CPS311 Lecture: CPU Control: Hardwired control and Microprogrammed Control

<p align="right">Last revised June 2, 2017</p>

Objectives:

1. To explain the concept of a control word
2. To show how control words can be generated using hardwired control
3. To explain the concept of microprogramming
4. To discuss the use of a RISC core in a CISC processor

Materials:

1. Projectable Version of Diagrams
2. MIPS Simulation
3. Code for Lab 5 - part 1 to demonstrate using microprogramming

I. Introduction

A. We have seen that a CPU - whether simple or complex - basically consists of a control unit, plus a data part, encompassing:

   1. A set of registers (including registers that interface to the system bus)

   2. A set of D-units (adders, shifters etc.)

   3. A set of data paths connecting the above.

   4. An interface to the "outside world" (memory, IO) - usually some sort of bus system.
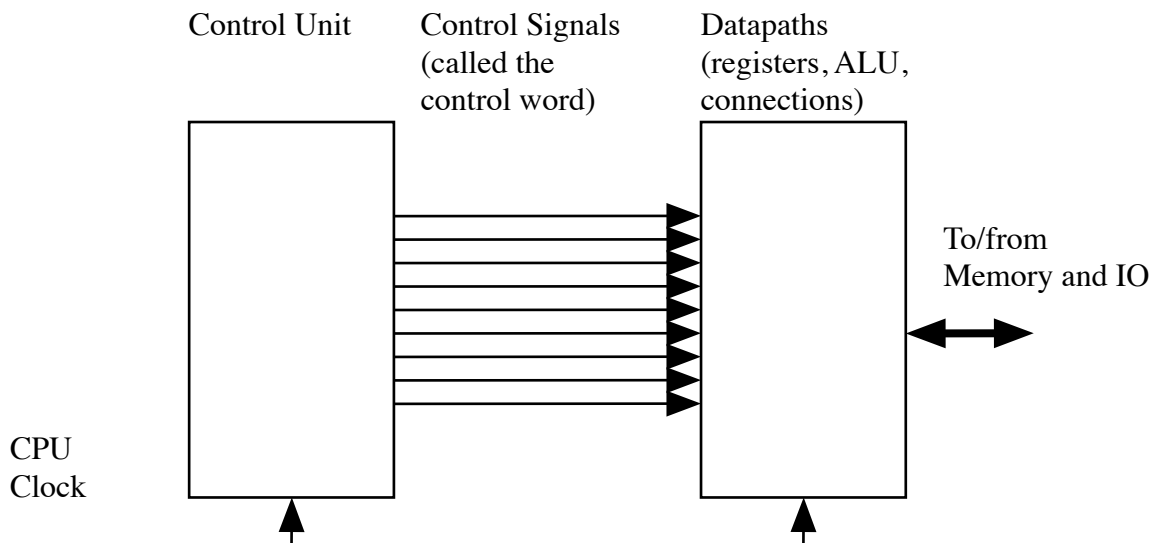
      (We continue to assume a single instance of all components shared by all steps of instruction execution, and sequential execution of instruction steps. When we discuss pipelining and other forms of parallelism, we will see that some components will have to be replicated.)

B. The data part is capable of performing a set of microoperations or primitive computations that can be performed in one cycle (clock pulse).

Each microoperation changes the contents of a single register. An instruction in the user-visible instruction set must be programmed as a series of microoperations (some of which may be done in parallel on the same clock pulse.)

C. Control of the system is accomplished by a control unit that - at the start of each clock cycle - activates the necessary control functions to cause the data part to perform the desired microoperation(s) on the next clock pulse. In the case of a multi-cycle CPU implementation (where a given component may perform different tasks on different cycles), this can be pictured as follows:

PROJECT

Control Unit   Control Signals   Datapaths
                  (called the   (registers, ALU,
                  control word)   connections)

To/from
Memory and IO

CPU
Clock

D. The set of control signals that pass from Control to the data part and bus system is called a micro-word or control word. Conceptually, each bit of this micro-word corresponds to the enabling of one particular microoperation that some system component can perform. (In practice, sometimes groups of bits are used to select from a set of mutually-exclusive options - e.g. the selection inputs to a MUX)

E. The job of the control unit designer for such a CPU is to develop a means whereby an orderly sequence of control words may be presented to the data part (and other hardware such as the memory) - one per clock pulse.

