Recitation 1: Orientation

Yaniv Eliezer

"Man is still the most extraordinary computer of all". J.F.K.

Contents

• Syllabus
  ▪ Course staff
  ▪ Goals and requirements
  ▪ Schedule
  ▪ Guidelines
    • Reports
    • Quizzes
    • Lab activity
• Recitation 1
  ▪ Orientation
Course staff

- Academic supervisor
  - Prof. Guy Even / Prof. Benny Applebaum / Prof. Dana Ron
    - faculty’s website

- Lab engineer
  - Marko Markov
    - marko@eng.tau.ac.il
    - http://www.eng.tau.ac.il/~marko/

- Recitations and grading
  - Yaniv Eliezer and Daniel Vana
    - Exercise submission and QnA – csl.bodek@gmail.com
    - Announcements, QnA – Moodle
    - Meeting hour – scheduled on demand

Goals and requirements

- Gradual logical design, realization and testing on an FPGA platform

- Final product
  - State of the art simplified RISC DLX CPU
Lab Equipment

- **PC**
- **XuLA2 board**
  - FPGA (Programmable hardware)
  - SDRAM (Memory)

**Xilinx software package**
**RESA software suite**

Schedule

<table>
<thead>
<tr>
<th>Recitation</th>
<th>Subject</th>
<th>Handout</th>
<th>WW</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orientation</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>The RESA bus</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Bus slave and I/O control logic</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Monitor slave</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Monitor slave continued</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Read/Write machine</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Load/Store machine</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Load/Store machine testing</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>DLX design</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>DLX simulation and implementation</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>If needed</td>
<td>DLX testing and timing</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>On demand</td>
<td>DLX assembly program</td>
<td>7</td>
<td>12</td>
</tr>
</tbody>
</table>

Midterm quiz

Final quiz
Guidelines

• Grading
  ▪ Personal evaluation according to
    • Quality of submitted reports
    • 2 quizzes (midterm and final, equal weight)
    • The way you work in the lab

![Pie chart showing grading components]

Reports guidelines

• Team work
  ▪ 2 brains

• Reports (30%+35%)
  ▪ Handout 1 is done in the lab with the lab’s engineer
  ▪ Checking standards
    • Will be detailed in the next recitation
Guidelines

- Quizzes (25%)
  - 4 subjects, equal weight
    - Busses
    - Debugging
    - Load/Store machine
    - DLX machine
      - Will be scheduled after the semester’s end
- Mandatory reading material
  - Matching chapter of the lab notes
- Lab Activity (10%)
  - Unjustified delays/misses
  - Under-preparedness
  - Procrastination
- Bonuses (G+=x)
  - As the lab’s engineer/instructor will see fit

Reading

- For the next recitation
  - Chapter 2: The RESA’s buses
Recitation 2: The RESA Buses

Contents

• Bus
  ▪ Busses in general
  ▪ RESA bus architecture
  ▪ RESA bus protocol
• Handout 2 highlights
The RESA architecture

- **XuLA2 board**
  - Embedded Spartan-6 FPGA
    - Field Programmable Gate Array
  - Embedded SDRAM
    - Synchronous Dynamic Random Access Memory
  - Embedded Bus Controller
  - USB interface
    - Universal Serial Bus

A Bus in general

- “In computer architecture, a **bus** is a communication system that transfers data between components inside a computer, or between computers.”
- The connected components follow a bus **communication protocol**

* from the Latin word “ omnibus”, meaning “for all”
RESA Bus protocol

- The communication between components is taken in turns which are called transactions.
- Each transaction involves the transmission of one data word.
- Master components
  - initiate the transaction
- Slave components
  - respond to it
- Write transaction
  - The masters writes data to a slave
- Read transaction
  - The masters fetches data from a slave

RESA Bus protocol

- RESA’s bus follows a synchronous protocol
  - A global clock is connected to all components
  - Every component is synchronized with the rising edge of the clock
  - Every transition of a signal on the bus is synchronized with the rising edge of the clock
**Bus protocol**

- **Serial communication**
  - Words are transmitted serially

- **Parallel communication**
  - Words are transmitted in parallel

---

**The RESA busses**

- **External bus**
  - Universal Serial Bus interface
  - PC and XuLA2 board communication

- **Internal bus**
  - Parallel bus
    - FPGA
    - SDRAM
    - Bus controller
The RESA’s internal bus

- FPGA
  - The place for our design and implementation
  - DLX CPU - Master
  - Monitor slave - Slave
- Memory
  - Slave
- PC interface
  - Master

RESA’s internal bus signals

- Read transaction

Master acquires the bus

\[ \text{IN\_INIT} \]

/AS

/WR

/ACK

\[ A[31:0] \]

\[ D[31:0] \]

M: “The bus is mine and mine alone!”

M: “Attention all slaves! take a note of an address”

M: “I wish to read”

\[ S*: \text{“acknowledging request! Requested data is ready!”} \]

\[ A[31:0] \]

\[ D[31:0] \]

Slave signals the info. will be ready in the upcoming CC

Slave transmits data. The master samples it.
RESA’s internal bus signals

- **IN_INIT**
  - Indicates the status of our primary master
  - Set to **HIGH** (default)
    - Our designed **Master is idle** and the bus belongs exclusively to the another master (PC interface)
  - Set to **LOW**
    - Our designed **Master is active and the bus is exclusively his to govern**

RESA’s internal bus signals

- **Address strobe (AS_N)**
  - The alerting of a data transaction is triggered by the master device on this signal
  - Slaves are alerted by this signal to “know” that the next clock cycle a transaction will commence

- **Address (A[31:0])**
  - The address multi-wire signal is used to transmit address of the slave **AND** requested data within the device
  - **MSB** bits hold the slave device address
  - **LSB** bits hold the data address within the device
RESA’s internal bus signals

• Data (D[31:0])
  - Holds the transaction’s data which is transmitted by the master or slave
    • Write transaction - The master uses the lines to transmit the data
    • Read transaction - The slave uses the lines to transmit the data

• Write (WR_N)
  - The master indicates the type of transaction
    • Write transaction – when the signal is LOW
    • Read transaction – when the signal is HIGH

• Acknowledged (ACK_N)
  - The slave acknowledges the transmission of data by lowering the signal
    • Write transaction - The master’s transmitted data is written to requested address.
      Written data will be valid at the beginning of the next CC.
    • Read transaction - The requested data (in the requested address) is currently transmitted to the master.

RESA’s internal bus signals

• Write transaction

---

M: "The bus is mine and mine alone!"
M: "I wish to write"
M: "This is the address I wish to write in"
M: "Data to write is 01011101..."
S*: "S* acknowledging request! Requested data is being handled!"
M: "This is the address I wish to write in"
M: "Data to write is 01011101..."
S*: "S* acknowledging request! Requested data is being handled!"
M: "The bus is mine and mine alone!"
M: "I wish to write"
M: "This is the address I wish to write in"
You shall implement a **BUS_INF** module designed to act as a functioning bus module between a **CPU** (master) and a **slave**

The **BUS_INF** module should be designed by you according to specifications

You get 2 modules for **free**
- You may observe the inside implementation of these modules

You will sample **ALL** control signals

Follow with care
- Handout instructions
- Preliminary questions

---

**General schematic**

- **CPU** - Given as is
- **slaves**
- **bus_inf**
- **Slave** - Given as is
- **Your implementation**
Handout 2 highlights

• CPU and bus interface communication is implemented using 3 registers.
• Control signals given to you
  - CLK
  - WR_REQ, RD_REQ
  - BUSY
  - ACK_N
• Required control signals
  - IN_INIT
  - DONE
  - AS_N
  - WR_N
  - CEO, CE1, CE2
  - DE1, DE2
• Missing blocks
  - Buffers
  - Additional logic, Flip-Flops

Handout 2 highlights

CPU 'read' instruction

CPU 'write' instruction
Handout 2 highlights

CPU ‘read after write’ instruction

Report guidelines

• Report submission standards
  • Handout 2 is introductory
    • Pay attention to the checking remarks of handout 2 report
    • All other handouts has the same basic standards
Report guidelines

• Report submission
  ▪ Report format
    • English only
    • Single file in a single Mail
    • PDF format
    • Group number on top (no names, no IDs)
  ▪ Report Quality
    • Each answer must be composed of a short and decisive explanation
  ▪ Signals and components behavior
    • Naming conventions – signals and ports should be named properly
    • Signal charts must be detailed in short

Report guidelines

• Reports delivery
  ▪ All reports must be submitted on time by mail
    • Pre-lab reports
      ▪ Submission before the relevant lab
    • Post-lab reports
      ▪ Submission at least 48 hours before the subsequent lab
  ▪ Unpermitted or uncoordinated delays will result in points reduction
  ▪ Mail subject and file naming formats
    • “ACSL Handout <handout number> <Prelab/Postlab> Group <group> B15.pdf” (e.g.: “ACSL Handout 2 Prelab C4 B15”)
  ▪ Once a report is submitted you’ll receive an automatic response
    • Make sure you receive it
State machine changes states with time

Arbitrary memory address we read from

Data we read from memory (as a check)

Reset signal. Machine is going back to initial state.

