Computer Structure Lab - Fall 1999

Dr. Guy Even

Handout #5: A Load/Store Machine

In this assignment you will practice designing memory accesses in the DLX design. To focus on memory accesses, we will consider a primitive application, called the Load/Store Machine. The Load/Store Machine executes DLX programs that consist only of simplified load and store instructions.

The only difference between the memory accesses of the Load/Store Machine and a word-addressable DLX is in the computation of the effective address (which is simpler in the Load/Store Machine).

1 Specification

The instruction set of the Load/Store Machine consists only of load and store instructions.

1.1 Modified Load/Store Instructions

Load/Store	Semantics
`lw    RD R0 imm`	RD := M(imm)
`sw    RD R0 imm`	M(imm) := RD

Note that we allow the source register to be only `R0`. This means that the computation of the effective address does not require adding the source register and the immediate constant.

1.2 I-Format

Load/store instructions are encoded in the I-Format. An instruction in the I-Type-Format is divided into four fields depicted below.

6	5	5	16
Opcode	***RS1***	***RD***	***immediate***

1.3 Encoding

The load instruction is encoded by IR$[31:26] = 100011$. The store instruction is encoded by IR$[31:26] = 101011$.

2 Memory Accesses

In the the Read and Write Machines a memory access was implemented using a state machine with 4 states. However, we would like to simplify the control of the DLX and allocate only one state for each of the actions: "fetch", "load", and "store". How do we do bridge the gap between 4 states for a memory access and a single state? The way this is done is by cascading state machines.

Consider the Load/Store Machine. It accesses the main memory during "fetch", "load", and "store" states. Communication between the Load/Store Machine and the I/O Control Logic is done via a state machine called the "Memory Access Control". The Memory Access Control resembles the Read and Write Machines. The connections between these three state machines is depicted in Figure 1. The state diagram of the Memory Access Control is depicted in Figure 2.

The control signals have the following meanings:

1. `mr` means memory read. It is active during the "fetch" and "load" states.
2. `mw` means memory write. It is active during the "store" state.
3. `req` means either `mr` or `mw` is active.
4. The other signals connect to the I/O control logic and are well understood by now.

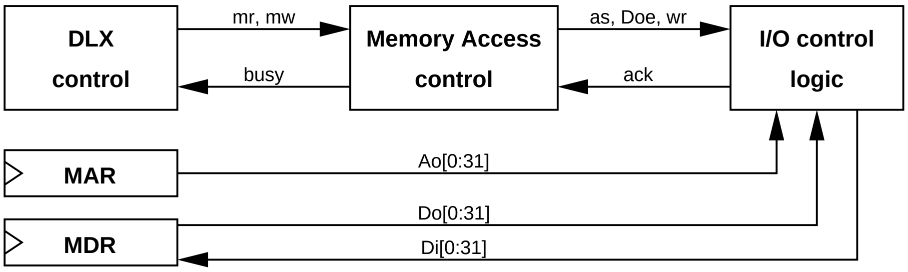

Figure 1: Memory accesses in the DLX machine

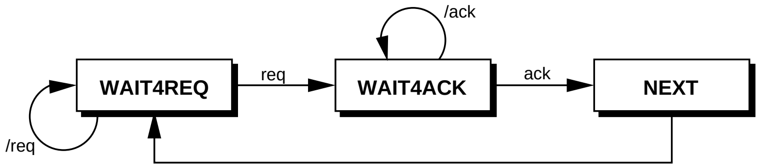

Figure 2: Finite state diagram of the memory access control.

3 Implementation

We outline the datapath and control of the Load/Store Machine.

3.1 Datapath

Figure 3 depicts a block diagram of the datapath of the Load/Store Machine. Busses are connected to the I/O control logic, as depicted in Fig. 1. Control signals are omitted from this figure, and you are asked to decide which signals are needed and when they are active.

