Instruction Usage

Usage of DLX Instructions By SPEC92 Integer Code

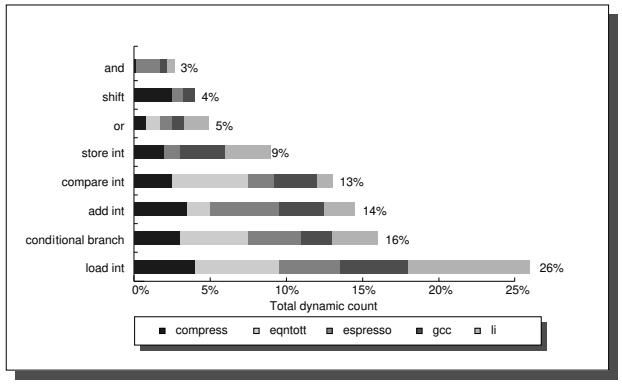

FIGURE 2.28 Graphical display of instructions executed of the five programs from SPECint92 in Figure 2.26.

04-1

Instruction Usage

Usage of DLX Instructions By SPEC92 Floating-Point Code

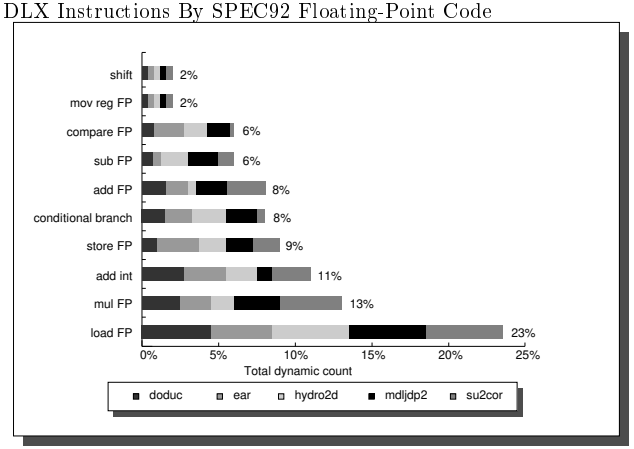

FIGURE 2.29 Graphical display of instructions executed of the five programs from SPECfp92 in Figure 2.27.

04-2

Control Transfer (Flow) Instructions

```
 beqz r1, SKIP  ! If r1 = 0 goto SKIP
 add  r2, r2, r5
SKIP
 add  r2, r3, r4
```

Control transfer (control flow) instructions (CTIs) ...
... may cause next instruction to be fetched from ...
... somewhere other than PC + 4 (assuming 4-byte instructions).

Programs would be trivial without them.

Occur frequently in programs.

Often a performance bottleneck.

04-3

Names used below are common:

- *Branch*:
  Conditional control transfer.

- *Jump*:
  Unconditional control transfer.
  Sometimes special case of branch.

- *Jump and Link*:
  Unconditional, return address (PC) saved in register.

- *Call*:
  Unconditional control transfer, PC, etc. saved.
  Sometimes special case of jump and link.

- *Return*:
  Unconditional, PC, etc. from most recent call restored.
  Sometimes special case of jump and link.

04-4

CTI Examples

C Code

```
 if( r2 == r3 ) goto TARGET1;
 r4 = r5 + r6;
 goto TARGET2;
TARGET1:
 r4 = r5 * r6;
TARGET2:
 PROCNAME();
 return;
```

DLX Assembler Code

```
 sub  r1, r2, r3   ! r1 = r2 - r3
 beqz r1, TARGET1  ! Branch if r1 = 0.
 add  r4, r5, r6   ! r4 = r5 + r6
 j    TARGET2      ! Jump unconditionally.
TARGET1:           ! Label. (Assembler and linker find address.)
 mult r4, r5, r6   ! r4 = r5 * r6
TARGET2:           ! Another label.
 sw   16(r20),r31  ! Save return address (of caller to this proc.)
 jal  PROCNAME     ! Call procedure PROCNAME.
 lw   r31,16(r20)  ! Restore return address (overwritten by jal).
 jalr r31          ! Return (from this procedure).
```


DLX Assembler Code (Same as last slide.)

```
 sub  r1, r2, r3   ! r1 = r2 - r3
 beqz r1, TARGET1  ! Branch if r1 = 0.
 add  r4, r5, r6   ! r4 = r5 + r6
 j    TARGET2      ! Jump unconditionally.
TARGET1:           ! Label. (Assembler and linker find address.)
 mult r4, r5, r6   ! r4 = r5 * r6
TARGET2:           ! Another label.
 sw   16(r20),r31  ! Save return address (of caller to this proc.)
 jal  PROCNAME     ! Call procedure PROCNAME.
 lw   r31,16(r20)  ! Restore return address (overwritten by jal).
 jalr r31          ! Return (from this procedure).
```

Branch instruction: `beqz`.