Master asks slaves for attention

Machine is out of initial state

State machine changes states with time

The time sequence we are not interested to sample.

Proper signal names (Justin Bieber CLK_IN)

Short explanations

Clearance

Proper signal names (Justin Bieber CLK_IN)

Short explanations
Slave Labels:
STATUS = 0x0000c406
addr = 0x00000001
do = 0x00003421

Memory Labels:
datain = 0x00003421

Memory Dump:
0x00000000 : 0x0000003421 0x00000000 0x00000000
0x00000004 : 0x00000000 0x00000000 0x00000000
0x00000008 : 0x00000000 0x00000000 0x00000000
0x0000000c : 0x00000000 0x00000000 0x00000000
0x00000010 : 0x00000000 0x00000000 0x00000000
0x00000014 : 0x00000000 0x00000000 0x00000000
0x00000018 : 0x00000000 0x00000000 0x00000000
0x0000001c : 0x00000000 0x00000000 0x00000000

Short explanations

Reading

- For the next recitation
  - Chapter 3: A simple slave device
Handout 2 highlights

• General schematic

HANDOUT 2: THE BUF_INF SOURCE
Design instructions

- Use ready sources from project HOME
- Create your own 32 bit source of BUFE32 like existing BUFE16
- Insert your logic design in the BUF_INF by one of the ways:
  - from scratch
  - mark your schematic without ports, copy (ctrl+c) – paste special(not ctrl+v).
  - add copy of source and place it within the page of BUF_INF.
    don’t use BUF_INF as a name for your source!!
Advanced Computer Structure Lab

Recitation 3: A simple slave device

Yaniv Eliezer

Contents

• RESA architecture
• RESA monitoring program
• I/O control logic
• Monitor slave
• Handout 3 highlights
The RESA architecture

- **XuLA2 board**
  - FPGA
  - SDRAM
  - Microcontroller (uC)
    - Bus controller
    - USB interface to PC

XuLA2-LX25 architecture

- Voltage Regulators
- SPI Flash
- microSD Card
- USB Port
- Oscillator
- Spartan-6 FPGA
- 32MB Memory
- XuLA2 Micro-Controller (uC)
Machine vs. machine

PC
- CPU
- RAM
- Bus controller (PCI)
- HDD controller
- Extension cards
- USB port
- HDD

HDD controller

Motherboard
- RAM
- CPU
- Bus controller
- Extension cards
- USB port
- HDD

HDD

1. OS code
2. Software code ("Minesweeper")
3. Data ("Torrents")

USB port

XuLA2 card
- FPGA
- SDRAM
- uC
- Ext

1. DLX CPU logic design
2. Monitor slave
3. BUS logic

1. DLX CPU code
2. Data

Data path

CPU
- RAM
- Bus controller
- Extension cards
- USB port
- HDD

HDD controller

PC Motherboard

-memory (RAM) loading
-memory (RAM) reading
-FPGA programming

Xilinx software (prog.)

RESA software (monitoring)

OS

1. Master
2. Monitor slave
3. I/O logic

Data request

Data reply

FPGA

Monitor
Slave

CPU

Master

I/O logic

RAM
The RESA monitoring programs

- The RESA monitoring programs are a bunch of programs designated to monitor and setup the RESA’s hardware

- Relevant features
  - Programming the FPGA
    - Xilinx software creates a ‘.bit’ file
    - RESA programs the FPGA gating using the ‘.bit’ file
    - Transfers gating configuration to the uC which programs the FPGA

The RESA monitoring program

- Slave monitoring
  - Accessing RESA memory space - Memory slave
    - Reading results from memory (i.e. after CPU completes execution)
    - Writing to memory space (i.e. writing DLX assembly programs into memory for the CPU to handle)
  - Monitoring program menus - Monitor slave
    - Initiates read and write transactions to Slave addresses
    - Controls built-in monitoring (will be elaborated...)
The RESA monitoring program

• Relevant features
  ▪ Running and debugging
    • Upload DLX assembly programs (Memory)
    • Set single step or continuous mode (CPU)

The RESA monitoring program

• DLX CPU programming steps
  ▪ Textual writing of DLX programs
    • DLX Assembly instructions
  ▪ Compilation
    • Text to binary instructions
  ▪ DLX simulator testing of your program
  ▪ Visual graphs generation
  ▪ HW interface communicates with the FPGA and Memory using the uC
**The I/O control logic**

- The I/O control logic serves as the bus interface for the master and slave devices (similar in functionality to what you have implemented in HO2).
- The RESA parallel bus is implemented within the I/O control logic since both RESA application and monitoring slave access the RESA bus via the I/O control logic.

---

**Diagram:**

- From outputs of Bus Master:
  - AS_N
  - MAC[31:0]
  - MDC[31:0]
  - WR_OUT_N
  - IN_INIT

- From outputs of Bus Slave:
  - SACK_N
  - SDO[31:0]

- To Inputs of Bus Master:
  - Stop_en
  - Reset
  - ACK_N

- To shared inputs of Bus master and slave:
  - clk
  - DQ[31:0]

- To inputs of Bus Slave:
  - Card_sel
  - A[31:0]
  - WR_IN_N
**Input signals of the master device**

- **STEP_EN** — One CLK cycle pulse which causes the master to perform a single step
- **RESET** — master reset signal
- **CLK**
- **ACK_N** — Sent by slave to indicate transaction acknowledgment
- **DO[31:0]** — Data out to master and slave devices (shared port)

**Output signals of the master device**

- **AS_N** — Address strobe signal (active low)
- **MAO[31:0]** — master address out port
- **MDO[31:0]** — master data out port
- **WR_OUT_N** — R/W signal of the bus master (High → Read, Low → Write)
- **IN_INIT** — indicated the status of the master. On low the master is **idle** and the bus is inaccessible. When High the master is **active** and the bus can be accessed by the monitoring program.
**Input signals of the slave device**

- **AI[9:0]** — Address in of the slave device on the FPGA (How many addresses?)
- **WR_IN_N** — slave writing indication signal (active low)
- **CLK**
- **DO[31:0]** — Data out for master and slave devices (shared port)
- **CARD_SEL** — Indicates that the current slave is on the FPGA. The signal is computed by the I/O control logic from the 22 MSBs of the address.

![Diagram of I/O Control Logic](image)

**Output signals of the slave device**

- **SDO[31:0]** — Slave data out port
- **SACK_N** — The acknowledge signal marks the Data Out validity (Active low)

![Diagram of I/O Control Logic](image)
Handout 3 highlights

• The monitoring slave
  ▪ A trivial master device will be granted to you for free
  ▪ You will design a slave device that can monitor the requested values from the master
  ▪ You will be using the monitor program to initiate read transactions from the slave device

As it should be

Handout 3 highlights

• The monitor slave device is connected to the master device by a set of private wires which are not part of the bus – a bypass

• An extension of this approach will be used as a built-in monitoring mechanism for the DLX CPU

As should be implemented
The master

- **The master device**
  - 32 bit binary counter connected to 32x32 bit RAM
  - The 5 LSB output bits of the counter (4:0) are used as RAM address
  - The full 32 bits of the counter are used as RAM data input
  - The master doesn’t initiate any bus transactions and is therefore a degenerated bus master
The slave

• The slave device will spy and relay to the bus one of the following
  ▪ The values stored in the 32x32 bit RAM
    • reg_out[31:0]
  ▪ The state of the “broja” machine
    • state(3:0)
  ▪ The value output by the counter
    • step_num(4:0)
  ▪ The writing address
    • reg_write(4:0)
  ▪ The lab group code
    • ID(7:0)
• The address space of the slave device is defined by 4 addresses

The slave

• When the PC monitor program (Master) wishes to read the counter’s values it initiates a read transaction with the address of the counter’s output
• The slave device should receive this request and route the counter’s output to the SDO port
• Finally, the slave device will ACK that the requested data has been sent and the read transaction is complete (SACK_N)
Slave address partitioning

- **AI(9:0)**
  - **BA(2:0)**
    - Block address – chooses a block (1/8)
  - **PA(1:0)**
    - Page address – chooses a page (1/4)
  - **WA(4:0)**
    - Word (32bit) address – chooses a word (1/32)

- A single slave device represents an entire block

---

Block diagram

Master-slave “Private wires”
Block diagram

Degenerated master
- CLK
- STEP_EN
- RESET
- REG_OUT(31:0)
- STEP_NUM(4:0)
- STEP_NUM(4:0)
- State(4:0)
- REG_ADR(4:0)
- REG_WRITE(4:0)

Monitor slave
- 0x00
- 0x20
- 0x40
- 0x80
- SDO
- Mux

Slave control

- Optional implementation of the slave control

I/O logic
- CARD_SEL
- WR_IN_N
- SACK_N
- REG_ADR(4:0)
- REG_OUT(31:0)
- STEP_NUM(4:0)
- State(4:0)
- REG_WRITE(4:0)

Give me the secret data!
Handout 3 highlights

• Design and Implement
  - Wiring, Logic components ...
  - Group ID component
• Simulate and verify your design
  - Data that should be additionally monitored by you and available in the report
    - Reg_out(31:0)
    - State(3:0)
    - Step_num(4:0)
    - Reg_write(4:0)
    - ID(7:0)

