

Preview

What RISC ISAs look like today

- the original model
- the new instructions
 - what they do
 - why they're used

64b architectures

- issues with backwards compatibility to old “word” sizes

RISC vs. CISC

RISC Instruction Set Architecture

Simple instruction set

- opcodes are primitive operations use instructions in combination for more complex operations
 - data transfer, arithmetic/logical, control
- few & simple addressing modes (register, immediate, displacement/indexed)

Load/store architecture

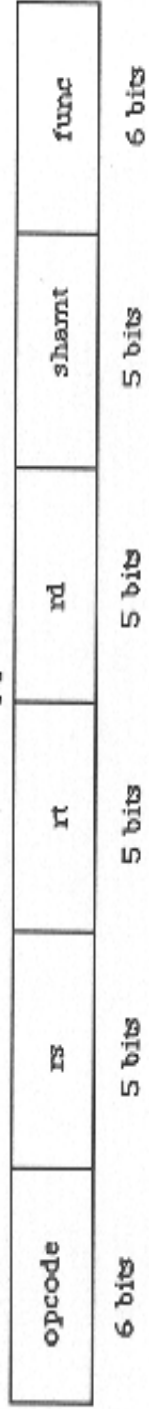
- load/store values from/to memory with explicit instructions
- compute in general purpose registers

Easily decoded instruction set

- fixed length instructions
- few instruction formats, many fields in common, a field in many formats is in the same bit location in all of them

R{2,3}000 Instruction Formats

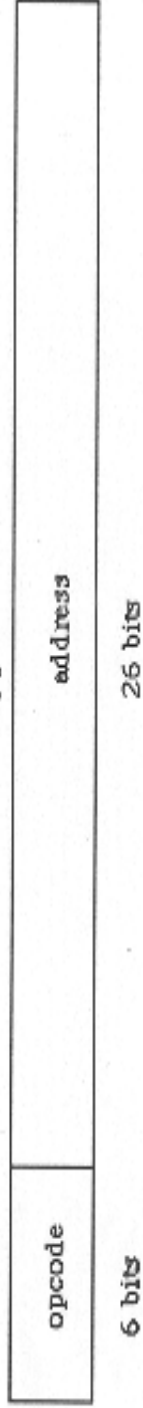
R-type



I-type



J-type



RISC Instruction Set Architecture

Designed for pipeline & superscalar efficiency

- simple instructions do almost the same amount of work
- instructions with simple & regular formatting can be decoded in parallel

Still some issues

- condition codes vs. condition registers
- GPR organization: register windows vs. flat register file
- sizes of immediates
- support for integer divide & FP operations
- how CISCy do we get?

