CPS311 Lecture: CPU Control: Hardwired control and Microprogrammed Control

Last revised July 25, 2021

Objectives:

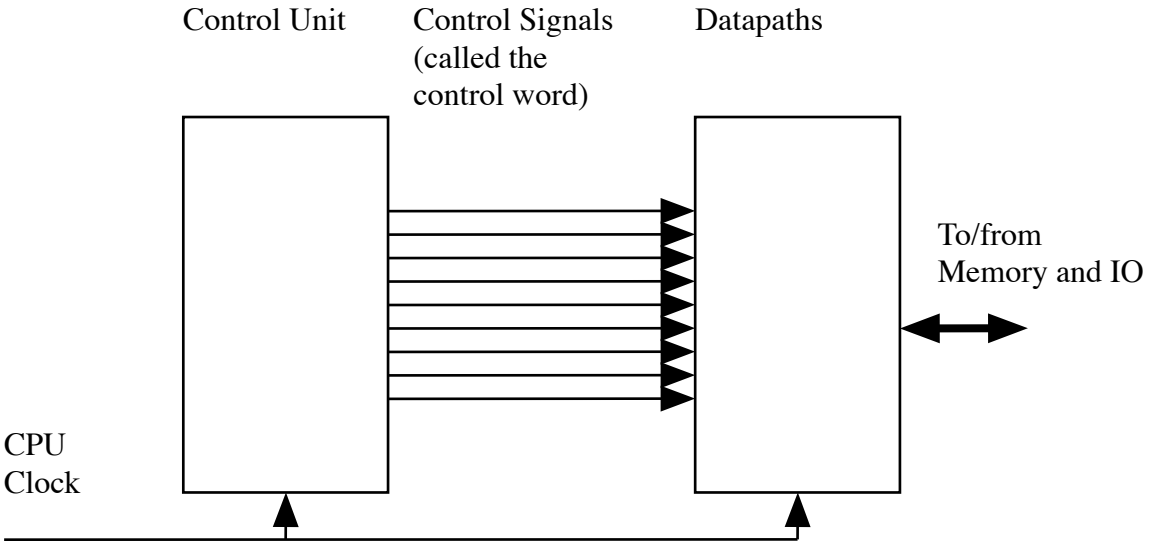
1. To explain the concept of a control word
2. To show how control words can be generated using hardwired control
3. To explain the concept of microprogramming
4. To discuss the use of a RISC core in a CISC processor

Materials:

1. Projectable Version of Diagrams
2. RTL Handout (will already have)
3. mmips Simulation
4. Code for Lab 5 - part 1 to demonstrate using microprogramming

I. Introduction

A. Recall that any CPU - whether simple or complex - consists of three principal parts:



(We continue to assume a single core. A multi-core CPU actually has two or more instances of the Control Unit an Data Paths)

PROJECT

B. The datapaths are capable of performing a set of microoperations or primitive computations that can be performed in one cycle (clock pulse). Each microoperation typically 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:

D. The set of control signals that pass from Control to the data part and bus system is called a 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)

   We looked at an example of how 17 of these could be used for our example MIPS implementation - but of course these would vary greatly from implementation fot implementation fo the same ISA or between different ISAs.

E. The job of the control unit for such CPU is to develop a means whereby an orderly sequence of control words may be presented to the datapaths (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.

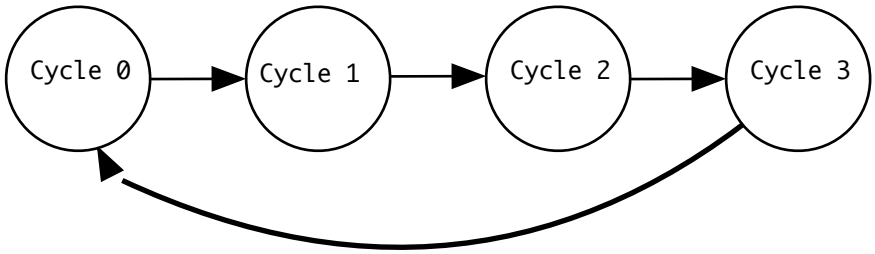
2. Microprogrammed control. The various control words needed to implement the user instructions are stored in a ROM, with a sequencer causing the appropriate control word to be fetched at each clock cycle and fed to the rest of the CPU.