F. Historically, there have been two basic ways such a sequence of control words has been generated:

1. Hardwired control: The control unit is implemented as a state machine, with combinatorial circuits generating each of the control functions on the basis of the current state and certain variables such as the op-code of the user instruction undergoing execution.

2. Microprogrammed control. The various control words needed to implement the user instructions are stored in a ROM, with a sequencer causing the appropriate control word to be fetched at each clock cycle and fed to the rest of the CPU.

V. **An Example of Hardwired Control**

A. To get some feel for what is involved in hardwired control, we will discuss a hardwired control unit for our multicycle MIPS simulation.

B. Observe that, in the RTL specification for this machine we discussed earlier, almost all instructions require exactly 4 cycles to fetch and execute. (Some - the branches and jumps - require only two - one for fetch and one for execute - but we'll use 4 for all instructions for simplicity - thus wasting two on these).

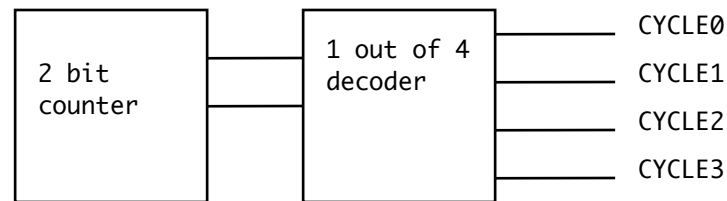1. Our state machine then looks like this:

PROJECT
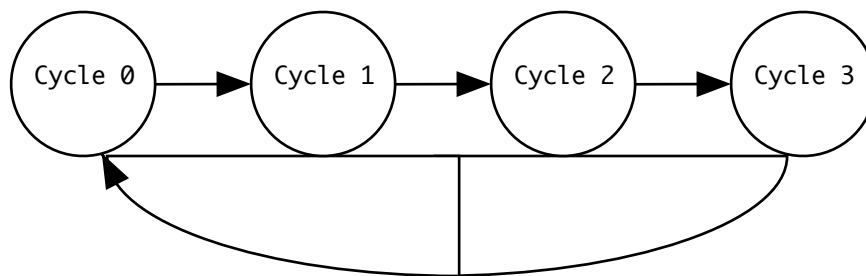
a) The simplicity of the state machine for MIPS is a consequence of the regularity of the instructions, which in turn is a

characteristic of the ISA designed to facilitate a pipelined implementation. (The ISA makes this part of the implementation easy)

b) Actually, a full implementation would need additional states to deal with issues like interrupts and exceptions.

c) This simplified state machine can be realized by a 2 bit counter, with its output decoded to yield 4 signals used internally in the control unit.

PROJECT

```
 ┌───────────┐        ┌───────────┐              CYCLE0
 │           │        │ 1 out of 4│────────
 │  2 bit    │────────│  decoder  │              CYCLE1
 │  counter  │────────│           │────────
 │           │        │           │────────      CYCLE2
 │           │        │           │
 └───────────┘        └───────────┘────────      CYCLE3
```

2. A CISC would require a much more complex state machines.

C. We will now explore the MIPS simulation in more detail. The control word for this simulated implementation contains 17 bits.

1. The meaning of the bits is as follows

(Project simulation, set to Manual control)

a) Some enable the loading of various registers (IR, PC, General) (Note that the ALU Input and Output registers are loaded on every cycle - there is nothing to be gained by having enables for them, since they can just be ignored when they're not needed.)

b) Some control the various MUXes. These may be single bits (for a 2-way MUX) or groups of bits - PC Source (2), Memory Address, Register Source, ALU Source A, ALU Source B (2).

c) One group of 2 controls _how_ the general register to be loaded (if there is one) is specified - i.e. a MUX that controls the input to the decoder that load-enables the correct register.

d) One group of 3 controls the ALU Function (i.e. the internal MUX in the ALU)

e) There is one bit each to control memory read and memory write.

