general, you should have a clear picture of the RTL design, before you start writing the HDL code.

A good design methodology is to do hierarchical design, i.e. create and instantiate a number of modules. Each module can then be tested with a separate test bench before integrating them. You are encouraged to use the suggested architecture discussed in section 6.3 but of course you are free to come up with an architecture of your own.

6.5 Software Implementation

When you think that your hardware works it is time to put the new instruction to use.

6.5.1 Running the Instruction

If you have built the hardware correctly it should now be possible to use the new instruction in your program. However the compiler does not know anything about the new instruction. So either you have to modify the compiler or you can use in-line assembler in your C program.

The in-line assembler directive allows you to insert assembly code verbatim into the object file. A short example is shown in Listing 6.1. This and the rest of the text will show you what you need to know for this task.

Listing 6.1: Inline assembler example.

```
asm volatile(assembler template
              : output operands
              : input operands
              : list of clobbered registers
              );
```

The **asm volatile** keyword instructs the compiler to insert exactly this assembler instruction at exactly this position into the compiled code.

The first part of the construct is an assembler template string, e.g. **"l.sd 0x0(%0), %1"**.

The output operands is a comma separated list of output operands, the one you most probably are going to use is **"=r"(var)** meaning that the output from the assembly instruction is a write only to a register operation and that the the variable **var** in the C code is assigned the output value.

The input operand list is a comma separated list of input operands. The format you most probably are going to use is the **"r"(var_in)**. This means that the variable **var_in** is assigned to a register used in the assembly instruction.

The input and output operands are mapped, by order of appearance, to the **"assembler template"** string. E.g. **"lor %0,%1,%2" : "=r"(res) : "r"(in_a), "r"(in_b)** will map the **res** to be the destination (**%0**), **in_a** source one (**%1**), and **in_b** source two (**%2**).

The list of clobbered registers is a comma separated list of registers that have been modified by the assembly and not assigned by the compiler with the **%** notation. I.e. if you have written **asm volatile("lori r20,r10,r11")** **r20** has to be listed as a clobbered register like this **asm volatile("lori r20,r10,r11"::"r20")**. This since the compiler must be informed if we modify some registers in the inline assembly instruction, otherwise it will happily assume that the value of a register is the same after the inline assembly block as it was before the section in question.

If you want to write a consecutive block of assembler code this is done as shown in Listing 6.2.

Listing 6.2: Consecutive block of inline assembler.

```
asm volatile ("l.instr_0,2,3" "\n\t"
              "l.bnf_label_1" "\n\t"
              "l.nop_0x0" "\n\t"
              "l.movhi_1,4" "\n\t"
              "l.j_label_2" "\n\t"
              "l.nop_0x0" "\n\t"
              "label_1:" "\n\t"
              "l.movhi_1,5" "\n\t"
              "label_2:" "\n\t"
              : "=r" (res1), "=r" (res2)
              : "r" (arg1), "r" (arg2), "i" (0x1), "i" (0x0));
```

To omit the **"\n\t"** is not a syntactic error, but if not used the disassembled object file will look messed up. We can also see from Listing 6.2 that it is possible to use labels in inline assembler constructs.

For a more detailed description of the gcc in-line assembly see [15]

6.5.2 Integration into jpegfiles

When you find that the hardware is working and you have written some test programs to verify that the the processor can execute your instruction. You are ready for the next step, to get the instruction to work for you in jpegfiles. You need to add code for three phases of operation, as described below. The only file you need to modify is `jchuff.c`, look for the `#ifdef HW_INST` blocks.

Phase 1: VLX Initialization

This, the first, phase is executed before processing of the bit stream starts. In this phase you must ensure that your hardware is properly initialized. It is important to set the address where your hardware shall start writing the bit stream to memory.

Phase 2: VLX Operation

This, the second, phase is executed as long as there is new data to encode. This phase is rather simple, just use your hardware to write the incoming bit stream to memory.

