Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

ECE 312

Computer Organization and Design

Spring 2001

MP2: The LC-2 Processor with a Unified 2-Way Set Associative Cache PART I

Version 3.0

J. Strasser
S. J. Patel

TABLE OF CONTENTS

Chapter 1: Introduction

This machine problem involves the design and simulation of a simple, non-pipelined processor that implements the entire LC-2 instruction set architecture (actually, we won't require the implementation of the RTI instruction). In addition, you are required to add a simple set associative cache to the memory hierarchy (Note: caches will be covered in class). You will design the processor using the LC-2 instruction set description included in this handout and using your design from MP1 as a starting point.

Unlike the previous MP, this handout is only intended to help you get started. You are responsible for implementing the additions and modifications to the MP1 design on your own. We will cover datapath, control, and cache design in lecture. In addition, Section 5.4 of Patterson and Hennessey provides an example of a non-pipelined design, which might supply you with ideas on how to proceed. This MP has been divided into two components to help you manage the complexity of this project: in the first checkpoint, you will deliver a working LC-2 processor. In the second (and final) checkpoint, you will deliver the same processor augmented with a unified set associative cache.

The first checkpoint requires the correct implementation of the whole LC-2 instruction set (implementing the RTI instruction is not mandatory) and is **due Friday, February 16th at 5pm**. Correctness will be verified with a test program that will be released near the checkpoint due date. **You are strongly encouraged to write your own test code to test your implementation and not wait until we release ours.** The first checkpoint is worth 15% of the MP2 grade.

The final checkpoint will require the correct implementation of the LC-2 instruction set with a unified set associative cache and is **due Wednesday, March 7th at 3pm**. The final turn-in is worth 85% of the MP2 grade. The specifics of the required cache configuration will be provided after we cover caches in class.

Grades will be based on the correctness of design, style of design, thoroughness of verification, and quality of documentation. Specifics of what to turn in are covered in Chapter 5. MPs can be turned in up to two days late with an automatic 20 point deduction (out of 100). MPs will not be accepted after two days past the due date.

The remainder of this MP2 write-up contains helpful reference information for completing the MP. The second chapter contains a description of the 16 instructions in the LC-2 instruction set. The third chapter contains a high-level description of the design. Chapter four lists some design constraints and Chapter five lists the specifics of what to turn in for Checkpoint 1.

Chapter 2: The LC-2 Instruction Set Architecture

Overview

For this project, you will be entering the VHDL design (using Renoir) of a non-pipelined version of the LC-2 ISA. A complete list of the LC-2 instruction set is shown below. For a more complete description of the LC-2 ISA, refer to the ISA reference manual distributed in class and available on the web page.

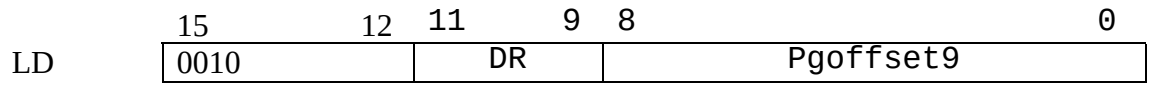
All sixteen instructions are 16 bits in length, having a format where bits [15:12] contain the opcode. The LC-2 ISA is a load-store ISA, meaning data values must be brought into the General-Purpose Register file before they can be operated upon. Each general-purpose register is 16 bits in length (meaning that the LC-2 is a 16-bit ISA). The address space of LC-2 is accessed using 16 bits (meaning that the LC-2 memory consists of 2^{16} locations). Each location contains a single word (meaning that the LC-2 memory is word-addressable). The General-Purpose Register file contains 8 registers.

The LC-2 program control is maintained by the Program Counter (PC). The PC is a 16-bit register that contains the address of the **next** instruction to be fetched.

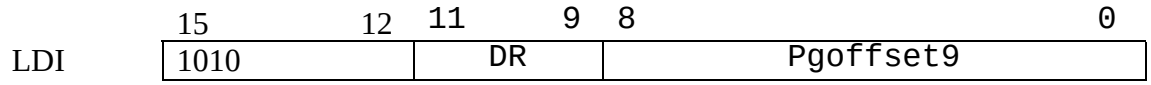
Data Movement Instructions

The memory reference instructions are used to transfer values between GP registers and the memory system. The load word instruction reads a 16-bit value from the memory system and places it into a general-purpose register. The store word instruction takes a value from a general-purpose register and writes it into the memory system.

