Branching in the ACS-1

Edward H. Sussenguth

BACKGROUND

The ability to make a decision gives the computer its power. For example, we ask a computer to sort information or to extract data similar to a given pattern, both powerful decision constructs. Without this ability, a computer would simply be a fast calculator or slide-rule.

At the processor level (i.e., assembly-language level) decision-making instructions are variously called: *branch, transfer,* or *jump*. For simplicity, we shall use branch. In this paper we investigate aspects of branch instructions as incorporated into the design of the ACS-1 [1,2].

Normally, a computer fetches instructions from memory in sequence, one after another. The branch instruction interrupts this flow by switching the sequence to another memory location. We call the "other" location the effective branch address (EBA). In the ACS-1 design, the EBA is either provided in an index register or calculated by adding the contents of an index register to a constant (i.e., immediate) field in the instruction.

Logically, a branch instruction consists of three parts:

(1) whether or not the branch is to be taken (i.e., the condition);
(2) the address to which the branch is to be made if successful (i.e., the EBA); and,
(3) when the branch is to be taken (i.e., the exit point specification).

Most computers combine all these components (test something, calculate the EBA, and branch after this instruction). Thus, the normal idea is that, when a branch occurs in the instruction stream, the result of that branch (i.e., to branch or not to branch) occurs immediately following it. However, this is not absolutely necessary, and the three parts may be separated.

BRANCH-AT-EXIT and EXIT

To explicitly designate when a branch is to be executed, I, along with John Cocke, Brian Randell, and Herb Schorr, invented a new instruction for ACS-1 called EXIT, and we renamed the branch instruction as BRANCH-AT-EXIT [3]. (This form of branch instruction was also called “Prepare to Branch.”) To implement the latter, as soon as possible the machine calculates both whether the branch condition is successful or not (part 1) and the EBA (part 2), but it will not actually perform the branching until an EXIT instruction is seen in the instruction stream (part 3). For example, for a *goto*, the EBA can be computed early in the instruction stream. However, when the EBA must be calculated (as in a *switch* statement), the EBA calculation may be quite close to the EXIT instruction, thereby losing the advantage.

Consider the structure of a loop for summing the elements of a 100-element array of single-precision floating-point values (sum = Σ x[i]). In C++ this could be written as:

```
     sum = 0.0;
     for (int j = 0; j < 100; j ++)
        sum = sum + x[j];
```

Coding using normal branches might be (here we use a normal branch-on-less-than-index instruction, BLTX):

```
     SX    1, 1, 1        clear index register 1 to hold array element index
     SN    2, 2, 2        clear arithmetic register 2 to hold partial sum
top  LAT   4, x(1)        load arithmetic register 4 with the value of x[j]
     AN    2, 2, 4        add to sum
     AXC   1, 1, “1”      add constant 1 to index register 1
     BLTX  1, “100”, top  branch to top if value in index register 1 < 100
     STA   2, sum         store arithmetic register 2
```

Coding using BRANCH-AT-EXIT and EXIT instructions might be (condition bit 0 is always 0):

```
     SX     1, 1, 1     clear index register 1 to hold array element index
     SN     2, 2, 2     clear arithmetic register 2 to hold partial sum
top  CGEXK  3, 1, ”99”  set condition bit 3 if value in index register 1 >= 99
     BEQ    3, 0, top   branch-at-exit when condition bit 3 is false
     LAT    4, x(1)     load arithmetic register 4 with the value of x[j]
     AN     2, 2, 4     add to sum
     AXC    1, 1, “1”   add constant 1 to index register 1
     EXIT               change instruction sequence if branch condition met
                        (i.e., value in index register 1 < 99 at top of loop)
     STA    2, sum      store arithmetic register 2
```

In the ACS-1 design, instructions between the BRANCH-AT-EXIT and EXIT instructions are executed normally, independent of whether the branch condition is successful or unsuccessful. The important difference between the two segments of code is that the BEQ instruction occurs three instructions (floating-point load, floating-point add, and index register increment) earlier than the EXIT instruction. This allows the instruction sequencing unit additional time to determine the validity of the branch, to calculate its EBA, and pre-fetch the proper instruction. Then, when the EXIT instruction is encountered, less time is lost preparing for the correct instruction (either the CGEXK instruction at ‘top’ or the STA instruction that saves the result to memory). Moreover, because a floating-point add typically requires more than one machine cycle, ACS-1’s indexing unit (in which the AXC address increment requires only one cycle) can be significantly ahead of the floating-point unit, which then allows the floating-point operands to be fetched earlier.