Phase 3: VLX Finalization

This, the third and final, phase is executed when there are no more data to encode. In this phase you must ensure that data not yet written to memory, still residing in your hardware, is correctly put in the memory. Also you must ensure that you make a clean hand off to jpegfiles so it can take care of some finalization operations, like writing the JPEG end marker. Take note of the `next_buffer` variable and think about what you have to do with it.

6.5.3 JPEG Markers

There are a number of bit strings called markers in the JFIF binary file. A marker is always byte aligned and starts with the byte `0xFF`, the next byte specifies what marker it is. For this reason if `0xFF` is encountered in the scan data, `0xFF` must be written as `0xFF00`. Some markers are stand alone markers, meaning that the marker has no additional information associated with it. Otherwise the marker has a two byte code immediately following the marker specifying the length in byte of extra data associated with the marker. A marker followed by data is referred to as a marker segment. Table 6.3 list a few important markers, see [17] for more information about markers and [18] for some specific restriction of markers in JFIF.

Laboration task 4.1

Implement the new instruction in hardware and show that you can use the new instruction using inline assembler in C-code.

Laboration task 4.2

Modify the jpegfiles code to use your new instruction.

Code	Name	Explanation	Stands Alone
0xFFC0	SOF ₀	Start of frame for baseline coded pictures.	No
0xFFE0	APP ₀	Application specific data used by JFIF.	No
0xFFDA	SOS	Start of scan. The image data starts after this marker segment.	No
0xFFD _n	RST _n	Restart marker $n(n = 0, 1...7)$, restart decoding after this marker.	Yes
0xFFD9	EOI	End of image, data after this marker is ignored.	Yes

Table 6.3: Important JFIF markers

6.6 Important Files For this lab task

In addition to the files listed in section 4.4.1 also the following files are important.

- `or1200_1su.sv`: Load Store Unit, the module used by the or1200 processor to load/store from/to memory.
- `or1200_defines.v`: Defines for the or1200 RTL code, look for `ifdef OR1200_SBIT_IMPL`.
- `or1200_vlx_top.sv`: A skeleton top level module for the set bit instruction.
- `or1200_ctrl.sv`: The instruction decoder.
- `or1200_sprs.sv`: Special purpose register control.
- `jchuff.c`: Code dealing with the Huffman-encoding.

6.7 Tips and tricks

In this section we have collected some notes that you might find useful.

- If you want to simulate the set bit instruction together with the rest of the system, you will have to modify the monitor to run your test code right after the system has started. See the directions in section 3.3.3 on how to modify the monitor.
- In order to improve the performance of the code you can remove some of the performance counters. But all performance counters in `jpegtest.c` has to remain!
- How to write to a special purpose registers with address `0xC000` in C-code:

```
asm volatile("l.mtspr %0,%1,0x0" : : "r"(0xc000), "r"(value))
```


- How to read from a special purpose registers with address `0xC000` in C-code:
`asm volatile("l.mfspr %0,%1,0x0" : "=r"(result) : "r"(0xc000))`
- If you are writing inline assembler with more than one instruction you should read the following paragraph for information about input and output operands[16]:

Unless an output operand has the `&` constraint modifier, GCC may allocate it in the same register as an unrelated input operand, on the assumption the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use `&` for each output operand that may not overlap an input.

6.8 What to Include in the Lab Report

The lab report should contain all source code that you have written. (The source code should of course be commented.) We would also like you to include a block diagram of your hardware. If you have written any FSM you should include a state graph of the FSM.

We would also like you to discuss the following questions in detail somewhere in your lab report:

- How does your hardware work?
- How did you verify that your set bit hardware worked?
- What is the performance with and without the set bit hardware? This should include measurements of both the entire application and the set bit instruction by itself, assuming good code in a software implementation (take a look at how the software solution in jpegfiles).
- How much of the FPGA does your hardware use?
- How would your design change if you had to achieve even higher speed using more hardware?
- How would your design change if you had to use less hardware at the cost of a slower solution?
- What are the problems with using your new hardware in a multitasking operating system? How can the problem(s) be solved?
- What is the performance of your final system?
- What was the hardest problems you encountered during the entire lab course?