32 reg GPR

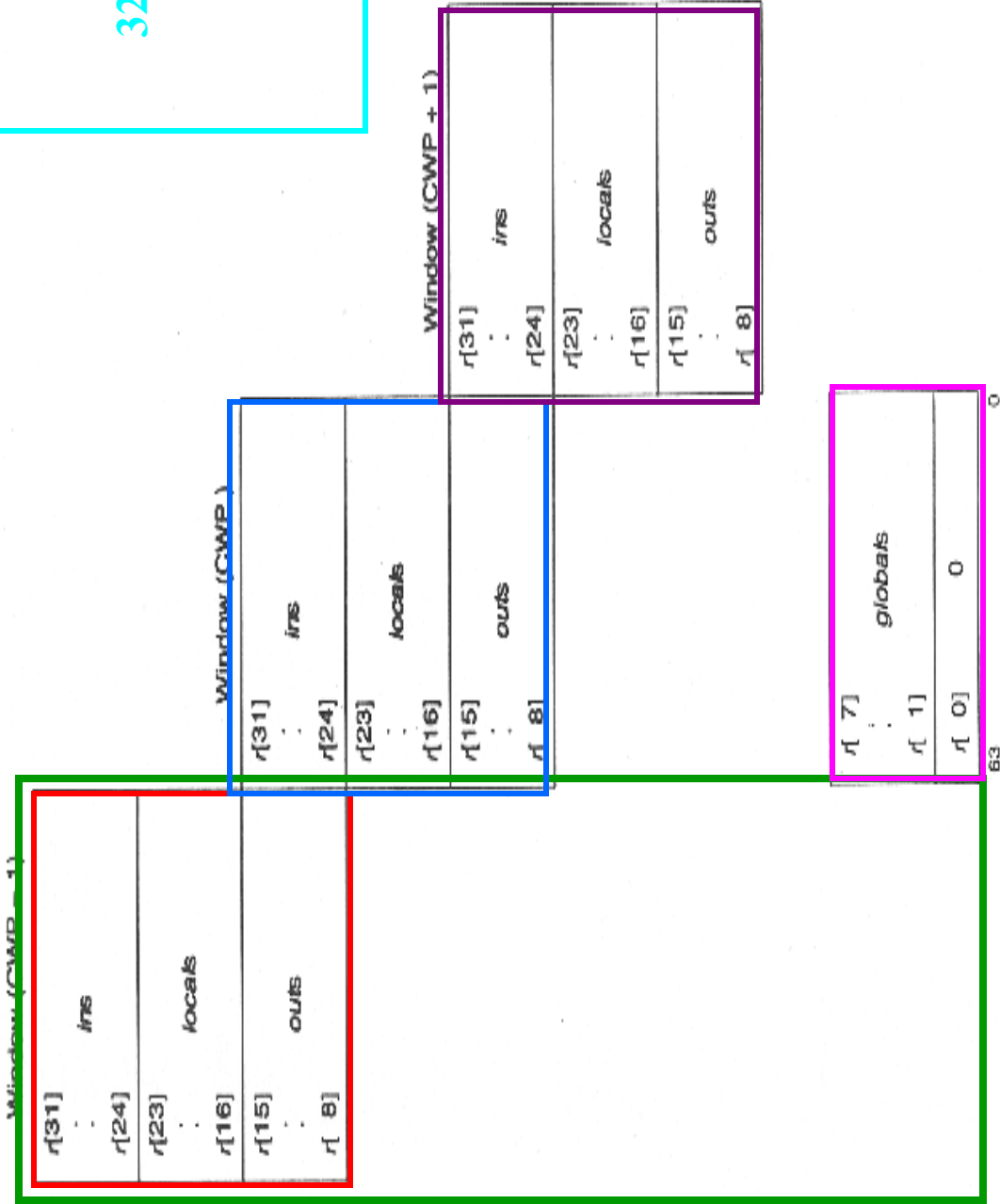


Figure 2—Three Overlapping Windows and the Eight Global Registers

Today's RISC Architectures

64-bit architectures

- 64b registers & datapath
- 64b addresses (used in loads, stores, indirect branches)
- linear address space (no segmentation)

New instructions

- better performance
- impulse to CISCyness

Backwards Compatibility

Problem: have to be able to execute the “old” 32b codes, with 32b operands

Some general approaches

- start all over: design a new 64-bit instruction set (Alpha)
- 2 instruction subsets, mode for each (MIPS-III)
 - 32b instructions from previous architecture
 - new 64b instructions: ld/st, arithmetic, shift, conditional branch
 - illegal-instruction trap on 64b instructions in 32 bit mode

Backwards Compatibility

ld/st

- datapath is 64b; therefore manipulate 64b values in 1 instruction
- when loading 32b data in 64b mode, sign extend the value
- when loading 32b data in 32b mode, zero extend the value for backwards compatibility to 32b binaries

shift right

- specify operand width in instruction
so can sign/zero extend from correct bit (either 31 or 63)

Backwards Compatibility

Handling conditions

- condition registers
 - still use the GPRs
 - separate 64b/32b integer add & subtract instructions
- separate 64b & 32b integer condition codes
 - 1 set of arithmetic instructions sets them both
 - conditional branches (positive/negative or 0/not 0) & overflow instructions (overflow/not overflow)
test a specific CC set

New Instructions

Purpose:

- for better performance
- to better match changes in process technology (e.g., a bigger discrepancy between CPU & memory speeds)
- to better match new microarchitectures (e.g., deeper pipelines)
- to take advantage of new compiler optimizations (e.g., statically determining which array accesses will hit or miss in the L1 cache)
- to support new, compute-intensive applications (e.g., multimedia)
- impulse to CISCyness (they think it's for better performance) (e.g., multiple loop-related operations)

New Instructions

predicated execution (wait for branch prediction)

data prefetching (wait for memory hierarchy)

loop support

- combine simple instructions that handle common programming idioms
 - scaled add/subtract/compare
 - branch on count
- are these a good idea?

