9.5 Translating a microprogram to hardware

(Original section[1])

To translate a microprogram into actual hardware, we need to specify how each field translates into control signals. We can implement a microprogram with either finite-state control or a microcode implementation with an explicit sequencer. If we choose a finite-state machine, we need to construct the next-state function from the microprogram. Once this function is known, we can map a set of truth table entries for the next-state outputs. In this section, we will show how to translate the microprogram, assuming that the next state is specified by a sequencer. From the truth tables we will construct, it would be straightforward to build the next-state function for a finite-state machine.

Assuming an explicit sequencer, we need to do two additional tasks to translate the microprogram: assign addresses to the microinstructions and fill in the contents of the dispatch ROMs. This process is essentially the same as the process of translating an assembly language program into machine instructions: the fields of the assembly language or microprogram instruction are translated, and labels on the instructions must be resolved to addresses.

The figure below shows the various values for each microinstruction field that controls the datapath and how these fields are encoded as control signals. If the field corresponding to a signal that affects a unit with state (i.e., Memory, Memory register, ALU destination, or PCWriteControl) is blank, then no control signal should be active. If a field corresponding to a multiplexor control signal or the ALU operation control (i.e., ALUOp, SRC1, or SRC2) is blank, the output is unused, so the associated signals may be set as don't care.

Figure 9.5.1: Each microcode field translates to a set of control signals to be set (COD Figure C.5.1).

These 22 different values of the fields specify all the required combinations of the 18 control lines. Control lines that are not set, which correspond to actions, are 0 by default. Multiplexor control lines are set to 0 if the output matters. If a multiplexor control line is not explicitly set, its output is a don't care and is not used.

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, lorD = 0, IRWrite	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, lorD = 1	Read memory using ALUOut as address; write result into MDR.
	Write ALU	MemWrite, lorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00, PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	Jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

The sequencing field can have four values: Fetch (meaning go to the Fetch state), Dispatch 1, Dispatch 2, and Seq. These four values are encoded to set the 2-bit address control just as they were in COD Figure C.4.4 (The values of the address-control lines are set in the control word that corresponds to each state): Fetch = 0, Dispatch 1 = 1, Dispatch 2 = 2, Seq = 3. Finally, we need to specify the contents of the dispatch tables to relate the dispatch entries of the sequence field to the symbolic labels in the microprogram. We use the same dispatch tables as we did earlier in COD Figure C.4.3 (The dispatch ROMs each have $2^6 = 64$ entries that are 4 bits wide ...).

A microcode assembler would use the encoding of the sequencing field, the contents of the symbolic dispatch tables in the figure below, the specification in the figure above, and the actual microprogram to generate the microinstructions.

Figure 9.5.2: The two microcode dispatch ROMs showing the contents in symbolic form and using the labels in the microprogram (COD Figure C.5.2).

dispatch table 1		
Opcode field	Opcode name	Value
000000	R-format	Rformat1
000010	`jmp`	JUMP1
000100	`beq`	BEQ1
100011	`lw`	Mem1
101011	`sw`	Mem1

Microcode dispatch table 2		
Opcode field	Opcode name	Value
100011	`lw`	LW2
101011	`sw`	SW2

Since the microprogram is an abstract representation of the control, there is a great deal of flexibility in how the microprogram is translated. For example, the address assigned to many of the microinstructions can be chosen arbitrarily; the only restrictions are those imposed by the fact that certain microinstructions must occur in sequential order (so that incrementing the State register generates the address of the next instruction). Thus the microcode assembler may reduce the complexity of the control by assigning the microinstructions cleverly.

Organizing the control to reduce the logic

For a machine with complex control, there may be a great deal of logic in the control unit. The control ROM or PLA may be very costly. Although our simple implementation had only an 18-bit microinstruction (assuming an explicit sequencer), there have been machines with microinstructions that are hundreds of bits wide. Clearly, a designer would like to reduce the number of microinstructions and the width.

The ideal approach to reducing control store is to first write the complete microprogram in a symbolic notation and then measure how control lines are set in each microinstruction. By taking measurements we are able to recognize control bits that can be encoded into a smaller field. For example, if no more than one of eight lines is set simultaneously in the same microinstruction, then this subset of control lines can be encoded into a 3-bit field ($\log_2 8 = 3$). This change saves five bits in every microinstruction and does not hurt CPI, though it does mean the extra hardware cost of a 3-to-8 decoder needed to generate the eight control lines when they are required at the datapath. It may also have some small clock cycle impact, since the decoder is in the signal path. However, shaving five bits off control store width will usually overcome the cost of the decoder, and the cycle time impact will probably be small or nonexistent. For example, this technique can be applied to bits 13—6 of the microinstructions in this machine, since only one of the seven bits of the control word is ever active (see COD Figure C.4.5 (The contents of the control memory for an implementation using an explicit counter)).

This technique of reducing field width is called *encoding*. To further save space, control lines may be encoded together if they are only occasionally set in the same microinstruction; two microinstructions instead of one are then required when both must be set. As long as this doesn't happen in critical routines, the narrower microinstruction may justify a few extra words of control store.

Microinstructions can be made narrower still if they are broken into different formats and given an opcode or *format field* to distinguish them. The format field gives all the unspecified control lines their default values, so as not to change anything else in the machine, and is similar to the opcode of an instruction in a more powerful instruction set. For example, we could use a different format for microinstructions that did memory accesses from those that did register-register ALU operations, taking advantage of the fact that the memory access control lines are not needed in microinstructions controlling ALU operations.

Reducing hardware costs by using format fields usually has an additional performance cost beyond the requirement for more decoders. A microprogram using a single microinstruction format can specify any combination of operations in a datapath and can take fewer clock cycles than a microprogram made up of restricted microinstructions that cannot perform any combination of operations in a single microinstruction. However, if the full capability of the wider microprogram word is not heavily used, then much of the control store will be wasted, and the machine could be made smaller and faster by restricting the microinstruction capability.

The narrow, but usually longer, approach is often called *vertical microcode*, while the wide but short approach is called *horizontal microcode*. It should be noted that the terms "vertical microcode" and "horizontal microcode" have no universal definition-the designers of the 8086 considered its 21-bit microinstruction to be more horizontal than in other single-chip computers of the time. The related terms *maximally encoded* and *minimally encoded* are probably better than vertical and horizontal.

(*1) This section is in original form.