• After implementing, produce a `.bit` file
• Create configuration labels
• Run the design on the RESA by using the monitoring program
• Include your report with
  - Design schematics
  - Simulations
  - Label reports
  - Data snapshots of three sequential steps which demonstrate
    - Advancement of step_num(4:0)
    - Advancement of reg_write(4:0)
    - Reading of ID(7:0)
Reading

- For the next recitation
  - Chapter 4: Built-in self monitoring

resa’s Internal Structure

Two Masters:
- PC with RESA debugging software
- Students’ Design of DLX CPU (on FPGA)

Two Slaves:
- External Ram (on Board)
- Students’ Monitor (on FPGA)
Design Flow

• 3.2 Design
• Understand in details the schematic and functionality of the given master device.
• Make address partitioning, single and 32 word blocks, according to the regulations described in Lab Notes and RESA monitor user guide. Create table with address of the block, pages and corresponding inputs of the slave device.
• Using the project home_v25 as a design environment, design a simple component ID_NUM with a single 8 bit constant output ID[7:0] of your group number and a slave device, capable of allowing reading information from four 32 bit inputs according to your address partitioning.
• Create the slave.t.vhd (VHDL test bench file) file for your slave device and check it using ISE Simulator.
• Use function add copy of source to transfer your designs from home_v25 project to project sources_v25. Place your components on the sheet and connect them to the master device and I/O Logic bus.
• Execute “Generate programming file” for your design in order to receive ‘.bit’ file.
• Run and debug your design using the Hardware monitor of the RESA program.

Slave address partitioning

• AI(9:0)
  ▪ BA(2:0)
    ▪ Block address – chooses a block (1/8)
  ▪ PA(1:0)
    ▪ Page address – chooses a page (1/4)
  ▪ WA(4:0)
    ▪ Word (32 bit) address – chooses a word (1/32)

• A single slave device represents an entire block
**Table of Slave’s inputs, addresses and labels**

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Addresses</th>
<th>Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0x180</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0x1A0</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0x1C0</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0x1E0</td>
<td></td>
</tr>
</tbody>
</table>

**Inputs and addresses** represent hardware design of the Slave and will be the same until the end of the Lab.

**Labels** represents connections between Master and Slave (i.e. private wires) and vary with the Labs.

The Labels have to be up to 8 symbols of ONLY small letters and numbers.

---

**Block diagram**

- **Degenerated master**
  - CLK
  - STEP_EN
  - RESET
  - SDO
  - ACK_N
  - AI(9:0)
  - CARD_SEL
  - WR_IN_N

- **D.M.**
  - REG_OUT(31:0)
  - STEP_NUM(4:0)
  - State(4:0)
  - REG_WRITE(4:0)

- **Monitor slave**
  - 0x180
  - 0x18a
  - 0x18c
  - SDO

- **Mux**
  - 0x00
  - 0x20
  - 0x40
  - 0x80

- **Slave control**
  - AI
  - CARD_SEL
  - WR_IN_N
  - SACK_N

- **I/O logic**
Sub Bus connection and concatenation

- The net MUST be with the same size with the corresponding port.
- The sub bus can be mapped within the corresponding bus by changing the label’s indexes while saving the size.
- When few sub buses have to be mapped, overlapping of indexes is forbidden.
- For every label it is recommended usage as LSB index one of nibble’s LSB (i.e. 0, 4, 8, 12, 16, 20…). As a result the numbers, displayed by the software will be in a very comfortable reading format.

How to use the RESA program

Open the RESA

The base functions are:

- **Compiler**: Used to write and compile programs in Assembly language
- **DLX Simulator**: Used to test Assembly programs
- **Hardware monitor**: To test the hardware designs using the FPGA board

Start testing with choice of the corresponding Lab
How to use the RESA program

Fill the Labels from the table

- **Slave Labels**: Read transaction from the defined address is executed and the result is presented.

- **CPU RAM start address**: Defines the page where the CPU Registers are placed. 32 Read transactions are executed by RESA, starting from defined address with increment by one.

Save Labels to file and Load Labels from file can be used to manage the Labels.

The Labels have to be up to 8 symbols of ONLY small letters and numbers.

---

How to use the RESA program

- In order to configure the FPGA browse and point to the proper .bit file.

- After configuration the FPGA screen with results of the performed read transactions is opened.

- Use the buttons: **Step**, **Reset** and **Refresh data** to control the Masters functioning and to receive the corresponding monitored data.

- The generated set of labels and the monitored data can be stored as text file by using the functions **Save labels file** and **Save Data Snapshots**.
Contents

• Built in monitoring
  ▪ Monitor slave
  ▪ CPU execution loop
  ▪ The logic analyzer
• Handout 4 highlights
High level architecture

- FPGA
  - DLX CPU – our future implementation
  - Monitor slave – our implementation
  - I/O control logic – given
- PC Monitoring program
- Memory

Monitoring device

- Primary goal
  - Debugging step by step
  - Monitoring
    - CPU registers
    - control signals
- Monitoring
  - Performed by HW implementation which will connect directly to the DLX CPU
- Monitor slave
  - We will evolve it into something more complex
**Primary lab goal**
- Designing a CPU which executes DLX assembly instructions

**Instructions execution cycle**
- **Fetch phase**
  - Fetches an instruction from Memory for execution
  - Holds the BUS
- Instruction execution is done in an interval of clock cycles between two consecutive fetch states

**Initial state**
- Init

**Two modes of operations**
- **Single step**
  Each execution of an instruction waits until a confirmation signal - HIGH STEP_EN pulse arrives from the PC monitoring program
- **Continuous mode**
  STEP_EN remains always HIGH
  The execution of the next instruction is totally unconditioned without passing in the INIT state
DLX CPU Monitoring

- **Debug mode**
  - The DLX CPU is running in the single step mode
  - The slave device allows reading valuable “snapshot” data when the DLX is in “Init” state

- **Not good enough**
  - Can’t monitor the bus signals flow
  - Can’t monitor the internal signals of the implementation over instruction execution

The logic analyzer

- **Real logic analyzer**
  - Buy one?! Naaa! too expensive...
  - Bad hardware connectivity

- **We are going to save money and make our own version**
  - A new monitoring slave capable of
    - Inner monitoring of control signals during execution
    - Inner monitoring of CPU registers
    - Post execution reporting to PC monitoring software
The logic analyzer

- Logic analyzer
  - HW Requirements
    - Memory to store data during instruction execution
  - Functionality requirements
    - Sampling and Storing of monitored signals cycle by cycle
      - During instruction execution
    - Answering bus read transactions
      - The PC Monitoring software asks the sampled values
      - After instruction execution cycle is complete

The logic analyzer low level design

- Logic analyzer main components
  - 32x32bit RAM
    - Samples monitored data sequentially “word by word”
    - The memory is filled until the instruction’s execution ends
The logic analyzer low level design

• Logic analyzer main components
  • 5bit Counter
    • Generates storage addresses
    • Counter value = number of sampling cycles
    • After instruction execution completion the counter
      is reset by the LA’s logic

2x5bit multiplexer
  • Enables multiplexing between
    • Internal counter address which is used to store data “word by word”
    • External address which is used by the PC software to read from the 32x32 bit RAM
The logic analyzer low level design

• Logic analyzer main components
  • Status register
    ▪ Latches the value of the 5-bit counter so the monitoring slave can report the number of memory filled rows (=number of sampling cycles)

Handout 4 highlights

• Design components
  ▪ I/O logic
  ▪ Handout 3 master
  ▪ Monitoring slave
    ▪ Handout 3 slave
    ▪ Handout 3 ID register
    ▪ Logic analyzer
Handout 4 highlights

- Monitoring slave
  - Handout 3 slave
  - Handout 3 ID register
  - Logic analyzer

- Logic analyzer control signals

Information from the monitor slave can be retrieved only here.

"Mind the gap"
Note the CC before and after!
Handout 4 highlights

• Logic analyzer control signals
  ▪ Waveforms

Handout 4 highlights

• Recommended work schedule
  ▪ Next week
    • Design over the Xilinx platform
    • Start simulating
  ▪ The week after that
    • Finish simulations
    • Implement design over the RESA platform
    • Monitor the design using the RESA PC monitoring software

• Additional guidelines
  ▪ Submit a simulation showing
    • The Sampling process of the logic analyzer
    • The Reading process from the monitor
      ▪ LA’s RAM
      ▪ LA’s status ID
      ▪ External inputs
    • Control signals
Handout 4 highlights

• Additional guidelines
  ▪ Submit graphical waveform snapshots of the RESA monitoring program showing 2 “STEP_EN” cycles
    • Sampled signals from the logic analyzer’s RAM
      ▪ In_init
      ▪ State(3:0)
      ▪ Step_num(4:0)
      ▪ Reg_write(4:0)
    • Logic analyzer’s “Status”+“ID”
    • External inputs:
      ▪ Master’s RAM
      ▪ Step_num(4:0)
    • Control signals
  ▪ Design correctness
    • Convince yourself and us the design is ultimately correct
    • Attach proper and concise documentation

Reading

• For the next recitation
  ▪ Chapter 5: A Read machine and a Write machine
**Block diagram of the Monitor**

The Monitor consists the following blocks: **Slave** and ID NUM from Handout N3. The new designed **Logic Analyzer**.