New Instructions

multimedia instructions

- targeted for graphics, audio and video data
- partitioned arithmetic
 - 64b wasted on common data
 - arithmetic on two 32b, four 16b or eight 8b data
 - example operations: add, subtract, multiply, compare
- special instructions that manipulate < 64b data:
 - complex operations that are executed frequently
 - expand, pack, partial store
 - pixel distance instruction for motion estimation, edges on convolution
- examples: MMX, VIS

New Instructions

multimedia instructions

- ramifications on the architecture
 - new instructions
 - new formats
- ramifications on the implementation
 - part of FP hardware
 - already handles multicycle operations
 - “register partitioning” already done to implement single-precision arithmetic
 - integer pipeline needed to execute integer instructions
- surprisingly small proportion of die

New Instructions

multimedia instructions

- ramifications on the programming:
 - call assembly language library routines
 - write assembly language code
- ramifications on performance
 - ex: VIS pixel distance instruction eliminates ~50 RISC instructions
 - ex: 5.5X speedup to compute absolute sum of differences on 16x16-pixel image blocks

Bottom line:

- + increase performance on an important compute-intensive application that uses MM instructions a lot
- + with a small hardware cost
- but a large programming effort

CISC Instruction Set Architecture, aka x86

Complex instruction set

- more complex opcodes
 - ex: transcendental functions, string manipulation
 - ex: different opcodes for intra/inter segment transfers of control
- more addressing modes
 - 7 data memory addressing modes + multiple displacement sizes
- restrictions on what registers can be used with what modes

Register-memory architecture

- operands in computation instructions can reside in memory

CISC Instruction Set Architecture, aka x86

Complex instruction encoding

- variable length instructions
(different numbers of operands, different operand sizes, prefixes for machine word size, postbytes to specify addressing modes, etc.)
- lots of formats, tight encoding

More complex register design

- special-purpose registers

More complex memory management

- segmentation with paging

Backwards Compatibility is Harder with CISCs

Must support:

- registers with special functions
 - when it is recognized that register speed, not how a register is used, is what matters
- multiple instruction formats & data sizes
 - when have to translate to RISClike instructions to easily pipeline
- special categories of instructions
 - even though they are no longer used
- real addressing, segmentation without paging, segmentation with paging
 - when addressing range is obtained with address size
- stack model for floating point
 - when most programs use arbitrary memory operand addresses

RISC vs. CISC

Which is best?

RISC vs. CISC

Advantage of RISC depends on (among other things):

- chip technology
- processor complexity

Pre-1990: chip density was **low** & processor implementations were **simple**

- single-chip RISC CPUs (1986) & on-chip caches
- instruction decoding “large” part of execution cycle for CISCs

Post-1990: chip density is **high** & processor implementations are **complex**

- both RISC & CISC implementations fit on a chip with multiple big caches
- instruction decoding smaller time component:
 - multiple-instruction issue
 - out-of-order execution
 - speculative execution & sophisticated branch prediction
 - multithreading

Other Important Factors

Clock rate

- dense process technology (currently 90nm)
- superpipelining (all pipelines manipulate primitive instructions)

Compiler technology

- architecture features that help compilation
 - orthogonal architecture, simple architecture
 - primitive operations
 - lots of general purpose registers
 - operations without side effects

Ability of the design team

New/old architecture

- application base

\$\$\$

Wrap-up

What RISC ISAs look like today

- the original model
- the new instructions
 - what they do
 - why they're used

64b architectures

- issues with backwards compatibility to old “word” sizes
(makes you realize how pervasive the “word” size is – it’s not just the addressable memory space)

RISC vs. CISC is not the simplistic debate it used to be