2. Each of these bits can be derived by a combinatorial network whose inputs are the current state of the machine plus certain fields in the IR. It will simplify the design work if we assume that the opcode bits in the IR are connected to a 1 out of 64 decoder, with exactly one line being asserted for any given instruction (or none if the instruction is undefined)

PROJECT

```
                        ┌──────────────┐ ───────── RTYPE(0)
                        │ 1 out of 64  │          ...
              ──────────│ decoder      │ ───────── J (2)
  op code     ──────────│              │ ───────── JAL (3)
  portion     ──────────│              │ ───────── BEQ (4)
  of IR       ──────────│              │ ───────── BNE (5)
  (6 bits)    ──────────│              │   ...      ...
              ──────────│              │ ───────── ADDI (8)
              ──────────│              │ ───────── ADDIU (9)
              ──────────│              │ ───────── SLTI (0xa)
                        │              │ ───────── SLTIU (0xb)
                        │              │ ───────── ANDI (0xc)
                        │              │ ───────── ORI (0xd)
                        │              │ ───────── XORI (0xe)
                        │              │ ───────── LUI (0xf)
                        │              │   ...      ...
                        │              │ ───────── LW (0x23)
                        │              │   ...      ...
                        │              │ ───────── SW (0x2b)
                        └──────────────┘            ...
```
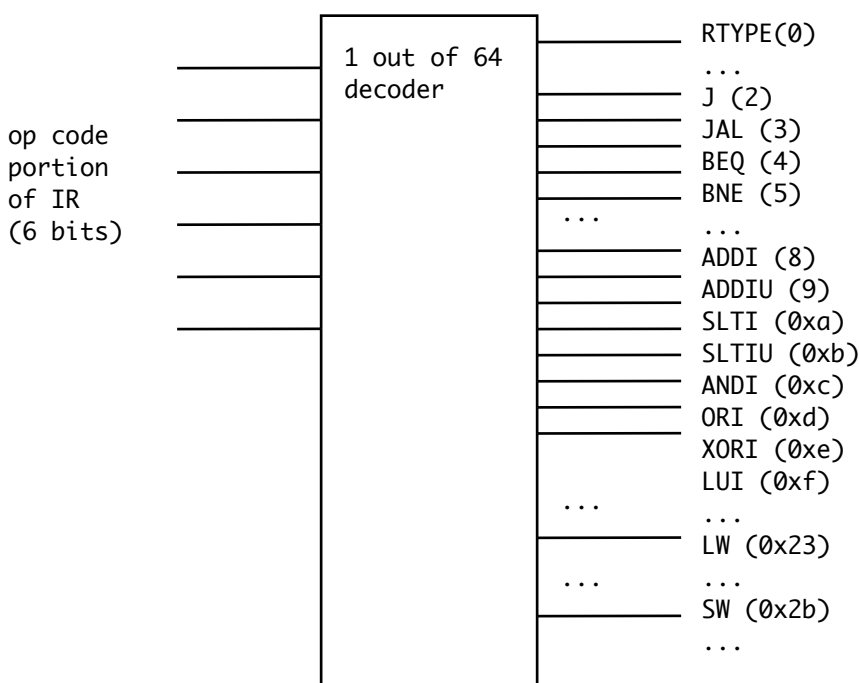
(A full implementation of the ISA would have many more!)