Jump instruction: `j`.

Call instruction: `jal`.

Return instruction: `jalr`.

Note: call and return are sometimes special case of *jump and link* instructions.

Usage of CTI By Type

CTI by type MIPS running SPEC92 benchmarks.

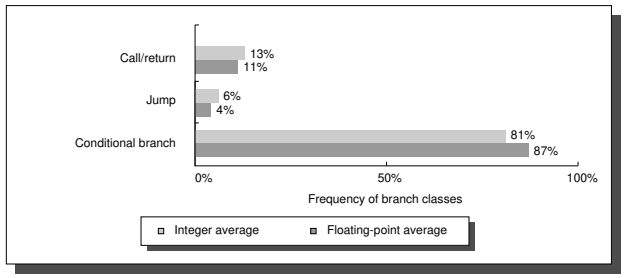

FIGURE 2.12 Breakdown of control flow instructions into three classes: calls or returns, jumps, and conditional branches.

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* or *Displacement*
  Destination is *displacement* added to program counter.

  ISA design choice: displacement size.

  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.

- *Indexed*
  Destination is sum of two registers.

  Useful for C `switch` and similar statements.

Destination Address Examples

Absolute and PC-Relative Examples

```
 jump_abs TARGET1    ! Instruction contains entire address, TARGET1.
 add r1, r2, r3
TARGET1:             ! When jump_rel executes, PC = TARGET1.
 jump_rel TARGET2    ! Instruction contains TARGET2-TARGET1.
 sub r4, r5, r6
TARGET2:
 and r7, r8, r9
```

To compute target address of `jump_rel` processor adds operand to PC.

Names `jump_abs` and `jump_rel` are made up.

In DLX the following CTIs use PC-relative addresses: `beqz` `bneq`, `j`, `jal`, `bfpt`, and `bfpf`

In DLX there are no CTIs that use absolute addressing.

Destination Address Examples

Jump and link SPARC instruction.

First two operands specify address, a register plus immediate.

Last operand is register to save PC in.

Register `g0` is a dummy register.

```
 jmpl %l0+0,%o7     ! Call address in %l0, PC saved in %o7
 nop
 ...

SUBROUTINE:
 add %l1, %l2, %l3  ! l3 = l2 + l1 (Note that l3 is desto.)
 jmpl %o7+0, %g0    ! Return using address in o7. Save PC in dummy reg.
 nop
```


Branch Displacement Size

Branch distances on DLX running SPEC92 programs.

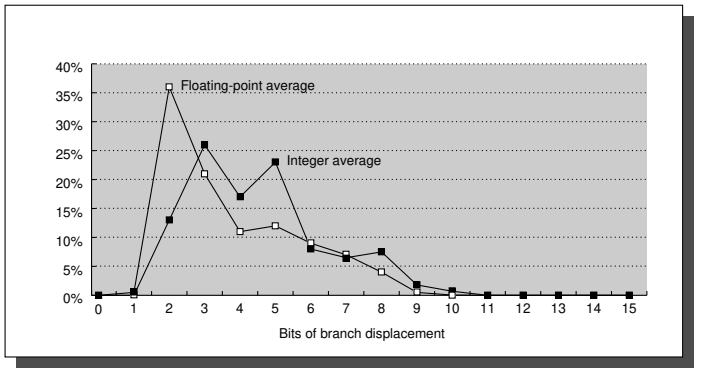

FIGURE 2.13 Branch distances in terms of number of instructions between the target and the branch instruction.

In DLX branch displacement limited to 16 bits.

Branch Conditions

How branch condition might be specified:

- Test value of general-purpose register (GPR).
- Test value of special-purpose *condition code register* (CCR).
- Test value of special-purpose *loop-counter register*.
- 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*.

A typical ISA would use one or two of these methods.