**Remark:** LA's Status resides in the 8 LSB of the corresponding input. No specific demands for ID_NUM location.

**Table of Monitor labels (Slave)**

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Addresses</th>
<th>Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0x180</td>
<td>la ram</td>
</tr>
<tr>
<td>B</td>
<td>0x1A0</td>
<td>STATUS</td>
</tr>
<tr>
<td>C</td>
<td>0x1C0</td>
<td>Ext1 (master ram?)</td>
</tr>
<tr>
<td>D</td>
<td>0x1E0</td>
<td>Ext2 (step num?)</td>
</tr>
</tbody>
</table>

**Inputs and addresses** represent Hardware design of the Slave, so are the same used in the previous Lab N3. **Labels** represent connections between Master, LA and Slave (i.e. private wires). **STATUS** is reserved word and must be written using the capital letters.
**Table of Monitor labels (Graphic)**

- The graphic Labels represent the list of sampled signals.
- The structure of every label is Name and Address.
- The name is associated with the function of the specific signal.
- The Address is associated with the position of the Signal within the LA input bus. So, the address range of the Graphic Labels is 0x00-0x1F.
- When sub bus is connected to the LA input bus, every signal is defined separately, with his own Name and Address.
- For Graphic Labels a list of 32 labels maximum can be created.

---

**How to use the RESA program**

- Start of the RESA Program is as in the Lab N3
- Choose the Lab N4 Build in Self monitoring.
- Two more windows are active: Graphic Labels and LA RAM start address.
- The Graphic Labels are generated in the same way as Slave labels.
- LA RAM start address defines the page where the LA RAM is placed. STATUS Read transactions are executed by RESA, starting from defined address with increment by one.
How to use the RESA program

- From the window Monitoring results sampled signals can be viewed using the button Signal Waveforms.
- With the presented graphics you can make combine, spread, change direction and order, zoom in and out, numeric representation and print to pdf file.
- **Important**: Combine sets the BUS without changing the order. The list of signals have to ordered before, the best is in the time of Labels’ generation.
- After every refresh data, the Graphic window is presented in spread view.
- Graphics can be stored to file and viewed later with graphic editor.

Advanced Computer Structure Lab

Recitation 5: Read/Write State machines

Yaniv Eliezer
Contents

• Read/Write machines
• Read/Write state machines
  ▪ State machines in VHDL
    • VHSIC (Very High Speed Integrated Circuits)
      Hardware
      Description
      Language
• Handout 5 highlights

High level architecture

• FPGA
  ▪ DLX CPU
  ▪ Monitor slave
  ▪ I/O control logic
• PC Monitor program
• Memory
Read machine and Write machine

• Read/Write Bus master
  ▪ A precursor to the Load/Store machine of the DLX CPU
  ▪ Initiates read and write bus transactions through the I/O logic

• Read machine
  ▪ Reads contents from memory to a designated register

• Write machine
  ▪ Writes arbitrary values to memory

---

Read machine

• Read state machine
  ▪ **Wait**
    ▪ The machine waits a STEP_EN signal
    ▪ Until then it’s trapped on ‘wait’ state
  ▪ **Fetch**
    ▪ The machine initiates a read transaction to the bus
  ▪ **Wait4ACK**
    ▪ The machine waits for an ACK signal
    ▪ When ACK arrives data is written into the register
  ▪ **Loaded**
    ▪ The register is now loaded with data
    ▪ An address counter is incremented by 1
Write machine

• Write state machine
  ▪ Wait
    • The machine waits a STEP_EN signal
    • Until then it’s trapped on ‘wait’ state
  ▪ Store
    • The machine initiates a write transaction to the bus
  ▪ Wait4ACK
    • The machine waits for an ACK signal
    • ACK signals data was written in memory
  ▪ Terminate
    • An address counter is incremented by 1

Read/Write machines

• Read machine
  Read state machine (VHDL)
  Counter (RAM address) (VHDL)
  register

• Write machine
  Write state machine (VHDL)
  Counter (RAM address) (VHDL)
  VHDL constant data
### State machines in VHDL

**Constants represent states and transitions**

- **Human**: `std_logic_vector(1 downto 0) := "00";
- **Zombie**: `std_logic_vector(1 downto 0) := "01";
- **Healing**: `std_logic_vector(1 downto 0) := "10";
- **RIP**: `std_logic_vector(1 downto 0) := "11";
- **Cure**: `std_logic_vector(1 downto 0) := "00";
- **Bite**: `std_logic_vector(1 downto 0) := "01";
- **Decap**: `std_logic_vector(1 downto 0) := "10";

**Transfer function**

```vhdl
case state is
  when HUMAN =>
    if (input="00") then
      state := HUMAN;
    else
      if (input="01") then
        state := ZOMBIE;
      else
        state := RIP;
      end if;
    end if;
  when...
end case;
```
State machines in VHDL

• If not all state combinations are populated don’t forget

```
    case state is
        when RIP =>
            state <= RIP;
        when others => null;
    end case;
```

State machines in VHDL

• Defining outputs

```
... constant YES : std_logic := '1';
constant NO : std_logic := '0';
constant NONE : std_logic_vector(1 downto 0) := "00";
constant NURSE : std_logic_vector(1 downto 0) := "01";
constant MILITIA : std_logic_vector(1 downto 0) := "10";
constant RABI : std_logic_vector(1 downto 0) := "11";
...

    BrainActive <= NO when ((state = ZOMBIE) or (state = RIP))
                        else YES; -- ((state = HEAL) or (state = HUMAN))

    Support <= NURSE when (state = HEAL)
                  MILITIA when (state = ZOMBIE)
                  RABI when (state = RIP)
                        else NONE;
```
Handout 5 highlights

• The post lab report should be submitted in 2 different projects
  ▪ Write machine
  ▪ Read machine

• There should be simulations (2 for each project) that show
  ▪ All control signals of the state machine
  ▪ RDO(31:0), WDO(31:0)
  ▪ The machine’s state
  ▪ A full cycle of the machine
  ▪ A reset disrupted cycle

• RESA monitoring program
  ▪ To prove your design works present before and after data snapshots
    of the reading and writing activities
  ▪ To complete your proof present appropriate LA waveforms

Handout 4 highlights

• Write machine wave forms
**Reading**

- For the next recitation
  - Chapter 6: A Load/Store machine

**How to use the RESA program**

- New active part in Labels window: Memory labels.
- Memory Labels have the same structure as other labels: Name and address.
- Memory labels are the user’s tool to deal with the external RAM (read and write).
- New active configuration window: Memory Configuration.
- Memory Configuration deals with .cod files. The .cod file contains program codes and/or constants that have to be loaded to the RAM. More than one file can be loaded, but in case of overlapping the last loaded value is relevant.
How to use the RESA program

- New part in the window of Monitoring results: Memory Dump.
- Memory Dump show values of one page (32 words) from the external RAM. First page (addr = 0) is shown by default, but user can change it by writing the desired address in the field Memory Dump from:
- Field Write to selected label is used to write the desired value to the specific RAM address.
- The Write to selected label operation can be done only to label, previously defined as memory label.
- Graphics can be stored to file and viewed later with graphic editor.

Advanced Computer Structure Lab

Recitation 6: A Load/Store machine

Yaniv Eliezer
Contents

• A Load/Store machine
  ▪ Basic design of a memory access machine
    ▪ Execution of DLX programs which consist from
      ▪ Load instructions
      ▪ Store instructions

• Handout 6 highlights
  ▪ Design, Testing and Simulating

I-type instructions format

• There are two kinds of instruction formats
  ▪ I-type
  ▪ R-type

• Load and Store instructions are in I-type instruction format

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
</table>

Operation code | Operand Register | Operand Register | Immediate value field
Load/Store instructions

- For now Load/Store instructions only
  - lw/sw
    - load word/store word instructions
  - RD
    - Load instruction - Loading RD with the retrieved value
    - Store instruction - Storing RD with the register’s data
  - R0
    - We assign the second register to be R0 which is always zero!
  - Imm
    - Immediate memory address to read from

<table>
<thead>
<tr>
<th>Load/Store instructions</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw RD R0 imm</td>
<td>RD := M(imm)</td>
</tr>
<tr>
<td>sw RD R0 imm</td>
<td>M(imm) := RD</td>
</tr>
</tbody>
</table>

Load/Store instructions coding

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode[31:26]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>100011</td>
</tr>
<tr>
<td>Store</td>
<td>101011</td>
</tr>
</tbody>
</table>
Design pattern

**FSM**
- DLX controller (Sequential logic)

**Data path**
- Collection of registers and logic (Combinatorial logic)

**Output**

Memory access

- Load/Store state machine
  - Accesses memory during “fetch”, “load” and “store” states

Flowchart diagram
Memory Access Controller

- The R/W machine used 4 states for memory access
- We wish to do that only in 1
- We will bridge the DLX control with the I/O control logic using a new state machine – the Memory Access Controller (MAC)

