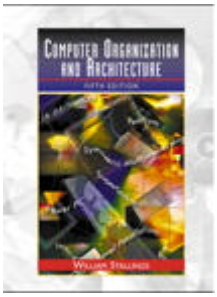
Campus de Gualtar
4710-057 Braga

UNIVERSIDADE DO MINHO
ESCOLA DE ENGENHARIA

Departamento de
Informática



Computer Organization and Architecture

5th Edition, 2000

by William Stallings

Table of Contents

III. THE CENTRAL PROCESSING UNIT.

8. ...
9. ...
10. ...
11. ...
12. Reduced Instruction Set Computers (RISCs). (5-Jan-01)

Introduction

- RISC is one of the few true innovations in computer organization and architecture in the last 50 years of computing.
- Key elements common to most designs:
    - A limited and simple instruction set
    - A large number of general purpose registers, or the use of compiler technology to optimize register usage
    - An emphasis on optimizing the instruction pipeline

Instruction Execution Characteristics (12.1)

- Overview
    - Semantic Gap - the difference between the operations provided in high-level languages and those provided in computer architecture
    - Symptoms of the semantic gap:
        - Execution inefficiency
        - Excessive machine program size
        - Compiler complexity
    - New designs had features trying to close gap:
        - Large instruction sets
        - Dozens of addressing modes
        - Various HLL statements in hardware
    - Intent of these designs:
        - Make compiler-writing easier
        - Improve execution efficiency by implementing complex sequences of operations in microcode
        - Provide support for even more complex and sophisticated HLL's
    - Concurrently, studies of the machine instructions generated by HLL programs
        - Looked at the characteristics and patterns of execution of such instructions
        - Results lead to using simpler architectures to support HLL's, instead of more complex
    - To understand the reasoning of the RISC advocates, we look at study results on 3 main aspects of computation:
        - Operations performed - the functions to be performed by the CPU and its interaction with memory.
        - Operands used - types of operands and their frequency of use. Determine memory organization and addressing modes.
        - Execution Sequencing - determines the control and pipeline organization.
    - Study results are based on dynamic measurements (during program execution), so that we can see effect on performance

- Operations
    - Simple counting of statement frequency indicates that assignment (data movement) predominates, followed by selection/iteration.
    - Weighted studies show that call/return actually accounts for the most work
    - Target architectural organization to support these operations well
    - Patterson study also looked at dynamic frequency of occurrence of classes of variables. Results showed a preponderance of references to highly localized scalars:
        - Majority of references are to simple scalars
        - Over 80% of scalars were local variables
        - References to arrays/structures require a previous ref to their index or pointer, which is usually a local scalar

- Operands
    - Another study found that each instruction (DEC-10 in this case) references 0.5 operands in memory and 1.4 registers.
    - Implications:
        - Need for fast operand accessing
        - Need for optimized mechanisms for storing and accessing local scalar variables

- Execution Sequencing
    - Subroutine calls are the time-consuming operation in HLL's
    - Minimize their impact by
        - Streamlining the parameter passing
        - Efficient access to local variables
        - Support nested subroutine invocation
    - Statistics
        - 98% of dynamically called procedures passed fewer than 6 parameters
        - 92% use less than 6 local scalar variables
        - Rare to have long sequences of subroutine calls followed by returns (e.g., a recursive sorting algorithm)
        - Depth of nesting was typically rather low

- Implications
    - Reducing the semantic gap through complex architectures may not be the most efficient use of system hardware
    - Optimize machine design based on the most time-consuming tasks of typical HLL programs
    - Use large numbers of registers
        - Reduce memory reference by keeping variables close to CPU (more register refs instead)
        - Streamlines instruction set by making memory interactions primarily loads and stores
    - Pipeline design
        - Minimize impact of conditional branches
    - Simplify instruction set rather than make it more complex

Large Register Files (12.2)