3. The function to be realized by each network is determined by examining the RTL to see what value of the bit is implied by   each.

a) Example: the Load IR bit. This is 1 on Cycle 0 of all instructions, and 0 everywhere else. Thus, we can derive this bit as

`CYCLE0` ——————— `LOADIR`

b) Example: the Load PC bit. This is 1 in four places, and 0 everywhere else
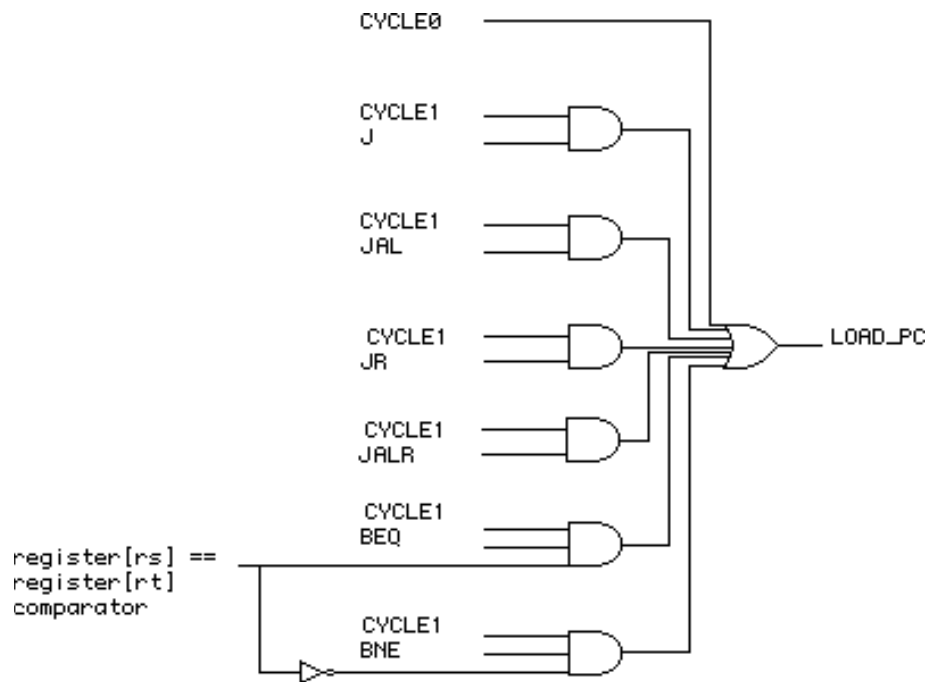
  (1) Cycle 0 of all instructions

  (2) Cycle 1 of `j`, `jal`

  (3) Cycle 1 of `jr`

  (4) Cycle 1 of `beq/bne` if and only if the branch condition is met
  Thus, we can derive this bit as:

  PROJECT

c) This same process can be continued for each bit of the control word. To simplify design, we can take advantage of don't-cares. Example: if `LOAD_PC` is 0, then we don't care about the value of `PC_SOURCE` It turns out we can make this 0 (PC + 4) on Cycle

6

0, 1 (IR J-Format constant) on Cycle 1, 3 (ALU Out) on Cycle 3, and we don't care what value it has on Cycle 2, since this yields the correct value whenever `LOAD_PC` is 1 and is ignored otherwise

d) etc.

4. When we looked at the execution of the program for Lab 5 Part I in the last lecture, we were actually looking at the control words generated by a simulated hard-wired control unit.
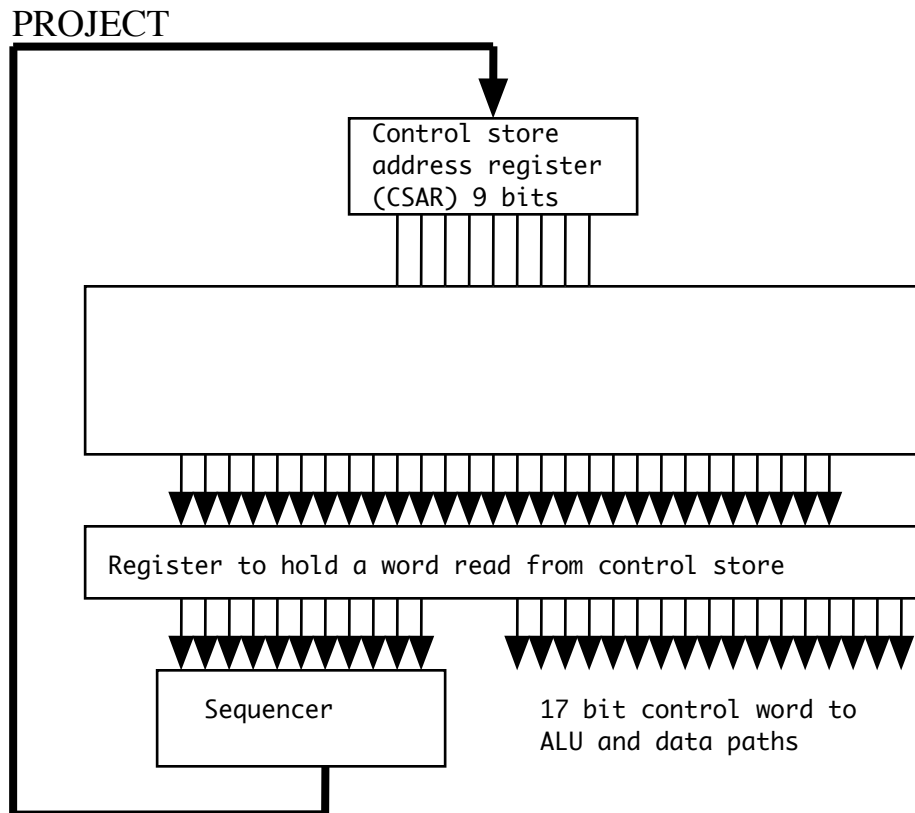
VI.**Microprogramming**-