And if you want to, we would appreciate some comments on the following questions, either in the lab report or by some other means of communications:

- What did you think of the TSEA44 course?
- What was good?

- What was bad?
- What can we improve for the next year?
- Do you have any other ideas for this course?
- Did you feel that you learned anything of value?
- Any other comments you may have.
- A rough estimation of time spent on the lab tasks.

And of course, the normal parts of a lab report such as a table of contents, an introduction, a conclusion, etc. The source code that you have written should be included in appendices and referred to from the main document.

6.9 Beyond tsea44

If you found this course fun and interesting you can probably find an interesting master thesis project with us. We have master thesis projects in the following areas:

- Application, Algorithm and Architecture
 - Wireless Baseband Processing
 - Internetworking (Network Processor and Security Issues)
 - Multimedia (Audio, Image and Video Processing)
 - Computer Graphics
- Processor and Circuit Design
 - General DSP Processor
 - Multicore SoC Platform
 - Network-on-chip (NoC)
 - High Performance Circuit
- SW/HW Codesign of Embedded System
 - Scheduling
 - RTOS for Embedded SoC
 - IDE and other toolkit for DSP
 - Design Methodology etc.

If any of this sounds interesting, or if you have a proposal for a master thesis project which might fit into our research areas, feel free to contact us to get more information.

Bibliography

- [1] *Xilinx Virtex-II Development Kit*, www.avnet.com
- [2] *Communications/Memory Module User's Guide*, www.avnet.com
- [3] Per Karlström, Mikael Andersson: Parallel JPEG Processing with a Hardware Accelerated DSP Processor, *LITH-ISY-EX-3548-20004*
- [4] Loeffler, Ligtenberg and Moschytz: Practical Fast 1-D DCT Algorithms with 11 Multiplications, *ICASSP-89*, pp. 988-991
- [5] *Virtex-II Platform FPGAs: Complete Data Sheet*, www.xilinx.com
- [6] *OpenRISC 1000 Architecture Manual*, www.opencores.org
- [7] Lampret, D: *OpenRISC 1200 IP Core Specification*, www.opencores.org
- [8] *OpenRISC 1200 RISC/DSP Core*, www.opencores.org
- [9] *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, www.opencores.org
- [10] Mohor, I: *Ethernet IP Core Specification*, www.opencores.org
- [11] Herveille, R: *VGA/LCD Core v2.0 Specifications*, www.opencores.org
- [12] Wiklund, D: *DVGA Core Specifications v1.0*, <http://www.da.isy.liu.se/courses/tsea02>
- [13] Gorban, J: *UART IP Core Specification*, www.opencores.org
- [14] Miano, J: *Compressed Image File Formats: JPEG, PNG, GIF, XBM, BMP*, Addison-Wesley, 1994
- [15] Sandeep, S: *GCC-Inline-Assembly-HOWTO*, <http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>
- [16] FSF: *GCC Manual, Extended Asm*, <http://gcc.gnu.org/onlinedocs/gcc-3.2.3/gcc/Extended-Asm.html#Extended%20Asm>
- [17] ITU T.81: *JPEG compression*, <http://www.w3.org/Graphics/JPEG/itu-t81.pdf>
- [18] JFIF File Format: *JFIF specification*, <http://www.w3.org/Graphics/JPEG/jfif3.pdf>

Appendix A

Open RISC Reference Platform

This is the ORP standard memory map. The actual memory map for our system is in section 1.4.1.