- How can we make programs use registers more often?
    - Software - optimizing compilers
        - Compiler attempts to allocate registers to those variables that will be used most in a given time period
        - Requires sophisticated program-analysis algorithms
    - Hardware
        - Make more registers available, so that they'll be used more often by ordinary compilers
        - Pioneered at Berkeley by first commercial RISC product, the Pyramid

- Register Windows
    - Naively adding registers will not effectively reduce need to access memory
        - Since most operand references are to local scalars, obviously store them in registers, with maybe a few for global variables
        - Problem: Definition of local changes with each procedure call and return (which happen a lot!)
            - On call, locals must be moved from registers to memory to make room for called subroutine
            - Parameters must be passed
            - On return, parent variables must move back to registers
    - Remember study results:
        - A typical procedure uses only a few passed parameters and local variables
        - The depth of procedure activation fluctuates within a relatively narrow range
    - So:
        - Use multiple small sets of registers, each assigned to a different procedure
        - A procedure call automatically switches the CPU to use a different fixed-size window of registers (no saving registers in memory!)
        - Windows for adjacent procedures are overlapped to allow parameter passing
    - Since there is a limit to number of windows, we use a circular buffer of windows
        - Only hold the most recent procedure activations in register windows
        - Older activations must be saved to memory and later restored
            - An N-window register file can hold only N-1 procedure activations
            - One study found that with 8 windows, a save or restore is needed on only 1% of calls or returns

- Global variables
    - Could just use memory, but would be inefficient for frequently used globals
    - Incorporate a set of global registers in the CPU. Then, the registers available to a procedure would be split:
        - some would be the global registers
        - the rest would be in the current window.
    - Hardware would have to also:
        - decide which globals to put in registers
        - accommodate the split in register addressing

- Large Register File vs. Cache
    - Why not just build a big cache? Answer not clear cut
        - Window holds all local scalars
        - Cache holds selection of recently used data
        - Cache can be forced to hold data it never uses (due to block transfers)
        - Current data in cache can be swapped out due to accessing scheme used
        - Cache can easily store global and local variables
        - Addressing registers is cleaner and faster

Compiler-Based Register Optimization (12.3)

- In this case, the number of registers is small compared to the large register file implementation
- The compiler is responsible for managing the use of the registers
- Compiler must map the current and projected use of variables onto the available registers
    - Similar to a graph coloring problem
    - Form a graph with variables as nodes and edges that link variables that are active at the same time
    - Color the graph with as many colors as you have registers
    - Variables not colored must be stored in memory

Reduced Instruction Set Architecture (12.4)

- Why CISC?
    - CISC trends to richer instruction sets
        - More instructions
        - More complex instructions
    - Reasons
        - To simplify compilers
        - To improve performance
- Are compilers simplified?
    - Assertion: If there are machine instructions that resemble HLL statements, compiler construction is simpler
    - Counter-arguments:
        - Complex machine instructions are often hard to exploit because the compiler must find those cases that fit the construct
        - Other compiler goals
            - Minimizing code size
            - Reducing instruction execution count
            - Enhancing pipelining
              are more difficult with a complex instruction set
        - Studies show that most instructions actually produced by CISC compilers are the relatively simple ones
- Is performance improved?
    - Assertion: Programs will be smaller and they will execute faster
        - Smaller programs save memory
        - Smaller programs have fewer instructions, requiring less instruction fetching
        - Smaller programs occupy fewer pages in a paged environment, so have fewer page faults

- Counter-
  - Inexpensive memory makes memory savings less compelling
- used may not be smaller

  Opcodes require more bits
- to register identifiers (which are the usual case for RISC)
- operations, so even the more often-
- The speedup for complex instructions may be mostly due to their implementation as simpler

  that the CISC designer must decide a priori which instructions to speed up in this way)
-
  - One instruction per cycle
    -
      registers, perform and ALU operation, and store the result in a register

      RISC machine instructions should be no more complicated than, and execute
      s fast as microinstructions on a CISC machine