A. As you can see, for even a very simple machine like the one we just looked at, hardwired control leads to very complex control logic. For a more complex machine, the control-unit complexity would make hardwired control virtually impossible. Thus, as computers became more complex, they began to use microprogramming as a means of keeping the complexity of control within limits (at the cost of a somewhat slower execution cycle.)

B. The basic idea is this: we build the control unit around a small, very fast memory (not visible to the programmer.)

   1. The width of this memory is equal to the width of the control word, plus some additional bits we will discuss shortly.

   2. We store the various control words in the memory (which is therefore called the CONTROL STORE). We connect the output of the memory to the control inputs of the ALU, data paths, etc.

   3. On each clock, we fetch a control word from control store and use it to determine what the ALU etc. do on that clock.

   4. We use a simple device called the SEQUENCER to arrange for the correct sequence of control words to be fetched. (The additional bits in each word in control store are used to control the sequencer.)

   5. The control store is generally a ROM; but it is also possible to use a writeable memory (PROM or RAM) for the control memory. This allows for:

      a) Dynamic microprogramming - e.g. for adding custom user instructions to the standard set or emulating another machine.

b) Diagnostics - a microprogram that exercises a suspected portion of the circuitry one micro-operation at a time may be loaded to assist in the isolation of hardware flaws.

C. A micro-programmed implementation of our example MIPS machine. (Note: this is strictly hypothetical to illustrate how it could be done. Actual MIPS implementations do not use microprogramming)

1. Structure of the control unit. (All micro-programmed CPU's use a structure like this, but of course the specific sizes will vary)

PROJECT

```
                    ┌──────────────────────┐
                    │ Control store        │
                    │ address register     │
                    │ (CSAR) 9 bits        │
                    └──────────────────────┘

        ┌──────────────────────────────────────────────┐
        │                                              │
        └──────────────────────────────────────────────┘

        ┌──────────────────────────────────────────────┐
        │ Register to hold a word read from control store │
        └──────────────────────────────────────────────┘

        ┌──────────────────┐     17 bit control word to
        │    Sequencer     │     ALU and data paths
        └──────────────────┘
```

a) On each clock, the address in CSAR selects one of the words in the control store (note: $512 = 2^9$). This word is read from control store into a register that is part of the control store.

b) This word has the following format:

Sequencing control (11 bits)	Control word to ALU and data paths (17 bits)

    (1)Part of this word (17 bits in this case) comprise the control word which is sent to the ALU and data paths.

    (2)Part of this word (11 bits in this case) serves as input to a sequencer, which determines the address of the next microinstruction to be executed and places it in CSAR.