A.1 Address map

Start	End	Cache	Size	Description
F0000000	FFFFFFFF	Y	256 MB	ROM
C0000000	FFFFFFFF	Y	768 MB	Reserved
B8000000	BFFFFFFF	N	128 MB	Reserved for custom devices
A6000000	B7FFFFFF	N	288 MB	Reserved
A5000000	A5FFFFFF	N	16 MB	Debug 0-15
A4000000	A4FFFFFF	N	16 MB	Digital Camera Controller 0-15
A3000000	A3FFFFFF	N	16 MB	I2C Controller 0-15
A2000000	A2FFFFFF	N	16 MB	TDM Controller 0-15
A1000000	A1FFFFFF	N	16 MB	HDL C Controller 0-15
A0000000	A0FFFFFF	N	16 MB	Real-Time Clock 0-15
9F000000	9FFFFFFF	N	16 MB	Firewire Controller 0-15
9E000000	9EFFFFFF	N	16 MB	IDE Controller 0-15
9D000000	9DFFFFFF	N	16 MB	Audio Controller 0-15
9C000000	9CFFFFFF	N	16 MB	USB Host Controller 0-15
9B000000	9BFFFFFF	N	16 MB	USB Func Controller 0-15
9A000000	9AFFFFFF	N	16 MB	General-Purpose DMA 0-15
99000000	99FFFFFF	N	16 MB	PCI Controller 0-15
98000000	98FFFFFF	N	16 MB	IrDA Controller 0-15
97000000	97FFFFFF	N	16 MB	Graphics Controller 0-15
96000000	96FFFFFF	N	16 MB	PWM/Timer/Counter Contr 0-15
95000000	95FFFFFF	N	16 MB	Traffic COP 0-15
94000000	94FFFFFF	N	16 MB	PS/2 Controller 0-15
93000000	93FFFFFF	N	16 MB	Memory Controller 0-15
92000000	92FFFFFF	N	16 MB	Ethernet Controller 0-15
91000000	91FFFFFF	N	16 MB	General-Purpose I/O 0-15
90000000	90FFFFFF	N	16 MB	UART16550 Controller 0-15
80000000	8FFFFFFF	N	256 MB	PCI I/O
40000000	7FFFFFFF	N	1 GB	Reserved
00000000	3FFFFFFF	Y	1 GB	RAM

A.2 Interrupts

Number	Peripheral
0	Reserved
1	Reserved
2	UART16550 Controller
3	General-Purpose I/O
4	Ethernet Controller
5	PS/2 Controller
6	Traffic COP , Real-Time Clock
7	PWM/Timer/Counter Controller
8	Graphics Controller
9	IrDA Controller
10	PCI Controller
11	General-Purpose DMA
12	USB Func Controller
13	USB Host Controller
14	Audio Controller
15	IDE Controller
16	Firewire Controller
17	HDLC Controller
18	TDM Controller
19	I2C Controller, Digital Camera Controller

Appendix B

The Wishbone specification

B.1 Introduction

The Wishbone specification basically dictates the interfaces and how they should behave. The method of connecting the interfaces to each other is very much up to the designer. This chapter contains a brief explanation of the Wishbone specification (which consists of approximately 140 pages). More information can be found in the official specification [9]. Some rules from the specification are cited in the following text.

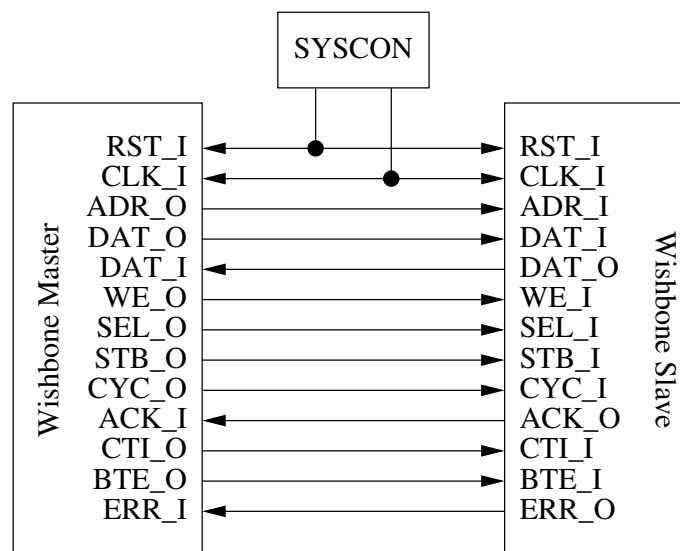


Figure B.1: Direct connection of Wishbone master and slave.

