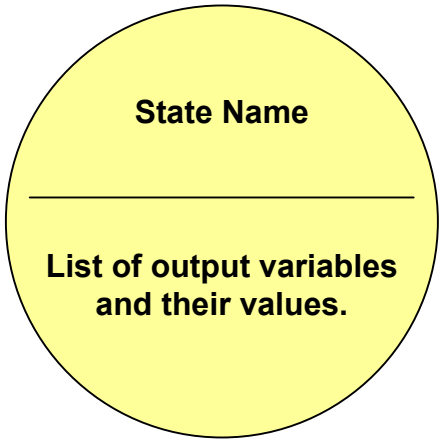
State Machine Implementation of the ARC CPU

Introduction

The textbook presents two approaches to the design of the control unit (CU) of the ARC CPU, "microprogramming," and "hardwired control." The textbook's approach to developing a hardwired CU is based on use of a hardware description language (HDL) called VHDL. This document shows an alternative approach based on FSM design covered earlier in the course.

Background

In the Moore model of a FSM, the outputs of the state machine are totally determined by the current state of the machine. This design sometimes requires more states to accomplish its task than a Mealy machine in which the present state and the present input values combine to determine the output values, but Moore machines are somewhat simpler to work with conceptually. In a Moore a circle containing the name of the state and the output values during that state, separated by a horizontal line, represents machine a state:



For the ARC datapath, there are 24 bits that the CU must generate values for on each clock pulse: six bits that go into the A-Bus decoder, six for the B-Bus decoder, six for the C-Bus decoder, four for the ALU function code, and two, Read (RD) and Write (WR), to control the memory. Rather than list all 22 bits and their values below the horizontal line in each state, we can use RTL statements to indicate how these 24 bits are to be set. We will use two forms of RTL statement, one for states where a memory operation takes place, and another for states when ALU operations are performed. To keep things as simple as possible, we will assume that all states fall neatly into exactly one of these two categories.

RTL for Memory Operations

A *read* operation takes the form Data ← Memory[Address] and a *write* operation takes the form Memory[Address] ← Data. In both cases, the Address is the contents of a register connected to the A Bus. For a read operation the Data is loaded into a

register from the C Bus, whereas for a write operation the Data comes from a register connected to the B Bus.

For memory read operations, the B Bus is ignored by the memory and may contain arbitrary data. We will connect Register number zero (%r0), the pseudo-register that can't be changed and always provides 32-bits of zeros, for all read operations. The output of the ALU is discarded for read operations because the C Bus gets its input from memory. So the ALU function code doesn't matter, except that the condition code bits must not be modified when reading from memory. We will use the ALU's ADD function (0101_2) for the function code during all read (and write) operations.

For memory write operations, the output of the ALU could in fact be loaded into one of the registers, but we will always load it into %r0, thus discarding it.

Here are some examples of RTL statements for memory operations, and how the 24 bits output by the CU would be set for each one:

RTL	A Bus	B Bus	C Bus	ALU	RD	WR
R[ir] ← Memory[R[pc]]	100000	000000	100101	0101	1	0
Memory[R[temp0]] ← R[$ir_{25:29}$]	100001	0, $ir_{25:29}$	000000	0101	0	1

In this table, the A Bus, B Bus, and C Bus columns give the six bit number the CU sends to the corresponding decoders. The busses actually are connected to the 32-bit contents of the correspondingly numbered registers. The subscript "25:29" means bit numbers 25 through 29, which is the RD field of the instruction register.

RTL for ALU Operations

We will use the following notation for ALU operations:

C-Bus ← ALU_op(A-Bus, B-Bus)

Here, ALU_op will be the name of one of the sixteen ALU operations listed in Figure 6-4 on page 194. For those operations that ignore the B-Bus, we will set the B Bus bits to all zeros. Here are some examples:

RTL	A Bus	B Bus	C Bus	ALU	RD	WR
R[temp0] ← Sext13(R[ir])	100101	000000	100001	1100	0	0
R[rd] ← Addcc(R[rs1], R[rs2])	0,$ir_{14:18}$	0, $ir_{0:4}$	0,$ir_{25:29}$	0011	0	0

Decoders

We will use three decoders connected to the bits of the Instruction Register to help with the design of the CU FSM.

- The *Instruction Format Decoder* decodes $IR_{30:31}$ to generate signals named *set-br_op* (00), *call_op* (01), *alu_op* (10), and *mem_op* (11).
- The *Op2 Decoder* decodes $IR_{22:24}$ to produce 8 signals. Two of these eight outputs are named *branch* (010), and *sethi* (100).

- The *Op3 Decoder* decodes $IR_{19:24}$ to produce 64 signals. Some of these are named *ld* (000000), *st* (000100), *addcc* (010000), etc. See Figure 6-2 on page 192 for the complete list of Op2 and Op3 outputs.

There would be a fourth decoder for $IR_{25:28}$ used for branch instructions.

The hardwired control unit presented in the textbook uses "one hot" encoding for the state machine in which there is a separate state flip-flop for every state, with exactly one flip-flop true at all times. We can accomplish the same thing by decoding all the state flip-flops to produce a number of signals that are all false except for one corresponding to the current state. In either event, we will have wires named *State_0*, *State_1*, etc. for all the possible states of the CU.

State Table

It would be impossible to develop a full state table for the CU FSM because with 40 bits of input from the datapath (32 bits from the Instruction Register plus four bits from the Condition Code) there would be 2^{40} rows. But by using the decoders listed above as shorthand notation for the inputs to the CU and RTL statements as shorthand for the outputs, we can show the some of the State Table as follows:

External Inputs **(40 bits from datapath)**	**Present State**	**Next State** **(also, flip-flop inputs)**	**Outputs** **(22 bits to datapath plus 2 bits to memory)**
Mem_op	State_0 (Instruction Fetch)	State_1	R[ir] ← Memory[R[pc]]
IR_{13}	State_1	State_2	R[temp0] ← Sext13(R[ir])
~IR_{13}	State_1	State_3	R[temp0] ← Sext13(R[ir])
	State_2	State_4	R[temp[0] ← Add(R[$ir_{14:18}$],R[temp0])
	State_3	State_2	R[temp0] ← Add(R[$ir_{0:4}$], R[r0])
ld	State_4 (Compute EA)	State_5	R[temp0] ← Add(R[$ir_{14:18}$], R[temp0])
st	State_4	State_6	R[temp0] ← Add(R[$ir_{14:18}$], R[temp0])
	State_5 (Execute *ld*)	State_99	R[$ir_{25:29}$] ← Memory[R[temp0]]
	State_6 (Execute *st*)	State_99	Memory[R[temp0]] ← R[$ir_{25:29}$]
	State_99	State_0	R[pc] ← Incpc(R[pc])

State Diagram

Finally, here is a State Diagram for the above portion of the State Table, with some indications of how this part of the diagram would relate to other parts, not shown here. To make the diagram more manageable, informal and simplified versions of the RTL statements are used to indicate the outputs.