MAC state machine

- **Signals**
  - **REQ**
    - either MW or MW is active (MR | MW)
  - **MR**
    - memory read, active during FETCH and LOAD states
  - **MW**
    - Memory write, active during STORE state
  - **BUSY**
    - Read/Write transaction is being performed (/ACK & REQ)
Memory access - waveforms

Wait4Req | Wait4Ack | Next | Wait4Req
----------|----------|------|----------
clk | | | |
req | | | |
ack | | | |
busy | | | |

Wait4Req | Wait4Ack | Next | Wait4Req
----------|----------|------|----------
\( /\text{REQ} \) | \( /\text{ACK} \) | | |
\( \text{REQ} \) | \( \text{ACK} \) | | |

DLX Data path and Control

<table>
<thead>
<tr>
<th>State</th>
<th>RTL Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>wait for step enable</td>
</tr>
<tr>
<td>FETCH</td>
<td>( I = M(\text{PC}) )</td>
</tr>
<tr>
<td>DECODE</td>
<td>( B = RD, PC = PC + 1 )</td>
</tr>
<tr>
<td>STORE</td>
<td>( M(\text{Imm}) = B )</td>
</tr>
<tr>
<td>LOAD</td>
<td>( C = M(\text{Imm}) )</td>
</tr>
<tr>
<td>WBI</td>
<td>( RD = C )</td>
</tr>
<tr>
<td>HALT</td>
<td>machine is stuck till reset</td>
</tr>
</tbody>
</table>
The GPR supports one of two operations each cycle:

- Write the value of the input C into R[Cadr] in case GPR_WE = 1
- Read the contents of the registers with indices Aadr and Badr

The outputs A and B are defined as follows:

\[
A = \begin{cases} 
R[Aadr] & \text{if } (Aadr \neq 0) \text{ and } (GPR\_WE = 0) \\
0 & \text{if } (Aadr = 0) \text{ and } (GPR\_WE = 0) \\
\text{arbitrary} & \text{else}
\end{cases}
\]

\[
B = \begin{cases} 
R[Badr] & \text{if } (Badr \neq 0) \text{ and } (GPR\_WE = 0) \\
0 & \text{if } (Badr = 0) \text{ and } (GPR\_WE = 0) \\
\text{arbitrary} & \text{else}
\end{cases}
\]
GPR Environment

- Register D
  - Clone register of A, B
  - Designed for debugging
  - RESA monitoring will read the contents of D by addressing it with Dadr

Address translation

- The logical addresses of the load/store machine are limited to 64K words (0x0000-0xFFFF), while the physical address space is 2M words (0x00000000-0x001FFFF)
- The **MMU Address translate** unit simply concatenates the 16 bits of the Load/Store machine with additional 16 bits constant which is no more than 0x001F
Handout 6 highlights

• Testing the Finite State Machine
  ▪ Goal – Make sure all FSM state transitions are correct

• Test vectors
  ▪ Generation of all input value combinations which will cause the control to traverse the control path starting from the INIT state
  ▪ Generation of all corresponding expected output combinations
Handout 6 highlights

**Simulation environment**

- Generation of the signals fed to the Load/Store machine is not an easy task
- The I/O SIMUL Module encapsulates the I/O control logic, the RESA's bus and the main memory
Handout 6 highlights

• Testing methodology
  ▪ Testing the combinatorial circuitry
  ▪ Testing the FSM by test vectors
    • Covering all transitions of the FSM by paths beginning in the INIT state
    • For each path, one needs to compute input values the will cause the control to traverse the path
    • One must also check if indeed the reset signal initializes the control, and if the step enable signal causes a transition to the fetch state
  ▪ Testing of whole RTL instructions off and on target
    • Off target simulation –
      ▪ Create your design on project “Home”
      ▪ Wire your design to the I/O logic and I/O SIMUL modules in a new schematic to simulate

Reading

• For the next recitation
  ▪ Chapter 7: A simplified DLX
Testing the Control of the Load/Store machine

- Test by simulation the transitions of the control of your design.
  - Testing Combinational Circuitry.
  - Testing finite state machines (*e.g.* Test Vectors)
    - “covering” all the transitions of the FSM by paths (beginning in the initial state).
    - For each path, one needs to compute input values that will cause the control to traverse the path.
    - Check if the BUS, Data Path and monitoring control signals meet requirements.
    - Check if indeed the reset signal initializes the control, and if the step enable signal causes a transition to the *fetch* state.
    - Check the control signals that are not directly involved in the L/S Machine Control, but are part of the Monitoring process, such as *states, stop_n, opcode, in_init, etc.*
      Take in account the change in the functionality of the *init*:
      ```
      in_init <= '1' (when state = Init OR state = Halt) else '0';
      ```
Simulation of the Control using Test vectors

- Simulation of the Control uses standard Xilinx tools: VHDL Test bench and ISE Simulator.
- Main differences are in the workflow:
  - Test vectors (table or equations or graphs) are prepared before or in parallel with the design.
  - List of the Test vectors includes not only combination of the inputs, but and outputs as well.
  - Inputs’ combination covers all the transitions of the FSM by paths (beginning in the initial state).
  - Outputs’ combination represents expected values generated by the Control.
- VHDL Test bench is created using the Inputs’ combination from the Test vectors.
- Within the Simulator the actual Outputs’ values are compared to the expected values from Test vectors.

Handout 6 highlights

<table>
<thead>
<tr>
<th>I/O</th>
<th>Signals</th>
<th>Expected values</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>PC</td>
<td>XXXXXXXX</td>
</tr>
<tr>
<td></td>
<td>RESET</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>STEP_EN</td>
<td>000000000000000</td>
</tr>
<tr>
<td>OUT</td>
<td>IN_INIT</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>OPCODE(5:0)</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>ADDR_MUX</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>IR_CE</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>B_CE</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>PC_STEP</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>MW</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>MR</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>C_CE</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>GPR_WE</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>STATE (mac)</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>MAC_BUSY</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>WR_N</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>AS_N</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>AS_N</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>STOP_N</td>
<td>000000000000000</td>
</tr>
<tr>
<td></td>
<td>ACK_N</td>
<td>000000000000000</td>
</tr>
</tbody>
</table>
Testing of the full Load/Store machine

- Building test environment with I/O SIMUL

- Simulation using I/O SIMUL
  Testing of RTL instructions.
  Testing executions of whole instructions.

I/O Simul Structure

- The I/O Simul block has a structure of 3 VHDL elements:
  the main, with support to the bus input/output signals.
  the external RAM emulator.
  the default RAM initialization code (package with CPU execution code).

- *The default* initialization code is set of 4 Load and 8 Store simplified instructions.
- This code can be used for the Load/Store machine and DLX as well.
- The initialization code can be changed for different CPUs.
- New initialization code can be prepared using the .lst file.
**I/O Simul default initialization code**

```
X'8c00111', -- 0x0 R1 := Mem[0x11]
X'8c00012', -- 0x1 R2 := Mem[0x12]
X'8c00013', -- 0x2 R3 := Mem[0x13]
X'8c00014', -- 0x3 R4 := Mem[0x14]
X'8a010091', -- 0x4 Mem[0x1d] := R1
X'ae020900', -- 0x5 Mem[0x2] := R2
X'ae030000', -- 0x6 Mem[0x1e] := R3
X'ae040000', -- 0x7 Mem[0x1f] := R4
X'ae040000', -- 0x8 Mem[0x1f] := R4
X'ae030000', -- 0x9 Mem[0x1e] := R3
X'ae020000', -- 0xa Mem[0x1d] := R2
X'ae010010', -- 0xb Mem[0x1c] := R1
X'ae000000', -- 0xc halt
X'00000000', -- 0xd work
X'00000000', -- 0xe work
X'00000000', -- 0xf work
X'00000000', -- 0x0 work
X'01234567', -- 0x11 constant = 0x1
X'01234567', -- 0x12 constant = 0x1234567
X'fedcba01', -- 0x13 constant = 0fedcba01
X'00000000', -- 0x14 constant = 0abcdef01
X'00000000', -- 0x15 N.C.
```

---

**Assembly language rules**

6.6.2 The assembly language rules

1. Everything that appears in a line after a `**` is a comment.
2. Every source code instruction is mapped as machine code to the next memory address of the previous one. The line p- 0x12315 is instruction for the Compiler only and means that next translated line will be mapped to address 0x12345. The instruction helps you to map the machine code in different memory locations. When the instruction is not used, the default value is p-0.
3. A label is defined by placing its name of up to 8 symbols at the beginning of a line followed by a `:` (in the example: `adr`). A label is a shortcut for an address. The address that corresponds to a label is the address of the line in which it is defined. Every occurrence of a label (even before the place in which the label is defined) in the instructions 'for', 'mov' and 'add' is replaced with the address that corresponds to the label, in the instructions 'beg' and 'bne' is replaced with the value IMM, calculated according to the following formula: IMM = address of the label-address of the branch instruction-1
4. dx means "place the constant x in the address that corresponds to the current line".
5. dy means "reserve y words in memory". The address of the next line is y plus the address of the current line. 
6. Instructions in the source assembly file are given in the mnemonics used in the description of the instruction set.
7. The empty symbol (space) is field separator within the instruction. After, but not instead, you can use Tab button in order to align your programs.