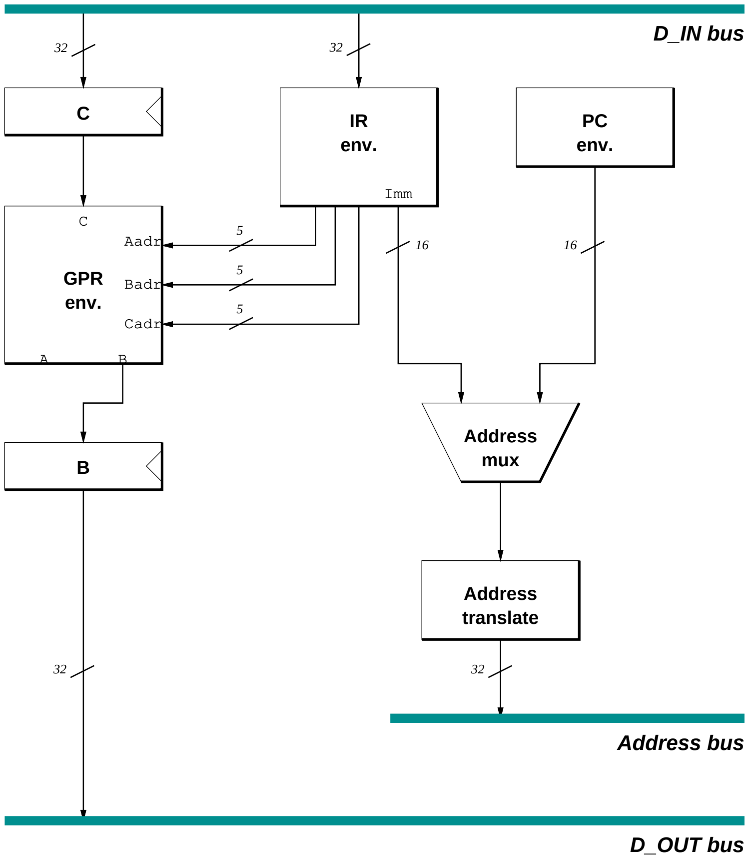

Figure 3: Block diagram of Load/Store Machine datapath.

3.2 Control

Figure 4 depicts a state diagram of the control of the Load/Store Machine. The RTL instruction in each state is specified in Table 1. A `reset` signal causes a transition to the INIT state.

4 The GPR environment

The GPR environment has the following inputs: `C`, `Aadr`, `Badr`, `Cadr`, `gpr_we`. The outputs are `A`, `B`. Note that the Load/Store Machine could do with one output, however, we specify two outputs so that this module can be used for the DLX design as well. It can support one of two operations in each cycle:

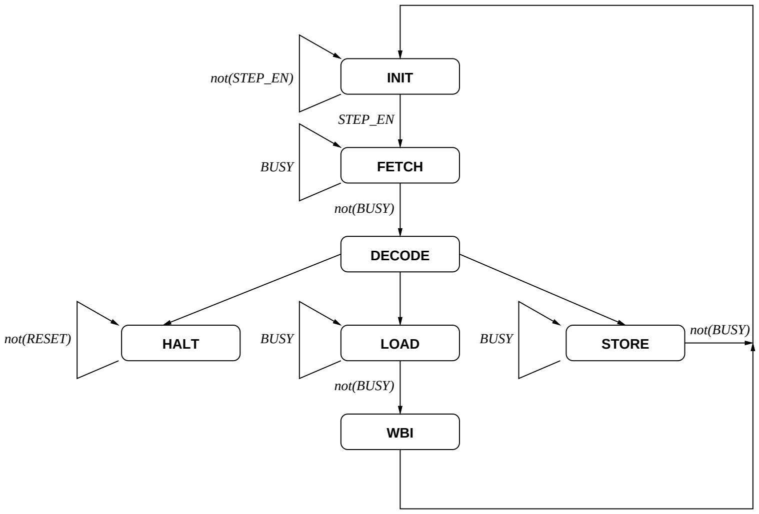

Figure 4: State diagram of the control of the Load/Store Machine. An active `reset` signal causes a transition to the INIT state.