      No microcoding needed, and simple instructions will execute faster than their
  - Register- -register operations
    - ions access memory

      Simplifies instruction set and control unit
      -
      - Ex. VAX has 25 different ADD instructions

      Encourages optimization of register use
  -
    - Almost all instructions use simple register addressing
    -
    - More complex addressing is implemented in software from the simpler ones

      Further simplifies instruction set and control unit
  -
    - Only a few formats are used
      -
    - Instruction length is fixed and aligned on word boundaries

      Optimizes instruction fetching
      -
    - Field locations (especially the opcode) are fixed

      Allows simul
- Potential benefits

  More effective optimizing compilers
  -
  - Instruction pipelining can be applied more effectively with a reduced instruction set
  -
    - They are checked between rudimentary operations

      No need for complex instruction restarting mechanisms
-

- Requires less "real estate" for control unit (6% in RISC I vs. about 50% for CISC microcode store)
- Less design and implementation time

RISC Pipelining (12.5)

- The simplified structure of RISC instructions allows us to reconsider pipelining
  - Most instructions are register-to-register, so an instruction cycle has 2 phases
    - I: Instruction Fetch
    - E: Execute (an ALU operation w/ register input and output)
  - For load and store operations, 3 phases are needed
    - I: Instruction fetch
    - E: Execute (actually memory address calculation)
    - D: Memory (register-to-memory or memory-to-register)
- Since the E phase usually involves an ALU operation, it may be longer than the other phases. In this case, we can divide it into 2 sub phases:
  - E1: Register file read
  - E2: ALU operation and register write

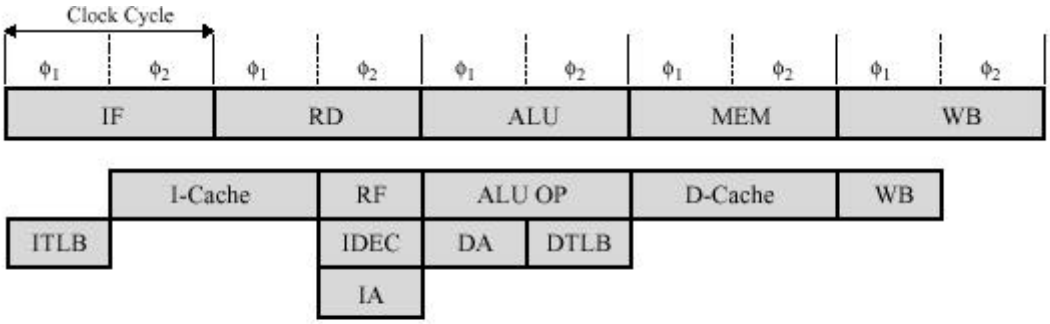
Optimization of Pipelining

- Delayed Branch
  - We've seen that data and branch dependencies reduce the overall execution rate in the pipeline
  - Delayed branch makes use of a branch that does not take effect until after the execution of the following instruction
    - Note that the branch "takes effect" during its execution phase
    - So, the instruction location immediately following the branch is called the delay slot
    - This is because the instruction fetching order is not affected by the branch until the instruction after the delay slot
    - Rather than wasting an instruction with a NOOP, it may be possible to move the instruction preceding the branch to the delay slot, while still retaining the original program semantics.
- Conditional branches
  - If the instruction immediately preceding the branch cannot alter the branch condition, this optimization can be applied
  - Otherwise a NOOP delay is still required.
  - Experience with both the Berkeley RISC and IBM 801 systems shows that a majority of conditional branches can be optimized this way.
- Delayed Load
  - On load instructions, the register to be loaded is locked by the processor
  - The processor continues execution of the instruction stream until reaching an instruction needing a locked register
  - It then idles until the load is complete
  - If load takes a specific maximum number of clock cycles, it may be possible to rearrange instructions to avoid the idle.

