Control Implementations

Control unit design:

- **Hardwired control**
  - expressed as a finite state machine (state diagrams & transitions between states)
  - good if the number is states is “small” (RISC architectures)
  - implemented with PLAs (programmable logic arrays) & random logic
- **Microprogrammed control**
  - expressed as a “micro” program
  - provides structure & modularity when the number of states is “large” (CISC architectures)
  - implemented with read-only memory (ROM) of simple instructions (**microcode** instructions) to drive the signals needed to execute instructions in the ISA

Finite State Machine

Sequential logic function that consists of:

- set of **inputs**
- **next state function**: maps current state & inputs to a new state
- **output function**: maps current state (Moore machine) or current state & inputs (Mealy machine) to a set of outputs

Finite state machine for control:

- state = a step in the execution cycle
- input = opcode & func fields
- output = control signals that drive the datapath

Microprogramming -- Basic Idea

Microprogramming: designing control that implements machine (architecture) instructions in terms of simpler microinstructions

- **microinstruction**: specifies the control signals that must be asserted in a given cycle
- fields in a microinstruction are represented symbolically (just like assembly language for machine instructions)
- each machine instruction is "interpreted" by a sequence of microinstructions
- **microcode** or **microprogram**: set of all microinstructions that control processor execution

Implemented by a simple auxiliary datapath & control unit that generates signals for the main datapath

- "computer within a computer"
- micro datapath fetches microinstructions
- micro control unit sends signals to the main datapath

Microinstruction Encoding

Multiple fields, multiple values per field

- width of field is determined by the number of values

Encoding the microinstruction:

- design so that each field specifies a non-overlapping set of control signals
  - signals that are not asserted together can share the same field
- don't put so many signals into the same field that it needs complex interpretation to get the individual signal values that will drive the macro datapath
- 2 different style extremes
  - **horizontal microcode**: 1 field for each value
    - no decoding
    - very wide microinstructions
  - **vertical microcode**:
    - very highly encoded
    - much narrower microinstructions
- a control signal can't be set to more than one value in a microinstruction
  - microassembler makes sure that conflicting signals aren't generated in a microinstruction

Microsequencing

Choosing the next microinstruction

(1) the next sequential microinstruction

- increment the microPC

(2) begin a new machine instruction

- branch to the microinstruction that controls fetching

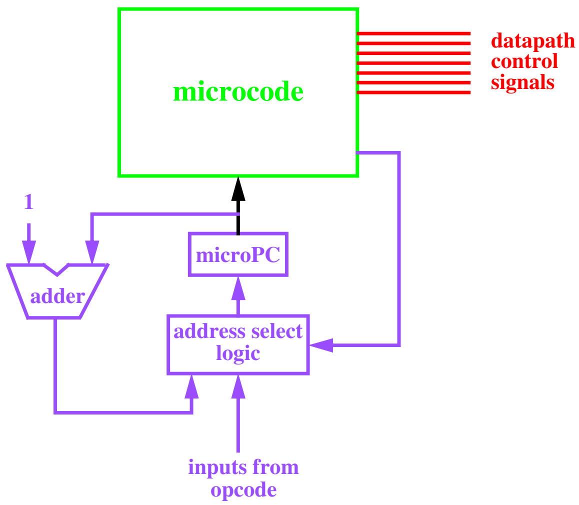
(3) have multiple options that depend on some control unit input

- dispatch through a table of microinstruction target addresses
- R2000 uses this for switching on the opcode

Microcode for R2000

Label	ALU Control	SRC 1	SRC 2	Register control	Memory	PCWrite control	Sequencing
fetch	add	PC	4		read PC	ALU	seq
	add	PC	extshft	read			dispatch 1
mem1	add	A	extend				dispatch 2
lw2					read ALU		seq
				write MDR			fetch
sw2					write ALU		fetch
rfmt1	func code	A	B				seq
				write ALU			fetch
beq1	subt	A	B			ALU-Out: cond	fetch
jump1						jump address	fetch

Block Diagram



Pentium Pro

Implementation

- PLA (hardwired) for RISC-like instructions
- microcode for more complex instructions
  - ~8000 microinstruction ROM
  - microsubroutines (nanocode)

Execution times are similar to RISC machines

- most instructions are RISC-like

A Comparison

Distinctions used to be clearer

- microinstructions in ROM faster than machine instructions in RAM (memory)
  - an argument for CISC also
- microcode could be expressed symbolically
  - microcode was easier to express -- hardwired random logic was too complicated to specify for a CISC architecture
  - microcode could be easily changed
    - $\Rightarrow$ new instructions could be easily added
    - $\Rightarrow$ bugs could be easily fixed
    - $\Rightarrow$ specifying the architecture & building the implementation could go on in parallel
- microcode was more modular
  - different ROMs could be used to emulate older architectures
  - microsubroutines could be used

A Comparison

Both have similar performance

- ROM no longer has the big speed advantage over RAM since instructions are cached
- PLA may be smaller & therefore somewhat faster than a ROM

Both have the same difficulty of design & debugging

- CAD tools allow hardwired control to be specified symbolically
  - same difficulty in specifying & debugging control
  - same difficulty in adding instructions to an existing ISA
- faster machines allow good simulation & therefore fewer bugs

Could probably use multiple PLAs as easily as multiple ROMs to allow several implementations of the same architecture on the same machine
(good for backwards compatibility)

Still the case that:

- hardwired control used for simple, regular instructions
- microcode used for complex, variable-length instructions