2. The sequencing control part of the word contained in control store would need two fields

    a) A 9 bit next micro-word address field that contains the address of the next microword. (Thus, each microword explicitly contains the address of its successor). This field is called "next".

    b) A 2 bit field used to allow branching in the microprogram - we'll discuss this shortly. This field is called "decode".

OMIT THE FOLLOWING (THROUGH P. 14) IF INSUFFICIENT TIME

3. Sequencing could be handled as follows:

    a) Ordinarily, decode is 0 and next contains the address of the next control word.

    b) If decode is non-zero, then some additional values are "orred" with next to form the address of the next instruction.

Decode	Orred with next

01   op of the instruction contained in IR, multiplied by 4 (therefore in the range 0 .. 11111100)

10   func field of the (R Type) instruction contained in IR (therefore in the range 0 .. 11111)

11   Result of comparison between registers selected by rs and rt (0 if not equal, 1 if equal)

This allows a form of conditional branching in the micro-program - e.g. if next contains 100000000 and decode is 01 and op in the IR is 000101 , then the next micro-instruction to be executed will be taken from location 100000000 or 00010100 = 100010100 in control store.

4. Control store could be organized as follows. (Note that quite a few locations in control store are unused - the structure is set up to facilitate quick computation of addresses by or-ring bits, rather than by doing addition (which takes more time).

  a) Words 0-1: microprogram for fetching and decoding an instruction

  b) 0x100 .. 0x1ff: Control words for executing the various instructions - up to 4 per instruction. (Actually, each instruction needs at most 3, but 4 is a power of 2 and allows us to multiply the op-code by shifting)

    The control words for a particular instruction are a the four successive locations beginning at 0x100 + 4 * opcode.

  c) 0x80..0xbf: Final control word of RType instructions (handled separately because the last control word for JR is different from other RType instructions)

    The final control word for a particular RType instruction is at address 0x80 + func.

d) Most of the remaining locations in the range 2..0x7f are unused. However, a few instructions needs some additonal control words which could go anywhere - e.g.

4-5: Final control word of beq instruction - first for registers not equal (don't branch); second for registers equal (branch)

6-7: Final control word of bne instruction - first for registers not equal (branch); second for registers equal (don't branch)

5. We now consider what the beginning of the microprogram for MIPS might look like:

Location in control store	Contents
000000000	Control word: `IR <- M[PC], PC <- PC + 4` Next: `000000001` Decode: `00`
000000001	Control word: (all zeroes) Next: `10000000` Decode: `01 (op)`

These two control words cause the next machine language instruction to be fetched from memory, and the program counter to be updated. Then, the instruction just fetched is decoded by orring its op-code (times 4) with 100000000 - which causes a branch to the appropriate portion of the microprogram for executing that instruction.

a) We can't decode an instruction as part of 000000000 because the opcode is not loaded into the IR until the clock at the end of the cycle, which is the same time we need to load a new address into CSAR, and thus cannot be used to help determine that address.)

b) This appears wasteful because it adds an extra cycle to each instruction. (I.e. most instructions now use 5). In practice, a richer ISA has operations that can be done speculatively at this point - e.g. `MAR <- Address portion of instruction` - not needed for every instruction but needed for enough to mak it worthwhile

6. DEMO: Lab 5 Part 1 program - note values in next / decode / CSAR at each step.

D. Advantages/disadvantages of micro-programming

1. Advantages

a) Great sophistication in the user instruction set can be achieved for relatively low cost. Adding new instructions is cheap. (This makes complex instruction sets possible - using hardwired control for a typical CISC ISA would be impractical due to the complexity).

b) Multiple user instruction sets can be available on the same machine. This allows a new machine to emulate a previous model to aid in the conversion process - e.g.

(1) Early IBM 360's contained microcode to emulate 1401's and/or 1620's

(2) Early DEC VAX's emulated PDP-11's.

(3) DEC Alpha's used a form of microcode (though different from what we have discussed here) to emulate VAX's.

c) New architectures can be tried out by simulating them using writeable control store on an existing machine. Special micro-engines have been built for just this kind of work.

d) Micro-code can be written to allow direct execution of high-level languages - e.g. LISP, Pascal.

e) For specialized applications (e.g. real-time systems), critical loops can be microprogrammed for faster execution time.

f) Micro-programmed diagnostics.