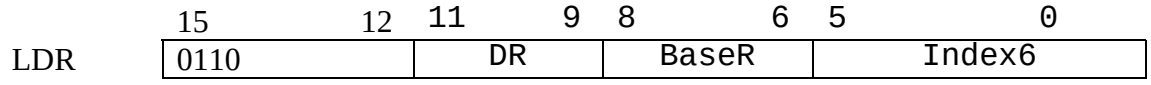
The format of the load direct instruction, or *LD*, is shown below. The opcode of the *LD* instruction (bits [15:12]) is 0010. The effective address (the address of the memory location that is to be read) is the memory location specified by the 9-bit *page offset* on the same page. One word (16 bits) is read from memory and put into register *DR*. For example, a LD into register R3 from memory address 0x1000 will cause the word at 0x1000 to be placed into R3. The condition codes are set, based on whether the value loaded is negative, zero, or positive.



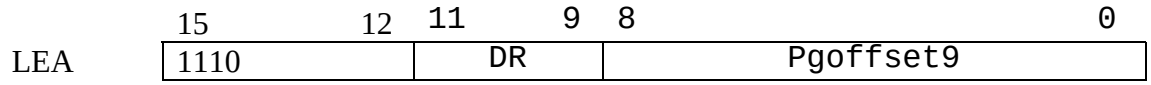
The format of the load indirect instruction, or *LDI*, is shown below. The opcode of the *LDI* instruction (bits [15:12]) is 1010. The effective address (the address of the memory location that is to be read) is *the address contained in the memory location* specified by the top 7 bits of the program counter concatenated with the 9-bit *page offset* field. Using the address formed by concatenating the PC with pgoffset9 is used to read one word (16 bits) from memory. This word is used as the effective address from where to read the actual operand. This operand is placed into register *DR*. The condition codes are set, based on whether the value loaded is negative, zero, or positive.



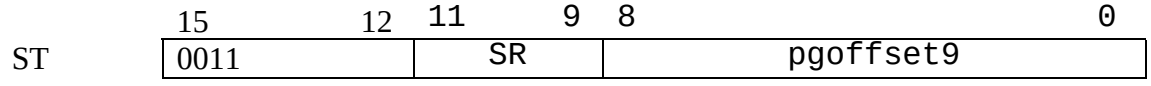
The format of the load base + index instruction, or *LDR*, is shown below. The opcode of the *LDR* instruction (bits [15:12]) is 0110. The effective address (the address of the memory location that is to be read) is the memory location specified by the contents of BaseR added to the value of Index6 zero-extended to 16 bits. One word (16 bits) is read from memory and put into register *DR*. The condition codes are set, based on whether the value loaded is negative, zero, or positive.



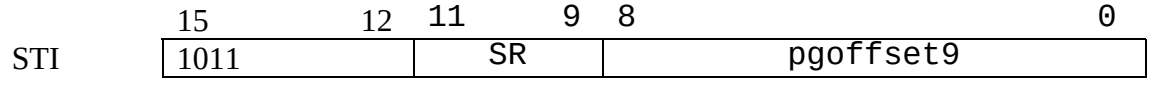
The format of the load effective address instruction, or *LEA*, is shown below. The opcode of the *LEA* instruction (bits [15:12]) is 1110. The value loaded into DR is the top 7 bits of the program counter concatenated with the 9-bit *page offset* field. The condition codes are set, based on whether the value loaded is negative, zero, or positive.



The format of the store instruction, *ST*, is shown below. The opcode of this instruction is 0011. As with the load instruction (*LD*), the effective address is the memory location specified by the 9-bit *page offset* on the same page. The instruction causes the value in register *SR* to be stored into the effective address specified by PC[15:9] concatenated with pgoffset9. No condition codes are set.

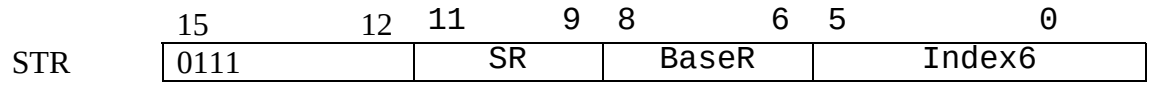


The format of the store indirect instruction, *STI*, is shown below. The opcode of this instruction is 1011. The address where the contents of SR are stored is formed as follows. First, an address is formed by concatenating the upper 7 bits of the PC with the 9-bit pgoffset9 field. Then the contents of that memory location specify the location to store the contents of SR. No condition codes are set.