State	RTL Instruction
INIT	*wait for step enable*
FETCH	$IR = M(PC)$
DECODE	$B = RD$, $PC = PC + 1$
STORE	$M(Imm) = B$
LOAD	$C = M(Imm)$
WBI	$RD = C$
HALT	*machine is stuck till reset*

Table 1: RTL instructions for the states of the control

1. Write the value of input $\mathtt{C}$ in register $\mathtt{R}[\mathtt{Cadr}]$ if $\mathtt{gpr\_we} = 1$.

2. Read the contents of the registers with indices `Aadr` and `Badr`. The outputs $\mathtt{A}$ and $\mathtt{B}$ are defined by:

$$\mathtt{A} = \begin{cases} \mathtt{R}[\mathtt{Aadr}] & \text{if } \mathtt{Aadr} \neq 0 \text{ and } \mathtt{gpr\_we} = 0 \\ 0 & \text{if } \mathtt{Aadr} = 0 \text{ and } \mathtt{gpr\_we} = 0 \\ \text{arbitrary} & \text{otherwise.} \end{cases}$$

and

$$\mathtt{B} = \begin{cases} \mathtt{R}[\mathtt{Badr}] & \text{if } \mathtt{Badr} \neq 0 \text{ and } \mathtt{gpr\_we} = 0 \\ 0 & \text{if } \mathtt{Badr} = 0 \text{ and } \mathtt{gpr\_we} = 0 \\ \text{arbitrary} & \text{otherwise.} \end{cases}$$

5 Pre-lab Assignment

1. (5 pts.) Address translation. The main memory address space to which the Load/Store Machine can both read and write is 0x00800000-0x008fffff.

   Suppose we are interested in giving the Load/Store Machine the illusion of a 16-bit main memory address space with the addresses 0x0000-0xffff. Show how the Address Translation Module can support this illusion. Refer to the PC register and the address used for memory accesses.

2. (5 pts.) Design the Memory Access Control module. Outline the differences between this module and the Read and Write Machines.

3. (25 pts.) Complete the the Load/Store Machine design. Your design should be as simple as possible. Do not try to make a design that will be easy to use as a basis for the DLX! That will complicate your design.

   (a) (15 pts.) Prepare designs of the blocks in the datapath (PC environment, GPR environment, IR environment, etc.). Each design should be organized as follows: list of inputs and outputs, definition of functionality (equations describing relations between outputs and inputs), and drawing (you may use counters, decoders, etc. as building blocks of your designs). Don't forget the control signals.

   (b) (10 pts.) Prepare a table listing *all* the control signals, their meanings, their equations , and the points the are connected to.

4. (5 pts.) Assume that due to the library, we always want to have 32 registers in the GPR, although we only really need 31 registers. There are two ways to implement the register `R0` in the GPR. In one way, write access to `R0` are disabled to keep it with a zero. In the second way, read accesses to `R0` are pulled down to output a zero. Compare these two methods and explain which method is cheaper/faster.

6 Lab Assignment

(60 pts.) Implement the Load/Store Machine with the monitor slave and the logic analyzer. Guidelines:

1. Name all signals and modules using only uppercase letters. Do not use non-letters in the names except for underscores.

2. When the Load/Store Machine fetches an instruction which is not a load or a store instruction it halts by entering a "halt" state.

3. Use DLX assembler to generate a short program. An example of a program appears in the appendix.

4. Submit printouts of the ABEL files, schematics, short Load/Store machine program, and the wave forms monitored when running a load instruction and a store instruction on the design. The waveforms should include the following signals and values: the GPR control signal, the memory access control inputs and outputs, control signals of the registers (A,B,C,PC,IR).

5. You may (but do not have to) generate in ABEL an output signal that shows the current state. This information can be then monitored to check if the control is functioning properly.