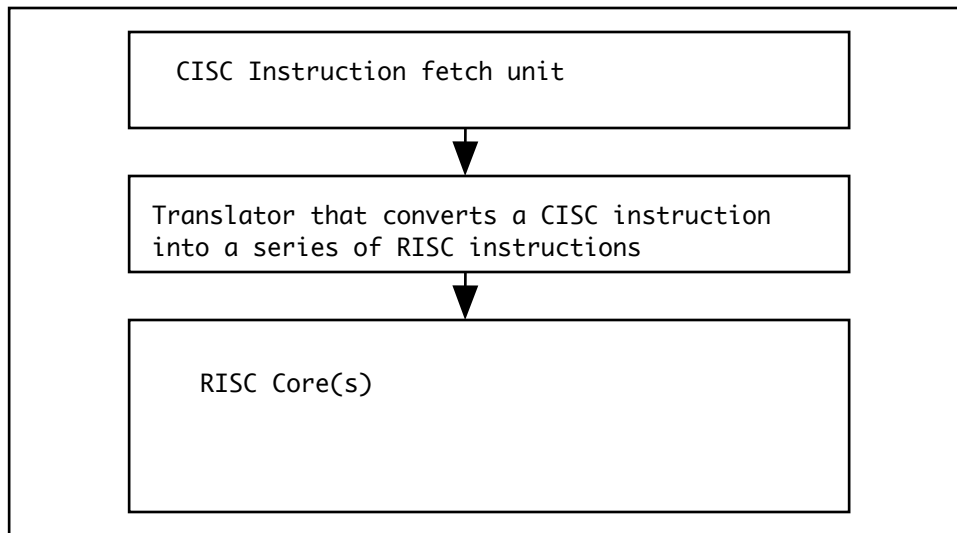g) Bit-sliced processors, allowing implementation of custom machines.

2. Disadvantages

a) For a given level of technology, hardwired control will be faster, since there is no delay for micro-instruction fetch from ROM before the control unit can produce a control word.

b) Does not lend itself well to parallelism.

3. Though some CISCs today are micro-programmed, high performance systems use another strategy (which the book called vertical microprogramming, though that's really referring to something quite a bit different which we won't discuss).

RESUME HERE IF NECESSARY TO OMIT THE ABOVE DUE TO TIME

VII.**Implementing a CISC with a RISC core**

A. High-performance CISCs cannot be built using hardwired control, due to the complexity, and are not built using microprogrammed control, for performance reasons.

B. Instead, the following is the way a high-performance CISC may be structured. (Note: there's not a lot of detail available about this structure, because manufacturers in a competitive industry don't tend to publish a lot of details about the internals of their systems!)

```
┌─────────────────────────────────────────────────────┐
│  ┌───────────────────────────────────────────────┐  │
│  │   CISC Instruction fetch unit                 │  │
│  │                                               │  │
│  └───────────────────────────────────────────────┘  │
│                        │                             │
│                        ▼                             │
│  ┌───────────────────────────────────────────────┐  │
│  │  Translator that converts a CISC instruction  │  │
│  │  into a series of RISC instructions           │  │
│  └───────────────────────────────────────────────┘  │
│                        │                             │
│                        ▼                             │
│  ┌───────────────────────────────────────────────┐  │
│  │                                               │  │
│  │     RISC Core(s)                              │  │
│  │                                               │  │
│  │                                               │  │
│  │                                               │  │
│  └───────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────┘
```

1. That is, inside a high-performance CISC is a RISC core (or, in some cases two). The RISC cores use standard performance enhancement techniques such as pipelining to maximize performance.

2. CISC instructions are fetched by an instruction fetch unit according to the rules of the CISC ISA.

3. Each CISC instruction is then translated into one or more RISC instructions, which are executed by the core(s).

C. In a system like this, the only ISA the programmer ever sees is the CISC ISA - the ISA of the RISC core is completely shielded from view.