Microprogramming

Tom Kelliher, CS 240

Apr. 2, 2004

1 Administrivia

Announcements

Collect homework.

Assignment

Read 5.6, 6.1.

From Last Time

Multicycle implementation control.

Outline

1. Introduction.
2. Implementation.
3. History.

Coming Up

Exceptions (briefly), pipelining.

2 Microprogramming

First the what, then the why.

2.1 Introduction to Microprogramming

1. Data path (defn):
   (a) Register file.
   (b) ALU.
   (c) MDR, other "data" registers.
   (d) Memory.
2. Control unit (defn):
   (a) IR, PC.
   (b) Instruction decoder/encoder and/or state machine.
3. Data path needs sequences of 0's and 1's on control inputs to execute instructions.
4. Control unit provides the sequence.
5. Can the control unit be replaced with a memory (control store) whose output is connected to the data path's control inputs?
6. Contents of the control store — a program for each instruction. *Microprogram*. *Microinstructions*.

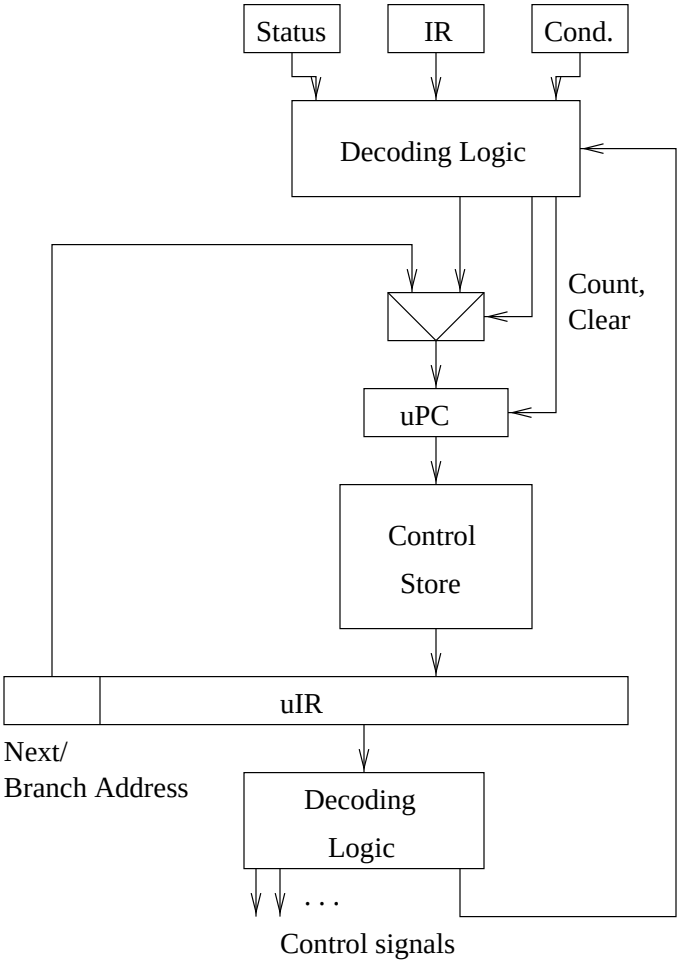
7. How do we *sequence* the control store? Required operations:

   (a) Straight line execution.

   (b) Unconditional branches.

   (c) Conditional branches.

A microsequencer:



1. Limiting the size of the control store:

   (a) Commonalities between microroutines.

   (b) Utilize branching to "factor out" common code.

(c) Micro-subroutines!!!

2.2 What a Long, Strange Trip It's Been

2.2.1 The Case for Microprogramming

Advantages off the bat:

1. Easier debugging.
2. Quicker to market.
3. Emulation.
4. Extending the instruction set.
5. Easier upgrades.

Disadvantages off the bat:

1. Slower than hard-wired.

1970s technology:

1. Main memory was core; control stores were solid state (10 times faster).
2. No caches.
3. 8Kb ROM = 8 bit register, space-wise.

Implications:

1. Program speed was proportional to program size (bandwidth).
2. Control stores were "cheap."

Solution: Microprogramming and richer instruction sets

1. Simplify compiler construction.
2. Close the "semantic gap."
3. Improve architectural quality by decreasing program size and bandwidth.
4. Microinstructions were "faster" than regular instructions.
5. Register-based architectures were unwieldy; use stack-based or memory-memory.

1980s technology:

1. Main memory was now solid state.
2. Caches were common.
3. CMOS VLSI.
4. Control store ROMs were becoming RAMs (bugs).
5. Compilers were sub-setting architectures.

Some weird developments:

1. Writable control stores.
2. Virtual memory at the control store level.
3. Nanocode.

Two CPUs:

	VAX-11/780	MIPS I
Year	1978	1982
Number of instructions	303	39
Control memory size	480Kb	0
Instructions sizes (bits)	16–456	32
Technology	TTL MSI	NMOS VLSI
Execution model	reg-mem mem-mem reg-reg	reg-reg
Cache size	64Kb	0

RISC design philosophy:

1. Functions should be kept simple unless there is very good reason to do otherwise.
2. Microinstructions should not be faster than simple instructions.
3. Microcode is not magic.
4. Simple decoding and pipelined execution are more important than program size.

RISC CPU traits:

1. Load/store; operations are register-register.
2. The operations and addressing modes are reduced.
3. Instruction formats are simple and do not cross word boundaries.
4. RISC branches avoid pipeline penalties.