The format of the store base + offset instruction, *STR*, is shown below. The opcode of this instruction is 0111. The contents of SR are stored in the memory location specified by adding

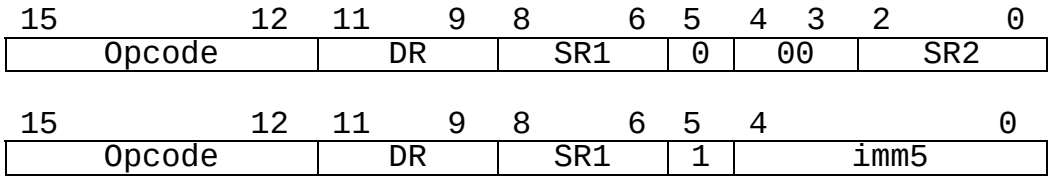
the contents of the BaseR with the index6 zero-extended to sixteen bits. No condition codes are set.



Operate Instructions

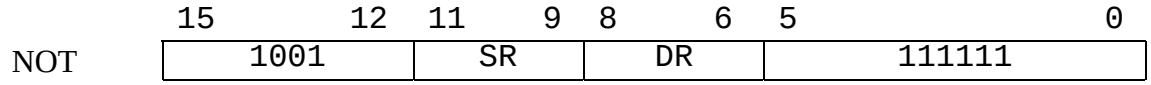
LC-2 has three integer operate instructions: ADD, logical AND, and NOT (logical inverse). We will cover these two operations first, and return to the NOT instruction later.

There are two modes of operations for the ADD and AND instructions. They are shown below.



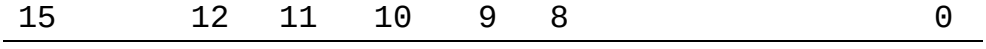
The opcode of the ADD instruction is 0001, and the opcode of the AND instruction is 0101. When bit 5 is 0, the values in registers SR1 and SR2 are operated upon and the result stored in register DR. When bit 5 is 1, the values in register SR1 and the sign extended imm5 field are operated upon and the result stored in register DR. In other words, if bit5 of the instruction is set, the second operand is specified as an immediate value. The condition codes are set, based on whether the value loaded is negative, zero, or positive.

This is the format of the NOT instruction. It performs a bitwise complement on SR and stores the result in DR. The condition codes are set, based on whether the value loaded is negative, zero, or positive. Note that the last six bits must be 1.

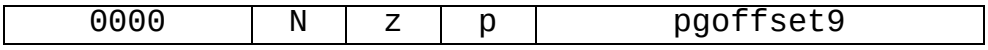


Control Instructions

The LC-2 branch instruction, BR, causes program control to branch to a specified address. The format of the instruction is given below. Specifically, it works as follows: if the n, z, and/or p bits are set to 1 within the instruction, and the corresponding condition codes are set in the processor, the processor will *take* the branch. This is done by loading the address specified by the page offset (pgoffset9) on the same page as the branch instruction into the PC (i.e., PC <- PC[15:9] || IR[8:0]).

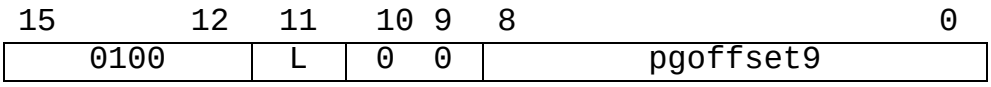


BR



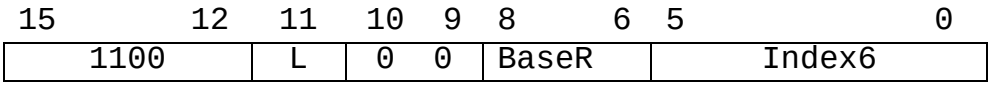
The LC-2 jump/jump-to-subroutine instruction, JMP/JSR, causes program control to unconditionally branch to an address specified by upper 7 bits of the PC concatenated with the pgoffset9 field. The opcode for the instruction is 0100. If the L or link bit is set to one, then the current PC is stored in R7. If the L bit is not set, then it is not saved.