A The DLX Assembly Language

Programs for the DLX need not be written in machine code. To simplify the task of programming the DLX an assembly language was written as well as an assembler program (this is a program that translates assembly programs to machine code programs).

The assembler is called `DLXASS.exe`. We use the following naming conventions: The suffix of an assembly file is `.s` and the suffix of a machine code file is `.cod`. If requested, the assembler program also outputs a "dump" of the machine code file with the suffix `.lst`.

An example of an assembly program is depicted below:

```
* Example file for the load/store machine

          pc= 0x00000000        * Address of program in main memory

          lw R1 R0 data1        * R1=M(data1)=0x00000001
          lw R2 R0 data2        * R2=M(data2)=0x01234567
          lw R3 R0 data3        * R3=M(data3)=0xfedcba98
          lw R4 R0 data4        * R4=M(data4)=0xaffeaffe

          sw R1 R0 adr1         * M(adr1)=R1
          sw R2 R0 adr2         * M(adr2)=R2
          sw R3 R0 adr3         * M(adr3)=R3
          sw R4 R0 adr4         * M(adr4)=R4

          sw R4 R0 adr1         * M(adr1)=R4
          sw R3 R0 adr2         * M(adr2)=R3
          sw R2 R0 adr3         * M(adr3)=R2
          sw R1 R0 adr4         * M(adr4)=R1

          dc 0x00000000         * This is an illegal instruction!


* Here the data of this program starts

adr1:   ds 1                    * First memory address
adr2:   ds 1                    * Second memory address
adr3:   ds 1                    * Third memory address
adr4:   ds 1                    * Fourth memory address

data1:  dc 0x00000001           * First data value
data2:  dc 0x01234567           * Second data value
data3:  dc 0xfedcba98           * Third data value
```

```
data4:  dc 0xaffeaffe          * Fourth data value
```

To generate the "dump" file type: `dlxass -d ls_ex.lst ls_ex.s ls_ex.cod`. Below is an example of a dump file.

```
0x00000000: 0x8C010011             LW R1 R0 data1
0x00000001: 0x8C020012             LW R2 R0 data2
0x00000002: 0x8C030013             LW R3 R0 data3
0x00000003: 0x8C040014             LW R4 R0 data4
0x00000004: 0xAC01000D             SW R1 R0 adr1
0x00000005: 0xAC02000E             SW R2 R0 adr2
0x00000006: 0xAC03000F             SW R3 R0 adr3
0x00000007: 0xAC040010             SW R4 R0 adr4
0x00000008: 0xAC04000D             SW R4 R0 adr1
0x00000009: 0xAC03000E             SW R3 R0 adr2
0x0000000A: 0xAC02000F             SW R2 R0 adr3
0x0000000B: 0xAC010010             SW R1 R0 adr4
0x0000000C: 0x00000000            DC 0x00000000

0x0000000D:                 adr1: DS 0x00000001
0x0000000E:                 adr2: DS 0x00000001
0x0000000F:                 adr3: DS 0x00000001
0x00000010:                 adr4: DS 0x00000001
0x00000011: 0x00000001     data1: DC 0x00000001
0x00000012: 0x01234567     data2: DC 0x01234567
0x00000013: 0xFEDCBA98     data3: DC 0xFEDCBA98
0x00000014: 0xAFFEAFFE     data4: DC 0xAFFEAFFE
```

The assembly language has the following rules:

1. Everything that appears in a line after a "*" is a comment.
2. The line `pc= 0x12345` means that next translated line will be mapped to address `0x12345`.
3. A label is defined by placing its name at the beginning of a line followed by a ":". (in the example: `adr1:`). 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) is replaced with the address that corresponds to the label.
4. `dc` x means "place the constant x in the address that corresponds to the current line".
5. `ds` y means "reserve y words in memory". The address of the next line is y plus the address of the current line.
6. Instructions are given the mnemonics used in the description of the instruction sets.