Control Transfer (Flow) Instructions

Control transfer instructions (CTIs) ...
... may cause next instruction to be fetched from ...
... somewhere other than $\mathrm{PC} + 4$ (assuming 4-byte instructions).

(Called control flow instructions in book.)

Names used for CTIs vary by architecture.

Names used below are common, if not standard.

Four Types:

- *Jump*:
  Unconditional control transfer.
  Accounts for 6% of CTIs in test code.

- *Branch*:
  Conditional control transfer.
  Accounts for 81% of CTIs in test code.

- *Call*:
  Unconditional control transfer, PC, etc. saved.
  With returns, accounts for 13% of CTIs in test code.

- *Return*:
  Unconditional control transfer, PC, etc. from most recent call restored.

Destination Address in CTIs

Any addressing mode *could* be used for destination.

Several are common:

- *Absolute*
  Destination address is an immediate.

  Best for procedure calls...
  ...because destination can be far away.

- *PC-Relative*
  Destination is immediate added to program counter.

  Good for conditional branches...
  ...because destination usually close by...
  ...and so small immediates suffice.

- *Register Indirect*
  Destination in register.

  Used for subroutine return addresses.

  Also used by ISAs in which immediates smaller than addresses.

- *Displacement*
  Destination is sum of two registers.

  Useful for C `switch` and similar statements.

Branch Conditions

Branch condition used for branch instructions.

How branch condition determined:

- Test value of general-purpose register (GPR).
- Test value of special-purpose condition code register (CCR).
- Condition based on outcome of last arithmetic operation.
- Comparison specified in branch instruction.

  Note: *test* value means *test if value is zero* ...
  ...which is much faster than *test if value greater than constant*.

Factors

Compact code, programmer convenience:

⇒Comparison in branch instruction.

Fast implementation:

Determine condition several instructions before branch.

⇒Test GPR. (But may "waste" registers.)

⇒Test CCR. (Maybe limited to one condition at a time.)

Lowest possible cost (relevant to older technology):

⇒Based on last arithmetic operation.

Procedure Call and Return

Procedures (A.k.a., subroutines, functions.)

Fundamental part of every nontrivial program.

Requires careful support in ISA.

Mandatory ISA Support

Call instruction saves PC in register.

Return restores saved PC.

Additional Support, Provided by ISA or Software (*ABI*).

Save and restore registers.

Prepare *stack frame* of called procedure.

Application Binary Interface (ABI)

Rules for writing machine-language programs.

More restrictive than ISA ...
... but not enforced by hardware.

Code adhering to ABI rules called *compliant*.

Given an ABI, ...
... any compliant procedure ...
... can call any other compliant procedure ...
... (if call parameter and return value types match).

⇒ABI determines how "Additional Support" provided.

Procedures and the Stack

Only procedure using top of stack can be running ...
... and thus can make a call.

Local memory for procedures provided on the *stack*.

Each procedure invocation has own part of stack.

Procedures and Registers.

Processor has one set of registers (usually), ...
... so register contents must be *saved* and *restored* ...
... for each procedure call and return.

If *caller-saved*, registers saved before call.

If *callee-saved*, registers saved after call.

Systems frequently use both.

Stack

Each thread[1] has own stack.

Area of memory storing data ...
... for each current procedure invocation.

For each procedure invocation stack may store:

Local variables.

Copies of register contents, including PC; ...
... (not always up to date).

Arguments for called procedure (if any).

[1] If you don't know what a thread is, ignore it.

Procedure Call Steps

For a procedure to make a call it ...

... must put call arguments in a predefined place, ...

... may save some or all of its registers in its part of the stack, ...

... must save the program counter (in a register or stack), ...

... must jump to the called procedure.

The called procedure ...

... must add space to the stack (move the top of stack up) for its own use ...

... may save register values, ...

... and start executing its own code.

Procedure Return Steps

For a procedure to return it ...

... must have any return value in the specified place, ...

... may restore registers used by the caller, ...

... must remove the space it allocated for the stack (move the top of stack down) ...

... restore the program counter to its previous value.

The returned procedure ...

... may restore some of its registers.

Implementation of Call and Return Steps

Mostly Hardware

Powerful call and return instructions do most of the work.

Call instruction ...
... saves program counter and other registers.

Return instruction ...
... adjusts stack and restores registers.

Mostly Software

Simple call and return only handle program counter.

Remainder done by general-purpose instructions ...
... using ABI guidelines.

Before call, using general-purpose instructions, ...
... procedure may save some registers.

Call instruction ...
... places return address in an ABI-specified register.

Called procedure, using general-purpose instructions, ...
... adjusts stack and may save registers.

Procedure return is similar.

CTI Variations

CTI Behaviors Chosen to Speed Implementation

- *Delayed Transfer*
  Control transfer occurs $d > 1$ instructions after CTI.

  E.g., consider execution of instruction 1 of DLX code:

```
1  J 1000   ! Jump to address 1000.
2  ADD R1,R1,R1
3  ADD R2,R2,R2
4  ADD R3,R3,R3
5  ADD R4,R4,R4
```

  Normally, instruction 2 not executed.

  When $d = 2$ instruction 2 is executed, but not 3, 4, and 5.

  When $d = 3$ instruction 2 and 3 are executed, but not 4 and 5.

- Branch Instructions with Prediction Hints
  Programmer indicates whether branch is likely.

  If programmer correct, execution may be faster.

- Predicated Execution
  Non-CTI instructions that only execute if some condition true.

  E.g., `movg r1,r2`, meaning ...
  ... move `r1` to `r2` if greater-than condition true. (Sun V9).

Size and Type of Operands

Common Sizes

- *Byte, char, octet.* 1 byte (8 bits here).
- *Half word.* 2 bytes.
- *Word.* 4 bytes.
- *Doubleword.* 8 bytes.
- *Quadword.* 16 bytes.

Common Types with Sizes

- *Unsigned integer* and *integer.* Byte, half word, word, doubleword.

  Integers are sign-extended when moved into a larger register ...
  ... while unsigned integers are not.

- *Floating-point.* Word, doubleword, quadword.

- *Packed BCD.* Word, etc.

  Each word holds several BCD digits of a fixed-point number.

  E.g., word holds a 8-digit BCD integer.

  Decimal fractions such as .03 exactly represented.

  Used for financial computations, typically in Cobol programs.

  Used primarily in older architectures.

- *Packed integer, packed fixed-point.* Word, double word.

  Holds several small integer or fixed-point values.

  Used by *packed-operand* instructions which operate on each small value in parallel.

  Used in newer ISA versions. *E.g.*, Sun VIS and Intel MMX.

Data Type Usage

Floating point: double, 69% and word, 31%.

Integer word, 75%; halfword, 19%; and byte, 7%.

Size Tradeoffs

Integer: size of fastest integer (usually) equals address size.

E.g., word on a 32-bit machine, doubleword on a 64-bit machine.

On most machines a smaller integer saves space, but not time.

Floating-point: doubleword usually best choice.

Word may be faster, but can be slower ...
... when double result must be rounded to word size.

Data Types and ALU Operations

How data type specified:

In opcode. (Used in many ISAs.)
Integer multiply instruction, floating-point add.

In instruction type field. (Used in many ISAs.)

Tagged, type specified in data. (Used in a few ISAs.)

Suppose data type were word-sized, ...

... 30 bits might hold the number ...

... 2 bits would indicate what type the data was ...

... such as integer, unsigned integer, float, or string.