JSR



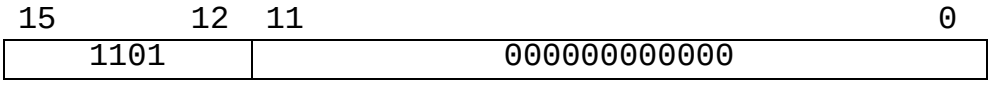
The LC-2 also contains a base+offset version of the JMP/JSR instruction. JMPR/JSRR causes program control to unconditionally branch to an address specified by adding the zero-extension of index6 to the contents of the BaseR. The opcode for the instruction is 1100. If the L or link bit is set to one, then the current PC is stored in R7. If the L bit is not set, then it is not saved.

JSRR



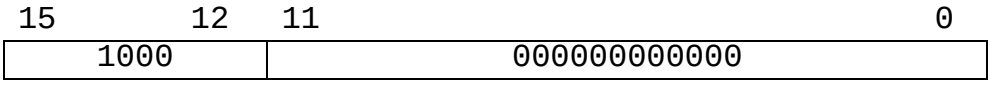
The LC-2 return from subroutine instruction, RET, causes program control to unconditionally branch to the address specified by the contents of R7. The opcode for the instruction is 1101.

RET



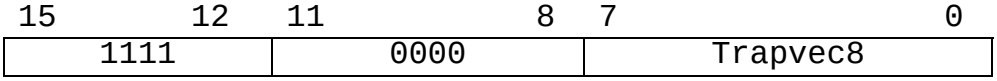
The LC-2 return form interrupt instruction, RTI, causes program control to return to an interrupted program. The opcode is 1000. Two elements are popped off the stack, which is referenced in memory by R6. The second element popped off the stack is then loaded into the PC. **YOU NEED NOT IMPLEMENT RTI FOR MP2.**

RTI



The LC-2 trap instruction, TRAP, causes program control to switch to a system routine. The opcode is 1111. The current PC is stored in R7 and the PC is loaded with the memory contents at the location referenced by zero-extension of trapvect8 (`PC<-MEM[ZEXT(Trapvec8)]`).

TRAP



Chapter 3: Specifications

Signals

The microprocessor communicates with the outside world through an address bus, a data bus, and five control signals, as well as a clock.

External inputs

RESET_L
: Active low signal that puts the processor in the initial state. Once in this state, only a START_H signal will cause the processor to leave the initial state.

START_H
: Active high signal that causes the processors to execute the instruction located at memory address 0x00000000. It must stay high for at least one clock cycle. Afterward, START_H may change state without affecting the microprocessor's operation.

CLK
: All components of the design are active on the rising edge.

Memory Subsystem Signals

ADDRESS(15:0)
: Memory is accessed using this 16-bit signal. It specifies the address that is to be read or written.

DATAIN(15:0)
: 16-bit data bus receiving data from memory.

DATAOUT(15:0)
: 16-bit data bus sending data to memory.

MREAD_L
: Active low signal that tells memory that the address is valid and that the processor is trying to perform a memory read.

MWRITE_L
: Active low signal that tells memory that the address is valid and that the processor is trying to perform a memory write.

MRESP_H

Active high signal generated by memory indicating that the memory has finished the requested operation.

The convention that is used for all signal names (except data paths, e.g. ADDRESS, DATA, etc.) is to append an underscore and polarity to the end of the signal name. For example, XYZ_H indicates that the signal XYZ is active when high.

Bus Control Logic

The memory system is asynchronous, meaning that the processor waits for the memory to respond to a request before completing the access cycle. In order to meet this constraint, inputs to the memory subsystem should be held constant until the memory subsystem responds. In addition, outputs from the memory subsystem should be latched if necessary.

The processor sets the MREAD_L control signal active (low) when it needs to read data from the memory. The processor sets the MWRITE_L signal active when it is writing to the memory. The memory activates the MRESP_H signal when it has completed the read or write request. We assume the memory response will always occur so the processor never has an infinite wait. **For this design, memory will respond 500 ns after the processor has initiated a read/write request.**

Reset Logic

It is necessary to be able to reset the processor (put the processor in a known state). An external signal, RESET_L, can put the microprocessor to a known state by clearing the PC register and sending your controller into a reset state. The controller will remain in the reset state until START_H is received. Then the instruction at location 0x0000 of the main memory is executed.