© Marko Markov
Assembly language example

Implementation of the Load/Store machine
Monitoring Data of the Load/Store machine

- External inputs:
  - **CPU Ram** (GPR): `Dadr` input and `D` registers output.
  - the Program Counter (PC): `AO` or dedicated PC output.
  - (define the appropriated value of the signal `MUX_SEL` for the INIT state when using AO as PC monitoring data.)
- The Monitored Signals: Defined in section 6.5.4.4 of Lab Notes.
- The default RAM initialization code (example.cod file with CPU execution code) is set of 4 Load and 8 Store simplified instructions and can be downloaded from the Lab site.

How to use the RESA program

- The slave label `PC`, like the label `STATUS`, is reserved word and have to be written in capital letters.
- In the window of the Monitoring results there is a new part: **Commands**, build of 3 columns and 9 lines.
- The columns are: the 32 bit of address, the 32 bit value of the corresponding address and the assembly source code of this value.
- The 9 lines show the assembly code execution. First 4 lines present the last 4 executed assembly instructions, while the next 5 – to be executed. The arrow points to the address defined by PC.
- Out of range is received when the read address is out of allowed range.
- No disassembly is received when the read value can’t be associated with any instructions’ opcode.
- The CPU Ram has no segmentation between instructions and data areas and some times data can be disassembled. The received source is meaningless.
Contents

- Simplified DLX architecture
  - DLX instruction set
  - Implementation
  - Control and path
    - And some of the data path
The load store machine

Dr. Strangelove has created a very strange machine at the urgent order of his obviously deranged dean.

The machine is kicked with a step ______, once all of its ports are correctly labeled.

While time seems to be stretched, an instruction from memory is being ______.

After the instruction is register loaded, it is almost immediately being ________.

In case the received instruction is ______, the machine will stop working with no result.

The load store machine

In case the received instruction is _____, a word from memory will leap like a toad, and into a specific register the word will be stowed.

In case the received instruction is _____, a registered word will come out with a roar. To memory the word will haste, making another word in memory replaced.

And after the machine finishes its cycle and returns to its ______ state, If Dr. Strangelove will not kick it again, the machine shall eternally wait.
Instruction format types

- There are two types of instruction formats
  - **I-type** instruction format (e.g. "lw RD1 RS1 0x00000001")
    
    6 5 5 16  
    Operation code  Operand Register  Operand Register  Immediate value field

  - **R-type** instruction format (e.g. "add RD1 RS1 RS2")
    
    6 5 5 5 6  
    Operand Register 1  Operand Register 2  Operand Register 3  Function code
**DLX instruction set**

- **Simplified DLX instruction set preliminaries**
  - **Imm**
    - The value of the immediate field in an I-type instruction
  - **Sext(Imm)**
    - 2’s complement sign extension of imm to 32 bits
  - **Memory is word addressable**
    - Words are the base blocks of memory reading and writing
    - Successive addresses in memory point to successive words

---

**DLX instruction set**

- **I-type**
  - **Load/Store instructions**
  - **Immediate instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw RS1 imm</td>
<td>RD := M(sext(imm)+RS1)</td>
</tr>
<tr>
<td>sw RD RS1 imm</td>
<td>M(sext(imm)+RS1) := RO</td>
</tr>
</tbody>
</table>

- **R-type**
  - **Shifting instructions**
  - **Computation instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll RS1</td>
<td>RD := RS1 &lt;&lt; 1</td>
</tr>
<tr>
<td>srl RS1</td>
<td>RD := RS1 &gt;&gt; 1</td>
</tr>
<tr>
<td>add RS1 RS2</td>
<td>RD := RS1 + RS2</td>
</tr>
<tr>
<td>sub RS1 RS2</td>
<td>RD := RS1 - RS2</td>
</tr>
<tr>
<td>and RS1 RS2</td>
<td>RD := RS1 AND RS2</td>
</tr>
<tr>
<td>or RS1 RS2</td>
<td>RD := RS1 OR RS2</td>
</tr>
<tr>
<td>xor RS1 RS2</td>
<td>RD := RS1 XOR RS2</td>
</tr>
</tbody>
</table>
DLX instruction set

- **I-type**
  - Testing instructions
  - Jumping instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi RD RS1 Imm</td>
<td>RD := 1, if condition is satisfied, RD := 0 otherwise</td>
</tr>
<tr>
<td>beqz RS1 Imm</td>
<td>PC = PC + 1 + sext(imm), if RS1 = 0</td>
</tr>
<tr>
<td>bnez RS1 Imm</td>
<td>PC = PC + 1, if RS1 ≠ 0</td>
</tr>
<tr>
<td>jr RS1</td>
<td>PC = RS1</td>
</tr>
<tr>
<td>jalr RS1</td>
<td>R31 = PC+4; PC = RS1</td>
</tr>
</tbody>
</table>

- **I-type**
  - Misc. instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>special-nop</td>
<td>causes transition to Init/Fetch states</td>
</tr>
<tr>
<td>halt</td>
<td>causes transition to HALT state</td>
</tr>
</tbody>
</table>

- **Binary encoding of the instruction set**
  - Tables 7.1, 7.2 in the course lab notes

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>111111000000000000000000000000000 = HALT!</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DLX instruction set

- Tables 7.1, 7.2

<table>
<thead>
<tr>
<th>BI[1:0]</th>
<th>Mnemonic</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 011</td>
<td>lw</td>
<td>ED = M(Ext(MEM) + RSI)</td>
</tr>
<tr>
<td>101 011</td>
<td>sw</td>
<td>M(Ext(MEM) + RSI) = ED</td>
</tr>
</tbody>
</table>

Arithmetic, Logical Operation

| 001 011 | addi     | ED = RSI + (Ext(MEM)) |

Test Set Operation

| 011 rel | slti     | ED = (RS1 rel (Ext(MEM))) |
| 011 010 | segi     | ED = (RS1 = (Ext(MEM))) |
| 011 011 | srqi     | ED = (RS1 ≥ (Ext(MEM))) |
| 011 100 | sbi      | ED = (RS1 < (Ext(MEM))) |
| 011 101 | sei      | ED = (RS1 = (Ext(MEM))) |
| 011 110 | slei     | ED = (RS1 ≤ (Ext(MEM))) |

Control Operation

| 000 100 | beqz     | PC = PC + 1; (RS1 = 0 & Ext (MEM) = 0) |
| 000 101 | bnez     | PC = PC + 1; (RS1 = 0 & Ext (MEM) = 0) |
| 010 110 | j         | PC = RS1 |
| 010 111 | jpl       | RS1 = PC + 1; PC = RS1 |

Miscellaneous Instructions

| 110 000 | special-op | No operation |
| 111 111 | halt       | Stop program |

Implementation

- Data path of the simplified DLX
Implementation

- Architectural registers
  - 32 bits wide
  - 32 General Purpose Registers
    - R0-R31
    - R0 always holds the 0x00000000 value
    - Identical to the Load/Store machine implementation
  - PC
    - Program counter register
  - IR
    - Instruction register
  - MAR, MDR, A, B, C
    - Special registers

Implementation

- Arithmetic Logic Unit environment
  - 2’s complement integer addition and subtraction
  - Bitwise logical instructions
    - AND
    - OR
    - XOR
  - Comparison instructions
Control path

- Memory access
  - Through the MAC
- Reset signal
  - INIT state
- Step_en signal
  - I/O control logic
- Busy signal
  - MAC

- D1...D12
  - According to instruction decoding
- Else
  - Illegal instruction
- Branch taken
  - In the event where a condition of a conditional branch is satisfied/unsatisfied
## Control path

### Instruction decoding table

- Table 7.5 on Lab's handbook

<table>
<thead>
<tr>
<th>Nontrivial DNF</th>
<th>Target State</th>
<th>$R[31:26]$</th>
<th>$R[5:0]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>IntFetch</td>
<td>110****</td>
<td>*****</td>
</tr>
<tr>
<td>D2</td>
<td>Add</td>
<td>0000**</td>
<td>0**</td>
</tr>
<tr>
<td>D4</td>
<td>Shift</td>
<td>0000**</td>
<td>0**</td>
</tr>
<tr>
<td>D5</td>
<td>AddI</td>
<td>001**</td>
<td>******</td>
</tr>
<tr>
<td>D6</td>
<td>TestI</td>
<td>011**</td>
<td>******</td>
</tr>
<tr>
<td>D7</td>
<td>Add Comp</td>
<td>10*****</td>
<td>*******</td>
</tr>
<tr>
<td>D8</td>
<td>JR</td>
<td>010(*)**</td>
<td>******</td>
</tr>
<tr>
<td>D9</td>
<td>SavePC</td>
<td>010(<strong>)</strong>*</td>
<td>*******</td>
</tr>
<tr>
<td>D12</td>
<td>Branch</td>
<td>0001**</td>
<td>******</td>
</tr>
<tr>
<td>D13</td>
<td>Copy GPR,MDR</td>
<td>0000**</td>
<td>*******</td>
</tr>
<tr>
<td>$D13$</td>
<td>Load</td>
<td>0000**</td>
<td>*******</td>
</tr>
</tbody>
</table>