II. 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 multicycle 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:

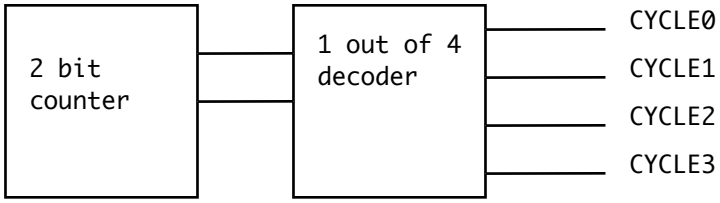


PROJECT

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. (We'll see laer how the ISA makes this sort of the implementation easy)

b) Actually, a full implementation would need additional states to deal with issues like interrupts and exceptions - but we will not get into this aspect.

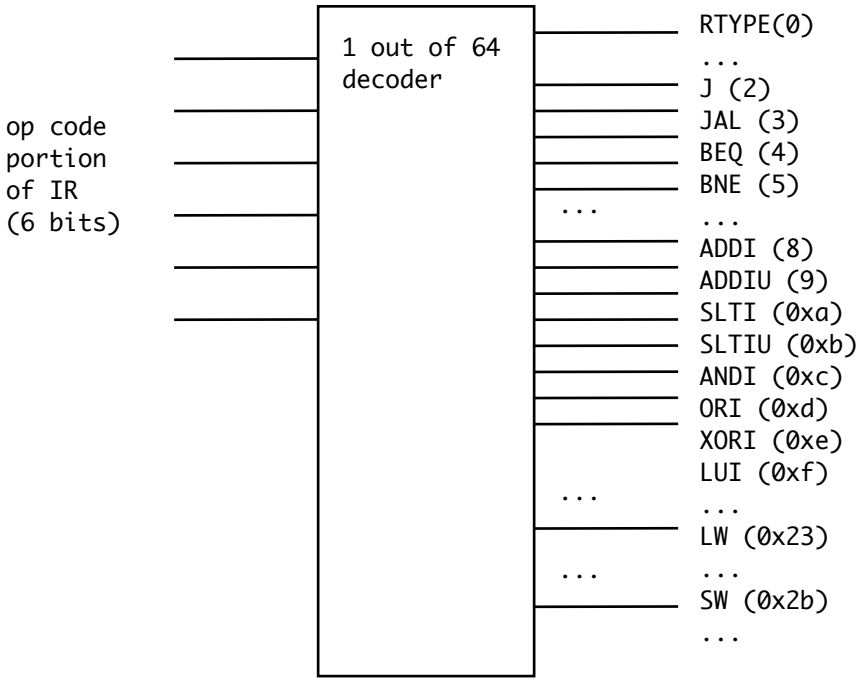
c) This simplified state machine can be realized by a 2 bit counter, with its output decoded to yield 4 signals used internally in the control unit.



PROJECT

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

C. It will simplify our example implementation if we use a decoder to decode the op-code part of an instruction.



PROJECT

D. We will now explore the MIPS simulation we have been looking at in more detail. Recall that the control word for this simulated implementation contains 17 bits.

PROJECT MIPS Simulation and show effect of each bit on the data paths

1. Three bits (shown as checkboxes on the simulation's manual input panel) disable/enable the loading of a particular register on a clock pulse.

2. Two bits control whether a memory read or write is done (cannot both be true at the same time, of course - but often are both false).

3. One bit to control where the address of a memory location to be read or written comes from (the PC or a value calculated in the ALU).

4. Several bits control <u>what</u> is loaded into a given register.

5. 2 bits to control <u>which</u> register in the register set is loaded (determined by rd field of instruction, determined by the rt field, or register 31 (required for JAL))

6. 3 bits to determine the operation performed in the ALU along with the funct field in the instruction. (For the immediate instructions, the funct field in the instruction is actually part of the constant.)

E. 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.

ASK CLASS TO TAKE OUT RTL HANDOUTS - PAGE 2

1. 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. Where does occur in the RTL?

      ASK

      This is 1 on Cycle 0 of all instructions, and 0 everywhere else. Thus, we can derive this bit as

      CYCLE0 ———————— LOADIR

      PROJECT

b) Example: the Load PC bit. Where does this occur in the RTL?