The interface specification can be used for connecting two cores directly just as well as it can be used for connecting several cores through some kind of bus. Fig. B.1 shows a direct connection of a master and a slave. This is the typical connection referred to in the timing diagrams later.

Table B.1: Wishbone signals (named from the master side).

Name	Direction	Width	Description
adr	M->S	32	Address bus
dat_o	M->S	32	Data bus out
dat_i	M<-S	32	Data bus in
we	M->S	1	Write enable
sel	M->S	4	Byte selects
stb	M->S	1	Strobe signal
cyc	M->S	1	Valid bus cycle
ack	M<-S	1	Bus cycle acknowledgment
cti	M->S	3	Cycle type identifier
bte	M->S	2	Burst type extension
err	M<-S	1	Bus cycle error

B.2 Interface signals

The Wishbone signals all use active-high logic (rule 2.30). Table B.1 shows a summary of the signals in the Wishbone interface. The following subsections will describe each signal in more detail. All signal names correspond to the master side.

B.2.1 adr

The `adr` signal is used to pass a binary address from the master to the slave. Note that the address granularity is on byte level.

B.2.2 dat_o and dat_i

The `dat_o` signal is used to pass a binary data from the master to the slave. The `dat_i` signal is used to pass a binary data from the (selected) slave to the master. The minimum granularity of the data is byte level as selected with the `sel` signal.

B.2.3 we

This signal indicated whether the current bus cycles is a write or a read. The signal is asserted for write cycles and negated for read cycles.

B.2.4 sel

The `sel` signal indicates where valid data is present/expected on the `dat_*` signals during the current bus cycle. With byte (8-bit) granularity on a 32-bit bus this signal will have four wires. Each wire will indicate that the corresponding byte is/should be valid.

B.2.5 stb

This signal indicates a valid data transfer cycle. The slave will assert either `ack` or `err` in response to every assertion of this signal.

B.2.6 *cyc*

This signal indicates that a valid bus cycle is in progress. This signal should be asserted for the duration of all (consecutive) bus cycles.

B.2.7 *ack*

This acknowledgment input indicates the normal termination of a bus cycle. Abnormal termination is indicated through the *err* signal.

B.2.8 *cti*

The *cti* signal indicates the current bus cycle type. This signal is described in more detail in the section on incrementing burst cycles, see section B.4.

B.2.9 *bte*

The *bte* signal indicates the current bus cycle burst type. This signal is described in more detail in the section on incrementing burst cycles, see section B.4.

B.2.10 *err*

This error signal indicates the abnormal termination of a bus cycle. The behavior of the master and slave in response to the *err* signal is not defined in the standard.

B.3 Wishbone classical cycles

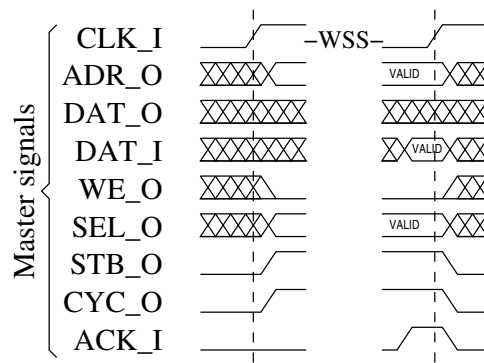


Figure B.2: Wishbone classical single read cycle.

A Wishbone classical bus cycle is signaled by asserting *cyc* and *stb* to the slave. These signals are acknowledged by the *ack* signal from the slave. Whether to read or write can then be selected by setting the other interface signals to appropriate levels. Fig. B.2 shows a timing diagram for a Wishbone classical single read bus cycle. The WSS (wait state slave) signifies that the slave can insert any number of wait states by keeping the *ack* signal low. Fig. B.3 shows a Wishbone classical single write.

The Wishbone classical bus cycles require the entire bus cycle to finish before starting a new cycle. This disallows bursty accesses from occupying a minimum of time since no overlapping is allowed. This has been addressed in revision B.3 of the Wishbone standard, see section B.4.

