Interrupt Controller and Programmer IO

Group 16 – Assignment 4(B)

# Introduction

In this project we extend our processor by adding to it two capabilities: programmer IO and an interrupt controller. For interrupt handling facilities, we use the round robin scheduling which has the advantage that all interrupt lines get equal weightage. We make a separate interrupt controller which detects when an interrupt line has gone high and uses two lines, INTR and INTA, to communicate this to the processor and give it the address of the interrupt service routine (ISR). While interrupts are being served, the interrupts that are in queue are stored in an interrupt request register (IRR). The addresses of the interrupt service routines are placed in an interrupt vector table (IVT) which is read by the interrupt controller and passed to the processor. We also implement another ‘interrupt acknowledge’ instruction that the ISRs use to indicate that the ISR has finished executing (as we explain later, this is essential in saving scarce stack space when spurious interrupt signals come because of poor quality of the switches on the FPGA

To add programmed IO, we add one more instruction to our instruction set which is an In/Out instruction. Since programmed IO is not our main objective (it is the inclusion of interrupt capabilities), we dedicate half of the eight ports to input and half to output. Hence we can make do with just one instruction, using one of three bit port address to determine direction of data transfer. We place this instruction as 11 1 11 xxx where xxx is the port address. The first 11 is for push/pop, jump or IO instructions and 1 specifies whether it is a Jump instruction or a push/pop or IO instruction. The second 11 specifies that it is an IO instruction.

# Architecture

## Overview

Both programmer IO and interrupt control are done through a module that is separate from the processor. This module is connected to the main data bus of the processor and a minimal number of control lines that are clock-synchronized flows between the two modules. Note that clock-synchronized communication means that communication is very efficient and fast (but the controller may become a bottleneck, something we believe is highly unlikely) and hence IO is quite fast in our processor. This is appropriate in a small processor design as it is likely to be used in embedded applications and is hence likely to run IO intensive applications.

## Block-Diagram

…

Address

Enable

Data Bus

Data Bus

Data Bus

…

Port 8 (out)

Port 7 (out)

Port 2 (in)

Port 1 (in)

Address

INTR

INTA

Interrupt

Wait

Controller (FSM)

16 Byte IVT

Interrupt Sensor and Scheduler

Enable   
 low/high

## Explanation

### Interrupt IO

In our system the programmed IO and interrupt IO are largely independent of each other. In interrupt IO, the sensor has an 8 bit IRR that becomes 1 when the interrupt line goes from low to high and is set to zero after the interrupt has been processed. It has a counter to enable the round robin scheme. We have a design where the current bit in the IRR (as determined by the counter) is simply output to the interrupt line going into the controller along with the contents of the counter. In case the controller is willing to process the interrupt, it sends a wait signal back to the sensor which causes the counter to stop. Note that the port address lines now contain the address of the interrupt line whose interrupt needs to be processed. Further the interrupt bit in the sensor is set to zero so that it is not processed repeatedly. Now the controller moves to the next state where it sends the INTR signal to the processor. The processor processes this when it reaches the fetch0 state. Once there it goes into a separate cycle where it stores the PC and the accumulator and sends back an INTA signal. Now the processor puts the enable line to 1 and low/high to 1 causing the IVT to transfer the lower byte of the address to the data bus which is read by the processor. In the next cycle, the controller complements the low/high line causing the higher byte to be transmitted to the processor.

Now the controller waits for the INTA to go high again, indicating that the processor is done executing the interrupt (we do this by implementing a special instruction that is executed at the end of every interrupt handler). This is essential as we do not want multiple interrupts to be processed at the same time because the switches on the FPGA are far from perfect and oscillate a lot. So if multiple pushes onto the stack are performed, the stack may overflow into the other parts of the memory causing bugs. Further it makes sense to save stack space on our very limited resources this way.

Once INTA goes high, the controller assumes that the ISR has done executing and it can listen for interrupts again. So it sets the wait signal to zero and starts responding to the interrupt signal being high. Note that the interrupt line is again set to 0 to ensure that the oscillations did not spuriously make it one again.

### Programmed IO

The programmed IO is rather simple and a lot of work is done simply in the mux/demux part of the circuit. The address lines give the address of the port which is to be read from/written to and this also decides whether to perform input or to perform output. Now, if enable line is up and the port is an input port the data appears on the data bus (which otherwise is in high-impedance mode). If the enable line is up and the port is an output port the data is written to the port register. Note that the output ports use a register to which data is written by the FPGA and can be read by the outside world. The input ports directly read from the input lines as it is much more straightforward.

# Design Issues

The major issue we faces is that the switches on our FPGA board oscillate a lot (sometimes faster than the clock itself). Hence we added an interrupt acknowledge instruction which also saves stack space and is much more efficient.

Further we separated our interrupt controller into modules so that the controller can be independent of the data path. This enables us to have a relatively simple design such that the address goes directly from the sensor to the IVT.

Another challenge we faced was to design the sensor, where we are greatly restricted in what signals we can detect as it is not possible to have conditional actions that are triggered simultaneously by both edge and level events. Hence the IRR is entirely edge triggered and all other functionality is implemented by other circuits. The addition of the wait signal greatly simplified are design and removed the requirement for a separate memory.

# Controller fsm