MULTIPLE CONDITION CODES AND MULTI-WAY BRANCHES

The ACS-1 design has 24 condition bits in a condition-code register, and the BRANCH-AT-EXIT instruction combines any two of these via any of eight Boolean functions to determine the success or failure of the branch. This permits less discontinuity in the instruction stream. For example, the high-level segment:

```
     if (A and B) goto α
     β <expressions>
```

becomes with:
```
     conventional coding           multiple condition-code tests

     BRANCH (if ~A) to β           BRANCH (if A and B) to α
     BRANCH (if B) to α          β <generated code>
   β <generated code>
```

Perhaps even more importantly, ACS-1 branching provides for multi-way branch specification. That is, multiple branch instructions can be executed and thereby set up multiple branch conditions and associated target addresses in a special exit table. Only the first successful entry is used by the EXIT instruction, and thus only one non-sequential change of the instruction counter is required.

For example, consider the nested if-then-else structure:

```
     if (condition_1 || condition_2) goto α
     else if (condition_3 && condition_4) goto β
     else if (condition_5 == condition_6) goto β
   γ <expressions>
```

Using multiple condition-codes (where the conditions have been set in the corresponding condition bits):

```
     BOR   1, 2, α
     BAND  3, 4, β
     BEQ   5, 6, β
     EXIT
   γ <generated code>
```

The EXIT instruction causes a change in instruction sequence to the EBA in the first exit table entry that contains a successful condition. There is no change in instruction sequence if all conditions are unsuccessful. In either instance, the execution of the EXIT instruction deletes all current exit table entries so that the next BRANCH-AT-EXIT instruction can begin with an empty exit table.

SKIP

A variant of the BRANCH-AT-EXIT, EXIT combination is the SKIP instruction [4]. A SKIP instruction takes advantage of the ability to easily invalidate specially-marked instructions so that they never contend with other instructions.

In the ACS-1 design, each instruction contains a “skip” bit. The SKIP instruction has multiple condition-code tests as described above and sets the skipping state for the processor. Subsequent instructions with the skip bit set execute only when the skipping state is false.

Consider the example:

```
     if ( (x != 0.0) && (y != 0.0) )
       z = x / y;
```

Coding using normal branches might be (arithmetic register 0 and index register 0 are always 0):

```
     LA    1, x     load arithmetic register 1 with x
     LA    2, y     load arithmetic register 2 with y
     BEQA  1, 0, β  branch if value in arithmetic register 1 == zero
     BEQA  2, 0, β  branch if value in arithmetic register 2 == zero
     DN    3, 1, 2  divide registers 1 and 2 and place result in register 3
     STA   3, z
  β
```

Coding using SKIPs might be (where * indicates that the skip bit is set):

```
     LA    1, x     load arithmetic register 1 with x
     LA    2, y     load arithmetic register 1 with x
     CEQN  4, 1, 0  set condition bit 4 if value in arith. reg. 1 == zero
     CEQN  5, 2, 0  set condition bit 5 if value in arith. reg. 2 == zero
     SKOR  4, 5     set skipping state true if condition bit 4 or 5 is true
     DN*   3, 1, 2  divide registers 1 and 2 and place result in register 3
     STA*  3, z
  β
```

Here, the advantage of the SKIP is that the two *-ed instructions may be entirely omitted in the instruction sequencing unit when the skip condition is true, and thus the disruption caused by a branch is avoided.

INSTRUCTION SEQUENCING UNIT

The instruction sequencing unit is that part of a computer which determines which instruction is to be executed next and fetches it from memory. The simplest instruction sequencing units fetch the next instruction when the current instruction has completed. These are inexpensive but slow, and more complex sequencing units are needed for better efficiency [5].

Almost all of today's computers pre-fetch instructions. That is, after fetching an instruction from location *n*, the instruction sequencing unit pre-fetches the instruction from location *n+1*. More sophisticated sequencing units may use branching history to control the prefetching.

The ASC-1 design includes a hardware facility called pre-fetch control registers to contain pairs of addresses: the first is the address of the last branch instruction encountered, and the second is the address which was taken as result of that branch, either *n+1* or EBA [6]. Using this facility, the sequencing unit can place the address of a branch instruction (address *n*) in the first register of the pair and either *n+1* or the EBA in the second register. When the sequencing unit subsequently encounters address *n*, it will fetch according to the address in the second register. Thus, for a loop, the sequencing unit fetches the wrong instruction twice, once the first time through the loop and once at the last time.

As an example consider the common loop (here α and β are used only to aid the explanation; they are not C++ labels):

```
     j = 0;
```

```
α while(j < 100)
    {
      op
      op
      …
      j++;
β }
```

On the first iteration (when j is 0), the pre-fetch control registers are {?,?} (i.e., whatever they were prior to this loop). Upon reaching location β, which will be a loop-closing branch instruction in conventional computers or an EXIT instruction in ACS-1, the instruction sequencing unit will fetch the instruction at β+1, which is incorrect, as it should have fetched the instruction at α. However, the pre-fetch control registers will be set to {β, α}.