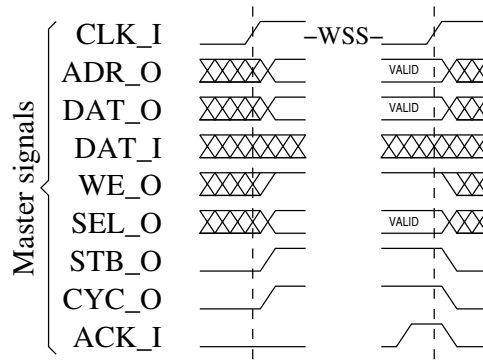


Figure B.3: Wishbone classical single write cycle.

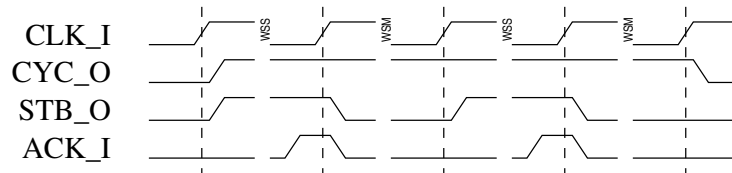


Figure B.4: Wishbone classical block cycles.

Burst-style accesses can be done through the classical bus cycles. This will (almost always) require wait-states to be inserted at various points during the access. Fig. B.4 shows a simplified timing diagram for a block (burst) access. The `cyc` signal indicates that a valid bus cycle is in progress. During the bus cycle the master can respond to the acknowledgment signal by either negating or asserting the `stb` signal in order to insert wait states or start new transactions respectively.

Just as with a single read/write, the slave can keep the acknowledgment signal negated to insert wait states if needed.

B.4 Wishbone incrementing burst cycles

So called Wishbone registered feedback bus cycles were introduced with revision B.3 of the standard. Of these bus cycles, the Wishbone incrementing burst cycles are most useful. This bus cycle can be used for doing burst read accesses with high throughput because the start of a new transaction is allowed to happen before the current transaction has finished.

The `cti` and `bte` signals indicate the type of burst cycle in progress. Table B.2 shows the meaning of these signals¹.

In order to keep compatibility between revisions of the standard, slaves can ignore these signals and revert to classical bus cycles. This should give the same behavior as if the extensions were not used at all.

Fig. B.5 shows a linear address burst read of four data. As can be seen the address from the master is held constant until an acknowledgment has been received. This ensures compatibility with the Wishbone classical cycles. After the first data has been read the acknowledgment is kept asserted and the data is delivered at a rate of one per cycle. This continues until the master

¹The 4-beat, 8-beat, and 16-beat wrap bursts mean that a set of 4, 8, or 16 addresses are repeated and is not very useful in this lab.

Table B.2: *cti* and *bte* signal values

Signal group	Value	Description
<i>cti</i>	000	Classic cycle
	001	Constant address burst cycle
	010	Incrementing burst cycle
	011-110	<i>Reserved</i>
	111	End of burst
<i>bte</i>	00	Linear burst
	01	4-beat wrap burst
	10	8-beat wrap burst
	11	16-beat wrap burst

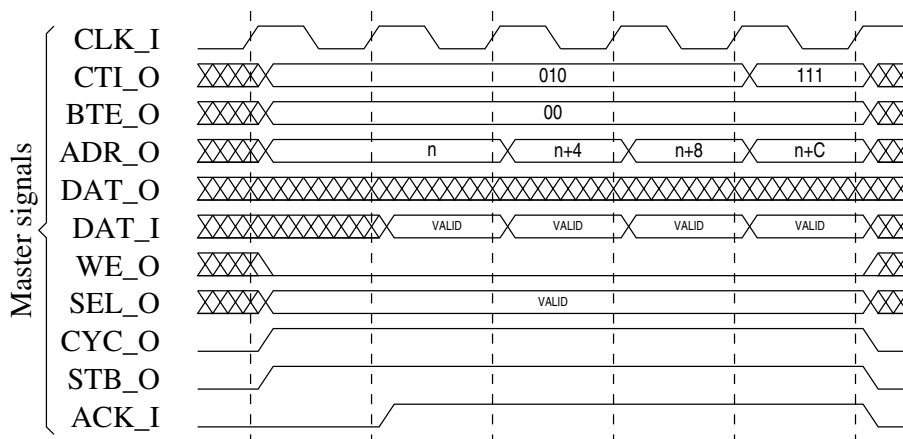


Figure B.5: Wishbone linear increment burst.

signals the end of the burst by setting the *cti* signal to 111.