ASK

This is 1 in four places, and 0 everywhere else

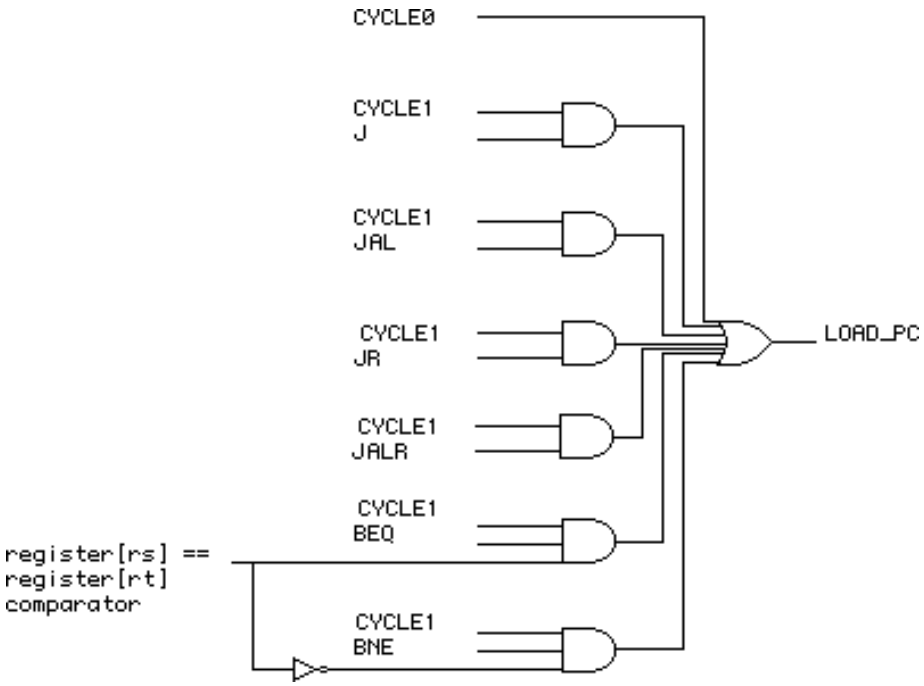
(1)Cycle 0 of all instructions

(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) Sometimes more than one control bit needs to be set to implement a given RTL operation. For example, consider the RTL operation `register[31] ← ALUOutput` . This occurs in the RTL for both the JAL and the JALR instructions in the ISA. But to carry this out, several control bits must be set:

(1)Register load must be 1.

(2)Register to load must be set to $31. (This involves two bits in the control word because it is one of three possibilities for Register load source)

d) Class Exercise: Design derivation for control bit Memory Read.

Boolean equation - ASK

Logic circuit - ASK

e) 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 0, 1 (IR J-Format constant) on Cycle 1 - regardless of what instruction we are executing, 3 (ALU Out) on Cycle 3 - regardless of what instruction se are executing, since this yields the correct value whenever LOAD_PC is 1 and is ignored otherwise

f) Class Exercise: Design derivation for Memory Address Source Bit

Boolean equation - ASKZ

(Note we don't care about what instruction is being executed)

Logic Circuit - ASK

g) This same process can be continued for each bit of the control word.

2. 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.

III. **Microprogramming**-

A. As you can see, for even a very simple machine like the one we just looked at, hardwired control leads to fairly complex control logic. For a CISC, 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. Historically this also allowed for:

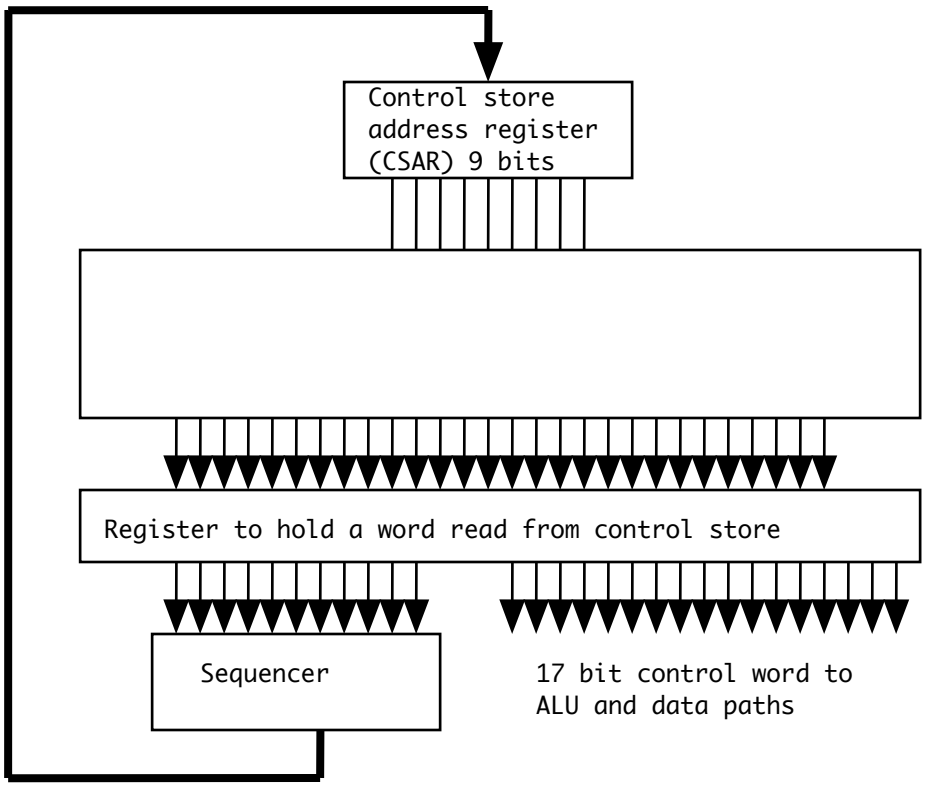
   a) Dynamic microprogramming - e.g. for adding custom user instructions to the standard set or emulating another machine.

   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.

(This was more relevant when CPU's were implemented by mulitiple circuit boards rather than on a single chip_

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)

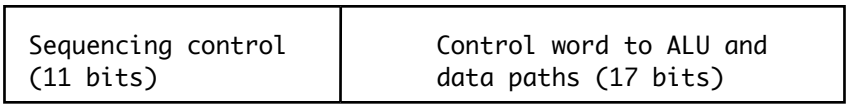
1. Structure of the control unit. (All micro-programmed CPU's use a structure like this, but of course the specific sizes will vary)



PROJECT

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:



PROJECT

(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)

PROJECT

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 additional 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)`

PROJECT

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 using microprogrammed control - 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 could be achieved for relatively low cost. Adding new instructions is cheap. (This made complex instruction sets possible - using hardwired control for a typical CISC ISA would be impractical due to the complexity).

b) Multiple user instruction sets could be available on the same machine. This allowed 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 could 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 could 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.

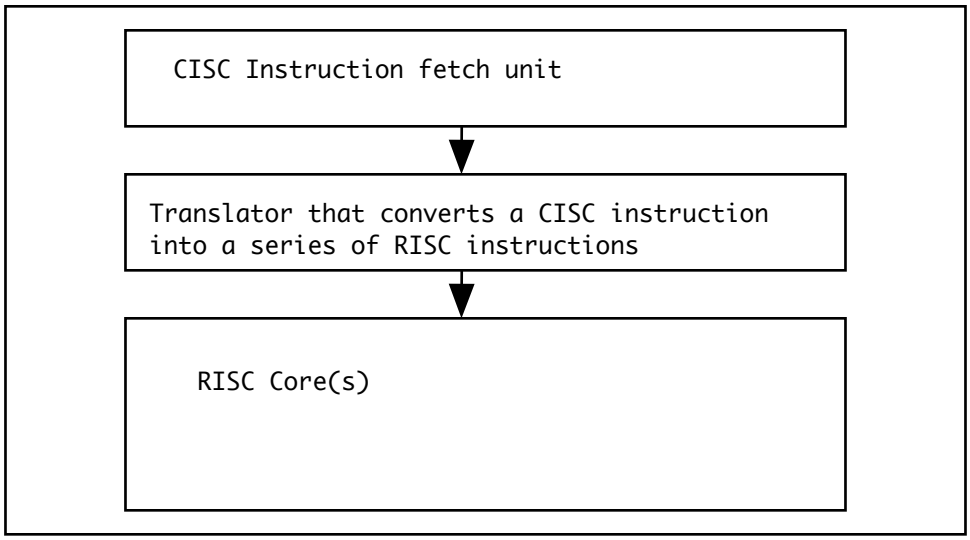
3. Though historically CISC architectures were generally implemented using microprogramming, another approach we will next look at tis generally used for today for a CISC architecture like x86.

RESUME HERE IF NECESSARY TO OMIT THE ABOVE DUE TO TIME

IV. **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!)



PROJECT

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.