Branch Condition Example (DLX)

DLX uses two methods: GPR and CCR:

Integer compare instructions (*e.g.*, `slt`) write result comparison into any GPR ...
... where it is read by a branch.

Floating-point compare instructions (*e.g.*, `gtd`) write result into special *FP status* (condition-code) *register*.

```
 slt  r1, r2, r3  ! Compare r2 and r3, set r1=1 if r2 < r3.
 beqz r1, TARGET1 ! Branch if r1 = 0. (r2 >= r3).
 gtd  f0, f2      ! Compare f0 and f2, set FP CCR true if f0 > f2
 bfpt TARGET1     ! Branch if FP CCR is true.
 ....
TARGET1:
```

DLX Mnemonics:

`slt`: Set (destination register if op1) less than (op2).

`beqz`: Branch if (op1) equal to zero.

`gtd`: Greater-than double: Set FP CCR if op1 > op2.

Branch Condition Example (Sparc)

Sparc uses an *integer condition codes* register (actually part of a larger status register).

ALU instructions can optionally set condition codes.

Register has several 1-bit fields which describe outcome: *negative*, *zero*, *overflow*, and *carry*.

Branch instruction can test for many combinations (*e.g.*, not negative and not zero, not negative, negative).

```
 subcc %l1, %l2, %g0  ! %g0 = %l1 - %l2, discard difference but set codes.
 sub   %l3, %l4, %l5  ! %l5 = %l3 - %l4. Doesn't set codes.
 bg    TARGET         ! Branch if subcc result pos.. (g in bg is for >0.)
 add   %l6, %l7, %l8  ! Branch delay slot: add always executed.
 ....
TARGET:
```


Tradeoffs between Methods to Specify Branch Condition

Factors

For compact code and programmer convenience:

⇒Comparison in branch instruction.

For fast implementation:

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

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

⇒Test loop counter.

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.

04-17

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.

04-18

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.

04-19

Implementation of Call and Return Steps

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.

04-20

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 TARGET        ! Jump to TARGET.
 2  add R1,R1,R1
 3  add R2,R2,R2
 4  add R3,R3,R3
 5  add R4,R4,R4
 ...
TARGET:
```

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.

CTI Variations

Branch Instructions with Prediction Hints

Programmer indicates (hints) whether ...
... branch is expected to be taken most of the time or ...
... not taken most of the time.

Hint only affects execution speed, not what the program does.

If programmer correct, execution may be faster ...
... if programmer wrong execution is correct but slower.

A low-cost implementation might ignore the hint.

Example: Prediction hints added to DLX.

Hint indicated by following instruction mnemonic with ...
... `,pt` (predict taken) or ...
... `,pn` (predict not taken).

```
beqz,pn  r1, LINE1   ! Programmer expects r1 to be nonzero most of the time.
beqz,pt  r2, LINE2   ! Programmer expects r2 zero most of the time.
add      r3, r4, r5
LINE1:
addi     r3, r3, #1
LINE2:
add      r3, r6, r7
```


Predicated Execution

Non-CTI instructions that only write results if some condition true.

If condition false instruction may still be executed ...
... though results not written.

When predicated execution used fewer branches needed ...
... though total number of instructions may be greater.

Because branches can slow implementations ...
... execution can be faster with predication even though more instructions executed.

Example: Add predication to DLX.

Predicate indicated in parenthesis before mnemonic.

(⟨**rp**⟩) means write results if ⟨**rp**⟩ nonzero ...
... (!⟨**rp**⟩) means write results if ⟨**rp**⟩ is zero.

```
! High level code: if( r1 == 0 ) r2 = r2 + r5 else r2 = r2 + r4;
! Without predicated execution.
beqz  r1, LINE1
add   r2, r2, r4
j     LINE2
LINE1:
add   r2, r2, r5
LINE2:

! With predicated execution.
(r1)  add   r2, r2, r4  ! Execute if r1 nonzero.
(!r1) add   r2, r2, r5  ! Execute if r1 zero.
```

Is it a coincidence that the destination and first source operands were the same in the `adds`?

ISA Classification

Consider the way cars are classified:

Criteria: performance, comfort, off-road capabilities.

Classifications: sports cars, luxury cars, SUVs.

A classification may be based on one or more criteria ...
... for example sports cars are high performance and comfortable (excluding ride).

Processors are also classified based on multiple criteria.

Processor Classification: Three Criteria

Address space size: 32-bit, 64-bit, etc.

Register use by instructions: Load/store, memory/memory, stack, etc.

Instruction format: Fixed length, variable length, *bundled* (VLIW).

Criterion: Address Space Size

Describes address space size.

n-bit processor indicates...
... n-bit address space and fast n-bit integer instructions for pointer arithmetic.

Terminology widely adopted but not universal.

Temptation to inflate n, *e.g.*, 128-bit processor (may refer to bus width).

Criterion: Register Use

Describes how instructions can use registers and addressing modes.

Load/Store, General Purpose Register

Memory only accessed by special load and store instructions.

Few special purpose registers.

Arithmetic and logic instructions use register or immediates.

Examples include most ISAs developed in 1980s and 1990s.

Memory/Memory, General Purpose Register

Memory can be read and written by arithmetic and logic instructions.

(Arithmetic and logic instructions can also use register and immediate operands.

Few special purpose registers.

Characteristic of older ISAs.

Criterion: Register Use

Accumulator

- Arithmetic instructions implicitly use accumulator.
- May include other special-purpose registers.

Stack

- Instructions refer to stack.
- Used in several old and new machines.

04-29

Criterion: Instruction Format

Describes allowed instruction sizes and something called bundling.

Fixed Size

- All instructions same size. (Eg/, 32 bits).
- All instructions do about same amount of work.
- Eases implementation.
- Characteristic of newer ISAs.

Variable Size

- Instruction size depends on instruction and operands.
- Potential for compact code.
- Implementation difficult.
- Characteristic of older and special-purpose ISAs.

04-30

Criterion: Instruction Format

Bundled (VLIW,EPIC).

- Instructions grouped into bundles, to be executed as a group.
- Bundles also contain information to speed execution.
- Used in new IA-64 (and older experimental machines).

04-31

ISA Classifications

Common Classifications and Status

- Current General Purpose: RISC (Reduced Instruction Set Computing)
- Former and Legacy (still used) General Purpose: CISC (Complex Instruction Set Computing)
- Future General Purpose?: VLIW (Very Long Instruction Word)
- Embedded (built in): Stack
- Past General Purpose: Accumulator

04-32

CISC (Complex Instruction Set Computing)

Criteria

Register Use: Memory/Memory, General Purpose Register

Instruction Format: Variable Instruction Size

Characteristics

Programmer friendly, lots of instruction choices.

Programmer un-friendly, may contain many almost-useful special purpose instructions.

Average code size.

Implementation cost inflated by ISA complexity.

From mid 1980s to late 1990s ISA complexity slowed implementations but ...
... *fin-de-siècle*[1] CISC processors nearly as fast as RISC processors.

Examples: VAX, 68000, maybe IA-32 (popularly known as 80x86).

[1] End of century, 20th century here.

RISC (Reduced Instruction Set Computing)

Criteria

Register Use: Load/Store, General Purpose Register

Instruction Format: Fixed Instruction Sizes

Characteristics

Work performed by each instruction balanced to ease implementation.

Few special purpose, rarely used instructions.

Code size large. (Because of fixed instruction size.)

RISC Characteristics, continued.

Instructions designed to speed implementation ...
... though sometimes design biased towards contemporary technology ...
... ignoring future technology that might be used in later implementations.

Some implementations rely on compiler to arrange (schedule) instruction for fast execution (discussed later).

Most major ISAs since 1985 are RISC types.

Examples: SPARC V9, Alpha, MIPS, Precision Architecture.

VLIW (Very Long Instruction Word)

Special case: EPIC (Explicitly Parallel Instruction Computing)

Criteria

Instruction Format: Bundled.

Characteristics

Extra information in bundles allows higher speed implementations.

Examples: TI 320C6 (meant for signal processing), the new Intel IA-64 (its introduction is considered a major event in computer architecture).

Discussed further later in the course.

04-37

Stack

Criteria

Register Use: Stack

Instruction Format: Variable Width

Characteristics

Small code size. Important for *embedded* [in cell phone, VCR, automobile engine, etc.] processors.

Limited speed because stack limits execution overlap opportunities.

Examples: Burroughs Corp. machines (1961-1977), INMOS Transputer (1983-).

EE 4720 Lecture Transparency. Formatted 10:24, 13 September 1999 from lsli04.

04-37