The slave will not get a new address until the last transfer is completed as is obvious from the figure. This means that the slave will have to generate the following addresses internally according to the *cti* and *bte* signals. In the case of a linear increment this implies that the slave must contain a counter that increments the address.

B.5 System Verilog Interface

This is the System Verilog interface declaration used in the lab course.

Listing B.1: System Verilog Interface for the Wishbone bus

```
interface wishbone(input logic clk, rst);
    typedef logic [31:0] adr_t;
    typedef logic [31:0] dat_t;

    adr_t      adr;          // address bus
    dat_t      dat_o;       // write data bus
```

```
dat_t      dat_i;      // read data bus
logic      stb;        // strobe
logic      cyc;        // cycle valid
logic      we;         // indicates write transfer
logic [3:0] sel;       // byte select
logic      ack;        // normal termination
logic      err;        // termination w/ error
logic      rty;        // termination w/ retry
logic      cab;        //
logic [2:0] cti;       // cycle type identifier
logic [1:0] bte;       // burst type extension

modport master(
  output      adr , dat_o , stb , cyc , we , sel , cab , cti , bte ,
  input       clk , rst , dat_i , ack , err , rty );

modport slave(
  input       clk , rst , adr , dat_o , stb , cyc , we , sel , cab , cti , bte ,
  output      dat_i , ack , err , rty );

modport monitor(
  input       clk , rst , adr , dat_o , stb , cyc , we , sel , cab , cti , bte ,
              dat_i , ack , err , rty );

endinterface : wishbone
```

Appendix C

Tips & Trix

In this appendix we have collected a number of tips and trix that you might find useful.

- If you encounter some weird problems with the hardware you can try to turn the power off to the FPGA system before configuring it. We have had some problems with the FPGA board which can be solved in this manner.
- You do not have to restart Modelsim or recompile all files to get new changes included in the simulation. Just recompile the file you changed and type `restart -f` in Modelsim. Now your new changes is included.
- If you use emacs you can compile your Verilog file from emacs.
M-x compile
`vlog -work {work directory} +incdir+{include directory} {file}`
- To log more signals than you want to display in the wave window you can use the log command.
To log all signals under e.g. `dafk_tb/dafk_top/cpu` you can use
`log -r dafk_tb/dafk_top/cpu/*`.
- To make backspace work in μ Clinux when using Teraterm. Under Setup->Keyboard... make sure that the check box for how the backspace key is sent is set.
- If you haven't setup the backspace key to work correctly in μ Clinux, you can still use some other standard unix key combinations: control-w will erase the previous word. control-u will erase the entire line.
- If you find it hard to see if a signal is a 1 or a 0 in modelsim when many 1 bit signals are next to each other in the wave window you might want to change the colors for 1 and 0 respectively. This can be done by using Tools -> Edit preferences.. menu selection on the modelsim main window. In the window that opens up, select WaveWindows.
- If you use verilog-mode in emacs, you can press control-c control-d to go to another module. (The default module name is taken from the location of the cursor in the buffer.)
- If you use the command `show_dafk_window` in modelsim you will get a window that allows you to add a number of different signals to your modelsim window. You can look at the file `modelsim.tcl` in the `simulator` directory to see how this is implemented. You may also press the button named *Show dafk window* in the wave window to open this window (the button is only visible if the wave window is removed from the main modelsim window).
- If you use some of the virtual signals declared by us, make sure to use the restart script `restart-f.do` instead of restarting Modelsim the normal way.

- If you cannot find certain commands, make sure that the TSEA44 module is loaded:

```
module load TSEA44
```