Cache Design

The final component of this MP is to add a unified set associative cache to your design. As you will learn in lecture, the data cache will be used to speed up the effective response time of the memory system. The exact requirements and hints for the cache design will be posted on the ECE312 web page after Exam1.

Do not attempt this until you have a working design capable of executing correctly all the required instructions of the LC-2 instruction set.

Remember that material from this machine problem will be used in the last MP and may appear on exams. If you are having difficulty understanding an aspect of the MP, ask a TA for assistance.

Setup

You will need to create a new library for this MP and copy over your design from MP1 to the new library. Refer to MP1 on how to create a library.

Chapter 4: Getting Started

Creating a Copy of a Design:

To create a copy of MP1 first follow the directions in the MP2 handout to set the new MP2 library and the types file. After creating the MP2 library switch to the design browser window. You should see the MP2 library open in the left window of Design Browser. Next you will need open the MP1 library. If MP1 is already open, skip to the next step. Click on File and select open library, then MP1. Click and hold the right mouse button on CPU in the MP1 list in design browser. While holding the button down, drag CPU over to your newly created MP2 library and drop it on the line in MP2 that has an icon of an open book. Let go of the mouse button and window box should appear. Select the "Hierarchy Copy here" option and then make sure that Copy through Components, Copy to target library, and All are checked in. Click Ok and your entire design should be duplicated in MP2. Now you will need to set your package references. Double click with the left mouse button on CPU. Your datapath diagram will open up. Double click on Package list on your diagram(or click on edit then Package References) Add the correct packages that were used in MP1 except for the LC2_types file. For the LC2_types package, click on MP2 in the Library box and then select LC2_types and then add to list. Then make sure the inherit from parent box is checked in. Finally click OK.

Chapter 5: Design Limitations for Checkpoint 1

You Cannot:

Add additional ports in/out to the LC-2 register file
Add additional datapath registers
Make additional alu-type boxes (e.g. comparators)
Design non-“realistic” VHDL. For example, muxes should be nothing but a switch statement

You Can:

Make additional muxes with the following limitations:
1. The delay for a 4 input MUX is 2 * that of a 2 input MUX.
2. The delay for an 8 input MUX (including 5,6,and 7 input variations) is 4 * that of a 2 input MUX

Add logic gates with the following limitations:
1. Logic is limited to 4 inputs per gate
2. The delay for a 2-input gate is 1 ns.
3. The delay for 3 and 4 input gates is 2 ns.

Add additional port in/outs between cpu components (datapath and control).

Lengthen your clock to as much as 50 ns to accommodate your critical path. Make sure that the MRESP_H signal is held high for the length of your clock cycle. This signal is found in the Memory.vhdl file.

Sext has the delay of a two-input mux (sext should use the msb to select between 0's and 1's).

Zext has no delay (it's just a concatenation).

Concatenation requires no delay (bunching or splitting wires incurs no delay).

Delays need to be coded into your individual component files.

If you have questions about design limitations, please consult a TA.

Part II will have additional design constraints and will be released after Exam I.

Chapter 6: MP2 Checkpoint 1 Hand-in Requirements

Please turn in the following in this order:

1. All Datapath diagrams
2. Control Unit diagrams including all hierarchical diagrams
3. Simulation trace of the last 100 CYCLES, ending when the last instruction of the test code has finished execution. In the trace please include:

   CLK, RESET_L, START_H, ADDRESS, DATAIN, DATAOUT, Opcode, Out_ALU, Out_PC, MREAD_L, MRESP_H, RegWrite, Dest, and Reg(0) through Reg(7).
   All values should be displayed in HEX. Please highlight your final register values.

4. Memory Dump with all store instruction values.

Use the test code provided from the webpage (to be supplied closer to the due date).
You are strongly encouraged to write your own test code to test your implementation and not wait until we release ours.

To perform a memory dump, please do the following:

1) Enter ModelSim
2) Open the Lists Window

3) Drag signals MWRITE_L, DATAOUT, and ADDRESS into the Lists Window

Next, you need to set the triggering for each of the three signals. Select a signal in the list window and go to Props->Signal in the menu. The triggering option should be at the bottom right of the options window. Set the triggering as follows:

MWRITE_L - Triggered
DATAOUT - NOT Triggered
ADDRESS - NOT Triggered

Now just run the simulation and every memory write should be recorded in the lists window. Write the output to a *.lst file, print, and you're done.