On the next iteration (when j is 1), and upon reaching location β, the instruction sequencing unit will fetch the instruction at α, which is correct and many machine cycles are saved. The pre-fetch control registers remain {β, α} for iterations 2 through 99. When j is 99, instruction sequencing unit will fetch the instruction at α, which is incorrect; β+1 should have been fetched. Thus, the pre-fetch control registers saved cycles on all but two of the iterations.

To extend this pre-fetching idea, we defined a set of such registers (say four). When a branch is encountered, its address is tested against the four first addresses; if there is a match, its EBA replaces the corresponding second address and the pair is moved to the bottom of the set. If there is no match, its address replaces the first address in the top register pair, and its EBA replaces the corresponding second address; the pair is moved to the bottom of the set. The movement to the bottom ensures that the most recently encountered branches are remembered. The use of a set of pre-fetch control registers was first introduced by ACS-1, but the concept has been used in other computers as well (e.g., the jump trace buffer in the MU-5 [7].)

START I/O

System/360 has an instruction to initiate an input-output operation, called SIO. When SIO is issued, a signal is sent from the processor to the channel, and then to its control unit, and finally to the I/O device. The processor does not execute the instruction following SIO but waits for a response. The I/O device determines its state (ready to accept the command or not) and sends a signal back to its control unit, and then to the channel, and finally to the processor. This back-and-forth is extremely time-consuming for high-speed processors.

ACS-1 invented a small modification to SIO and called it start-I/O-fast-release, or SIOFR. The difference between SIO and SIOFR is the processor does not wait for the reply from the device. It simply executes the next instruction in sequence. When the reply is received, a soft interrupt is set; few modifications to the operating system are required to handle SIOFR. (SIOFR was eventually accepted as a System/360 instruction.)

INTERRUPTS

Although an interrupt is not a branch, strictly speaking, it is closely related.

An interrupt occurs when an unusual condition occurs such as over- or underflow in a multiply or divide instruction or an abnormal input-output response. Typically, when an interrupt occurs, the main instruction stream is broken and the next instruction is fetched from a predetermined location.

However, the ACS-1 design has two ways to handle interrupts, called hard and soft interrupts. Over- and underflow are examples of hard interrupts: action must be taken immediately. SIOFR is an example of the usage of a soft interrupt: action may be delayed.

To process soft interrupts a specialized, non-programmable, branch instruction (IC - interrupt call) is inserted into the instruction stream with its EBA set to a predetermined exception-handling location. IC is fully interlocked, and it allows all instructions prior to it to complete but none of the interrupt handler instructions to start early. A companion instruction (IR – interrupt return) is provided to resume the interrupted program.

To handle hard interrupts, program execution is immediately stopped and all important internal processor state (such as all registers, instruction stacks, and scheduling matrices) is saved to memory. A special privileged instruction, SCAN, is provided for the operating system to reload the internal data and restart the processor from where it left off.

Our estimates were that soft interrupts would have a variable response latency, from 0.1 up to 15 microseconds depending on the instruction mix in flight at the time of the IC insertion, while hard interrupts would have a fixed response latency of 2 microseconds.

ACKNOWLEDGMENTS

The author is indebted to the many individuals who contributed to the IBM ACS-1 design, and in particular to John Cocke who inspired so many innovative ideas. Fran Allen, Dick Arnold, Phil Dauber, Charlie Freiman, Leo Hasbrouck, Merle Homan, Jake Johnson, Bill Madden, Robert Rew, Brian Randell, Russ Robelen, Don Rozenberg, Herb Schorr, and John Wierzbicki to name but a few, contributed on an hour-to-hour day-to-day basis, making the ACS project a very exciting place to work.

Mark Smotherman has been of major assistance, not just for his comprehensive work on ACS-1 [2], but for significant help preparing this paper.

REFERENCES

1. H. Schorr, "Design Principles for a High-Performance System," Proceedings of the Symposium on Computers and Automata, New York, April 1971, pp. 165-192.

2. M. Smotherman, "IBM Advanced Computer Systems (ACS) – 1961 – 1969," web site: www.cs.clemson.edu/~mark/acs.html

3. J. Cocke, B. Randell, H. Schorr, and E. Susssenguth, “Apparatus and method in a digital computer for allowing improved program branching with branch anticipation reduction of the number of branches, and reduction of branch delays,” U.S. Patent 3,577,189, May 1971.

4. J. Cocke, P. Dauber, H. Schorr, and E. Sussenguth, “Apparatus in a digital computer for allowing the skipping of predetermined instructions in a sequence of instructions, in response to the occurrence of certain conditions,” U.S. Patent 3,577,190, May 1971.

5. L. Hasbrouck, W. Madden, R. Rew, E. Sussenguth, and J. Wierzbicki, “Instruction execution unit,” U.S. Patent 3,718,912, February 1973.

6. E. Susssenguth, “Instruction sequence control,” U.S. Patent 3,559,183, January 1971.

7. R. Ibbett, “The MU5 instruction pipeline,” The Computer Journal, vol. 15, February 1972, pp. 42-50.