### Control signals table

- Communication signals between data path and control

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AEB[0]$</td>
<td>0</td>
<td>Control the functionality of ALU</td>
</tr>
<tr>
<td>J</td>
<td>0</td>
<td>Register clock enable</td>
</tr>
<tr>
<td>$Sel[10]$</td>
<td>00</td>
<td>PC</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>MDR</td>
</tr>
<tr>
<td>$Sel[11]$</td>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>$PUSH[0]$</td>
<td>0</td>
<td>ALU</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Status</td>
</tr>
<tr>
<td>$MCR[0]$</td>
<td>0</td>
<td>COPF</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>DC</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>PC</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>MAR</td>
</tr>
<tr>
<td>Shift</td>
<td>0</td>
<td>Replace Shift Instruction</td>
</tr>
<tr>
<td>Right</td>
<td>1</td>
<td>Shift to the right</td>
</tr>
<tr>
<td>Add</td>
<td>0</td>
<td>Force an addition</td>
</tr>
<tr>
<td>Test</td>
<td>1</td>
<td>Force a test (in the ALU)</td>
</tr>
<tr>
<td>MR</td>
<td>0</td>
<td>Memory Read</td>
</tr>
<tr>
<td>MSW</td>
<td>1</td>
<td>Memory Write</td>
</tr>
<tr>
<td>GPR, WFE</td>
<td>0</td>
<td>GPR, WFE write enable</td>
</tr>
<tr>
<td>Instr</td>
<td>0</td>
<td>Instr instruction</td>
</tr>
<tr>
<td>RAMP</td>
<td>1</td>
<td>Ramp and link</td>
</tr>
</tbody>
</table>
Control path

- State control signals table
  - The active control signals activated at each state
  - Table 7.3 on Lab's handbook

<table>
<thead>
<tr>
<th>Name</th>
<th>RTL Instruction</th>
<th>Active Control Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>( R \sim (PC) )</td>
<td>ME, BcRe</td>
</tr>
<tr>
<td>Decode</td>
<td>( A = RS1 )</td>
<td>Ace, Bc, Ssize[1], Ssize[0] PCo, add</td>
</tr>
<tr>
<td>Addr</td>
<td>( C = A \uparrow F )</td>
<td>Ssize[0], Cte</td>
</tr>
<tr>
<td>Text</td>
<td>( C = (A \downarrow \text{imm}) )</td>
<td>Ssize[0], Ssize[0], Cte, test, byte</td>
</tr>
<tr>
<td>Add (add)</td>
<td>( C = A + \text{imm} )</td>
<td>Ssize[0], Ssize[0], Cte, add, byte</td>
</tr>
<tr>
<td>Shift</td>
<td>( C = A \text{ shift} )</td>
<td>Ssize[0], Cte</td>
</tr>
<tr>
<td>Load</td>
<td>( MDX = MD(\varnothing) )</td>
<td>MD(0), Aux, ME, MDsize</td>
</tr>
<tr>
<td>Store</td>
<td>( X \uparrow (A) = \text{MD} \times \text{addr} )</td>
<td>Faci, ME</td>
</tr>
<tr>
<td>CopyMD/KCC</td>
<td>( \text{C\rightarrow MD/\uparrow kCC} )</td>
<td>Ssize[0], Ssize[0], Ssize[0], Ssize[0], LINTTest, Cte</td>
</tr>
<tr>
<td>CopyMD/INDE</td>
<td>( \text{MDR = BcRef} )</td>
<td>Ssize[0], Ssize[0], LINTTest, MDRef</td>
</tr>
<tr>
<td>WBT</td>
<td>( RD = C \text{ (byte)} )</td>
<td>GPB, WE, byte</td>
</tr>
<tr>
<td>Branch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trun</td>
<td>( PC = PC \uparrow \text{imm} )</td>
<td>Ssize[0], Add, PCo</td>
</tr>
<tr>
<td>JR</td>
<td>( PC = A )</td>
<td>Ssize[0], Ssize[0], add, PCo</td>
</tr>
<tr>
<td>Save PC</td>
<td>( C = PC )</td>
<td>Ssize[0], Add, Cc</td>
</tr>
<tr>
<td>JALR</td>
<td>( PC = A )</td>
<td>Ssize[0], Ssize[0], add, PCo</td>
</tr>
<tr>
<td>Add</td>
<td>( RJ = C )</td>
<td>GPB, WE, byte</td>
</tr>
</tbody>
</table>

**lw RD RS1 imm**

<table>
<thead>
<tr>
<th>Name</th>
<th>RTL Instruction</th>
<th>Active Control Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync</td>
<td></td>
<td>ME, BcRe</td>
</tr>
<tr>
<td>Fetch</td>
<td>( R \sim (PC) )</td>
<td>ME, BcRe</td>
</tr>
<tr>
<td>Decode</td>
<td>( A = RS1 )</td>
<td>Ace, Bc, Ssize[1], Ssize[0] PCo, add</td>
</tr>
<tr>
<td>Addr</td>
<td>( C = A \uparrow F )</td>
<td>Ssize[0], Cte</td>
</tr>
<tr>
<td>Text</td>
<td>( C = (A \downarrow \text{imm}) )</td>
<td>Ssize[0], Ssize[0], Cte, test, byte</td>
</tr>
<tr>
<td>Add (add)</td>
<td>( C = A + \text{imm} )</td>
<td>Ssize[0], Ssize[0], Cte, add, byte</td>
</tr>
<tr>
<td>Shift</td>
<td>( C = A \text{ shift} )</td>
<td>Ssize[0], Cte</td>
</tr>
<tr>
<td>Load</td>
<td>( MDX = MD(\varnothing) )</td>
<td>MD(0), Aux, ME, MDsize</td>
</tr>
<tr>
<td>Store</td>
<td>( X \uparrow (A) = \text{MD} \times \text{addr} )</td>
<td>Faci, ME</td>
</tr>
<tr>
<td>CopyMD/KCC</td>
<td>( \text{C\rightarrow MD/\uparrow kCC} )</td>
<td>Ssize[0], Ssize[0], Ssize[0], Ssize[0], LINTTest, Cte</td>
</tr>
<tr>
<td>CopyMD/INDE</td>
<td>( \text{MDR = BcRef} )</td>
<td>Ssize[0], Ssize[0], LINTTest, MDRef</td>
</tr>
<tr>
<td>WBT</td>
<td>( RD = C \text{ (byte)} )</td>
<td>GPB, WE, byte</td>
</tr>
<tr>
<td>Branch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trun</td>
<td>( PC = PC \uparrow \text{imm} )</td>
<td>Ssize[0], Add, PCo</td>
</tr>
<tr>
<td>JR</td>
<td>( PC = A )</td>
<td>Ssize[0], Ssize[0], add, PCo</td>
</tr>
<tr>
<td>Save PC</td>
<td>( C = PC )</td>
<td>Ssize[0], Add, Cc</td>
</tr>
<tr>
<td>JALR</td>
<td>( PC = A )</td>
<td>Ssize[0], Ssize[0], add, PCo</td>
</tr>
<tr>
<td>Add</td>
<td>( RJ = C )</td>
<td>GPB, WE, byte</td>
</tr>
</tbody>
</table>
Handout 7 Highlights

- Implementation
  - Control path, signals
  - Data path
- No preliminary report
Test vectors

<table>
<thead>
<tr>
<th>Machine cycle</th>
<th>Input Signals</th>
<th>Expected values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input NCC</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12</td>
<td></td>
</tr>
<tr>
<td>Input RESET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input STEP_EN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output IN_INIT</td>
<td></td>
<td>INIT INIT FETCH... AS_N</td>
</tr>
<tr>
<td>Output PCODE(5:0)</td>
<td>000000 000000 000000 000000 000000</td>
<td>100011 100011 100011 100011 100011</td>
</tr>
<tr>
<td>Output IN_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output B_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output C_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output MDR_SEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output MDR_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output TEST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output DINTSEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output PC_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output MAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output S1SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output S2SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output DINTSEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output PC_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output MAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output S1SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output S2SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output DINTSEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output PC_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output MAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output S1SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output S2SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output DINTSEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output PC_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output MAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output S1SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output S2SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output DINTSEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output PC_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output MAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output S1SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output S2SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output DINTSEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output PC_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output MAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output S1SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output S2SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output DINTSEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output PC_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output MAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output S1SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output S2SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output DINTSEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output PC_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output MAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output S1SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output S2SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output DINTSEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output PC_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output MAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output S1SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output S2SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output DINTSEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output PC_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output MAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output S1SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output S2SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output DINTSEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output PC_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output MAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output S1SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output S2SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output DINTSEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output PC_CE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output MAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output SRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output S1SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Output S2SEL(1:0)</td>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

Handout 7 Highlights

- This handout includes 4 sections handed together
- Recommended schedule and guidelines are within the handout
- The weight of the handout is 400 points
- Questions 2 and 4 require the approval of the lab’s engineer
- The approval form is located on the website, attach it signed to the submitted project
- Add to your report the timing report produced by Xilinx which verifies your design meets with timing requirements
- The programming assignment will be published very soon
Contents

• Simplified DLX architecture
  ▪ Handout 7 highlights
  ▪ Data path building blocks
    • GPR
    • IR
    • MMU
    • ALU
Building blocks

• The GPR environment
  ▪ Identical to the Load/Store machine
  ▪ An AEQZ output port should be implemented
  ▪ A Cadr signal mechanism should be implemented. 
    according to the JLINK control signal, R31 address should be 
    enforced.

Building blocks

• The IR environment
  ▪ Inputs
    • CLK
    • IR_CE
    • IR_IN[31:0]
  ▪ Outputs
    • IR_OUT[31:0]

• The PC environment
  ▪ 32 bit register with a RESET port
Building blocks

• The MMU
  • Input
    • \(AO[31:0]\)
  • Output
    • \(0^8 \& AO[23:0]\)

• ALU environment
  • Inputs
    • \(A[31:0], B[31:0], ALUF[2:0], TEST, ADD\)
  • Outputs
    • \(ALU\_OUT[31:0]\)

Reminder 1

\(\text{Let } A[n-1:0], B[n-1:0] \in \{0,1\}^n\)
\(\text{Where } [\cdot] \text{ denotes 2's comp. representation}\)
\(\text{Then } A[n-1:0] - B[n-1:0] = [A[n-1:0]] + [B[n-1:0]] + 1\)

Reminder 2

\(\langle A[n-1:0] \rangle + \langle B[n-1:0] \rangle + C[0] = \langle C[n] \cdot S[n-1:0] \rangle\)
\(NEG = XOR_5(A[n-1] B'[n-1] C[n])\)
\(B'[n-1] = XOR(B[n-1] \cdot ADD)\)
Building blocks

- ALU environment
  - We recommend using the ADDSUB16 ("ADDSU16") component

Building blocks

- ALU environment
  - We suggest using three 16-bit adders/subtractors from the Xilinx library (ADDSU16) to construct a 32-bit Conditional Sum Adder
ALU

- ALU functionalities in the different DLX control states

<table>
<thead>
<tr>
<th>State</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>add</td>
</tr>
<tr>
<td>Alu</td>
<td>op, op=add/sub/and/or/xor.</td>
</tr>
<tr>
<td>AluI</td>
<td>add</td>
</tr>
<tr>
<td>TestI</td>
<td>rel, rel=lt, eq, gt, le, ge, ne.</td>
</tr>
<tr>
<td>Adr. Comp.</td>
<td>add</td>
</tr>
<tr>
<td>B. Taken</td>
<td>add</td>
</tr>
<tr>
<td>JR</td>
<td>add</td>
</tr>
<tr>
<td>SavePC</td>
<td>add</td>
</tr>
<tr>
<td>JALR</td>
<td>add</td>
</tr>
</tbody>
</table>

ALU

- Input
  - ALUF[2:0]
- Control signals
  - ADD (Decode, AluI, Adr. Comp, B. taken, SavePC, JR, JALR)
  - TEST (TestI)

<table>
<thead>
<tr>
<th>ALUF[2:0]</th>
<th>Arithmetic logic operations</th>
<th>ALUF[2:0]</th>
<th>Test conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>add</td>
<td>001</td>
<td>gt</td>
</tr>
<tr>
<td>010</td>
<td>sub</td>
<td>010</td>
<td>eq</td>
</tr>
<tr>
<td>110</td>
<td>and</td>
<td>011</td>
<td>ge</td>
</tr>
<tr>
<td>101</td>
<td>or</td>
<td>100</td>
<td>lt</td>
</tr>
<tr>
<td>100</td>
<td>xor</td>
<td>101</td>
<td>ne</td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
<td>le</td>
</tr>
</tbody>
</table>

ALU

• Building blocks

ALU

• The Comparator in details
Recitation 7: DLX assembly Part 3

Language hierarchy

- High-level programming language (C, C++)
- Low-level programming language (Assembly, RTL)
- Machine language (binary)
DLX assembly

• Conversion to DLX assembly
  - Example 1

\[
\begin{align*}
\{f, g, h, i, j, k\} & \rightarrow \{R16..R21\} \\
\text{If } (i==j) & \rightarrow \text{xor } R1 \ R19 \ R20 \\
\text{goto } L1; & \rightarrow \text{beqz } R1 \ 1 \\
f=g+h; & \rightarrow \text{add } R16 \ R17 \ R18 \\
L1: f=f-i & \rightarrow \text{sub } R16 \ R16 \ R19
\end{align*}
\]

DLX assembly

• Conversion to DLX assembly
  - Example 2

\[
\begin{align*}
\text{Astart} & \rightarrow & \& A[0] \\
\{f, g, h, i, j, k\} & \rightarrow \{R16..R21\} \\
\text{LOOP: } g = g + A[i]; & \rightarrow & \text{addi } R4 \ R0 \ \text{Astart} \\
& \rightarrow & \text{add } R1 \ R4 \ R19 \\
& \rightarrow & \text{lw } R2 \ R1 \ 0 \\
& \rightarrow & \text{add } R17 \ R17 \ R2 \\
& \rightarrow & \text{add } R19 \ R19 \ R20 \\
i = i + j; & \rightarrow & \text{xor } R3 \ R19 \ R18 \\
\text{if ( } i \neq h \text{ ) goto LOOP;} & \rightarrow & \text{bnez } R3 \ -6
\end{align*}
\]
**DLX assembly**

- Addition of a new DLX instruction
  - We wish to add a new instruction with minimal changes
    - Chkbit17  RD  RS1  imm
      
      \[
      RD = \begin{cases} 
        0^{31} \cdot 1, & \text{if } RS1[17] = 1 \\
        0^{32}, & \text{else} 
      \end{cases}
      \]
    - Our TODO list
      - Change the data path
      - Change the control path
      - Assign a new RTL instruction

**DLX assembly**

- Addition of a new DLX instruction
  - Data path changes
    - Addition of a MUX component
    - Zero padding
    - Sampling bit17 of register A
**DLX assembly**

- Addition of a new DLX instruction
  - CHK17 new state

If OPCODE == OPCODE(chkbit17)

**DLX assembly**

- Addition of a new DLX instruction
  - New RTL instruction

<table>
<thead>
<tr>
<th>Name</th>
<th>RTL Instruction</th>
<th>Active Control Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>A = RHI</td>
<td>ME, Be</td>
</tr>
<tr>
<td>Decode</td>
<td>A = RSI</td>
<td>Ace, Be, Siat[1], Siat[0]</td>
</tr>
<tr>
<td></td>
<td>S = RSI</td>
<td>FCo, add</td>
</tr>
<tr>
<td></td>
<td>PC = PC + 1</td>
<td></td>
</tr>
<tr>
<td>Alu</td>
<td>C = A op B</td>
<td>Siat[0], Cte</td>
</tr>
<tr>
<td></td>
<td>New instruction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C = A + imm</td>
<td>Siat[0], Siat[0], Cte, add, Bepe</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift</td>
<td>C = A shift as</td>
<td>Siat[1], Cte</td>
</tr>
<tr>
<td></td>
<td>3b = L<a href="1">d</a></td>
<td>DBTrG0, shift (right)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addr Comp</td>
<td>MAR = A + imm</td>
<td>Siat[0], Siat[1], MARe, add</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>MDR = M(MAR)</td>
<td>MDr, cc, Art, ME, MDrxt</td>
</tr>
<tr>
<td>Store</td>
<td>M(MAR) = MDR</td>
<td>Art, MDr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CopyMDRCC</td>
<td>C = MDR[&lt;]&lt;8</td>
<td>Siat[0], Siat[1], Siat[1], Lsntre, Cte</td>
</tr>
<tr>
<td>CopyMDRDEC</td>
<td>MDR = B(&lt;&lt;&lt;8)</td>
<td>Siat[1], Siat[1], DBTrG0, MDRec</td>
</tr>
<tr>
<td>Write</td>
<td>RD = C(r-type)</td>
<td>GFP, WE</td>
</tr>
<tr>
<td></td>
<td>RD = C(t-type)</td>
<td>GFP, WE, type</td>
</tr>
<tr>
<td>Branch</td>
<td>Branch taken?</td>
<td></td>
</tr>
<tr>
<td>Braken</td>
<td>PC = PC + imm</td>
<td>Siat[0], add, FCo</td>
</tr>
<tr>
<td></td>
<td>PC = A</td>
<td>Siat[0], Siat[1], add, FCo</td>
</tr>
<tr>
<td></td>
<td>C = PC</td>
<td>Siat[1], add, Cte</td>
</tr>
<tr>
<td>JALR</td>
<td>PC = A</td>
<td>Siat[0], Siat[1], add, FCo</td>
</tr>
<tr>
<td></td>
<td>RJ = C</td>
<td>GFP, WE, jmp</td>
</tr>
</tbody>
</table>

**CHK17**

\[ C = 0^{31}A[17] \]

**Chk17, Cce**