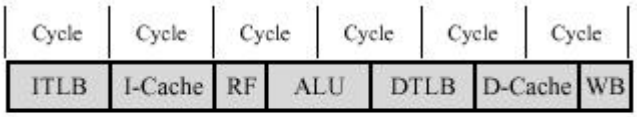
Superpipelining

- A superpipelined architecture is one that makes use of more, and finer-grained, pipeline stages.

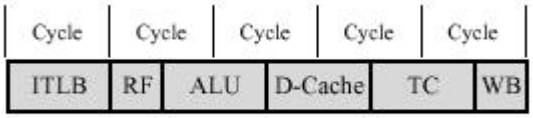
- The MIPS R3000 is an example of superpipelining
  - All instructions follow the same sequence of 5 pipeline stages (the 60-ns clock cycle is divided into two 30-ns phases)
  - But the activities needed for each stage may occur in parallel, and may not use an entire stage
- Essentially then, we can break up the external instruction and data cache operations, and the ALU operations, into 2 phases



(a) Detailed R3000 pipeline



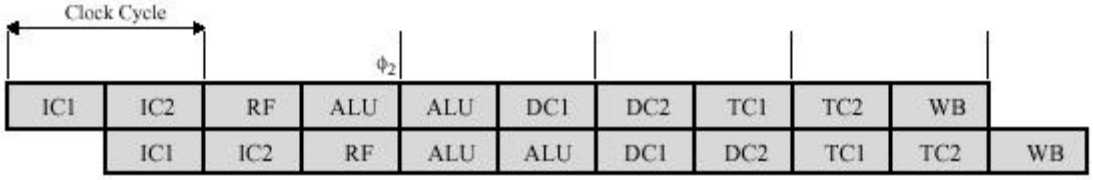
(b) Modified R3000 pipeline with reduced latencies



(c) Optimized R3000 pipeline with parallel TLB and cache accesses

IF = Instruction fetch
RD = Read
MEM = Memory access
WB = Write back
I-Cache = Instruction cache access
RF = Fetch operand from register
D-Cache = Data cache access
ITLB = Instruction address translation
IDEC = Instruction decode
IA = Compute instruction address
DA = Calculate data virtual address
DTLB = Data address translation
TC = Data cache tag check

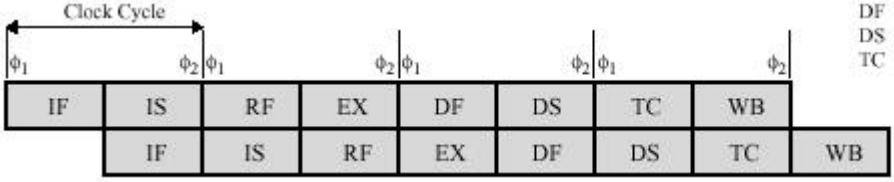
- In general:
  - In a superpipelined system existing hardware is used several times per cycle by inserting pipeline registers to split up each pipe stage
  - Each superpipeline stage operates at a multiple of the base clock frequency
  - The multiple depends on the degree of superpipelining (the number of phases into which each stage is split)
- The MIPS R4000 (which has improvements over the R3000 of the previous slide) is an example of superpipelining of degree 2 (see section 12.6 for details).



(a) Superpipelined implmentation of the optimized R3000 pipeline

IF = Instruction fetch first half
IS = Instruction fetch second half
RF = Fetch operands from register
EX = Instruction execute
IC = Instruction cache
DC = Data cache
DF = Data cache first half
DS = Data cache second half
TC = Tag check

(b) R4000 pipeline



The RISC vs. CISC Controversy (12.8)

- In spite of the apparent advantages of RISC, it is still an open question whether the RISC approach is demonstrably better.
- Studies to compare RISC to CISC are hampered by several problems (as of the textbook writing):
    - There is no pair of RISC and CISC machines that are closely comparable
    - No definitive set of test programs exist.
    - It is difficult to sort out hardware effects from effects due to skill in compiler writing.
- Most of the comparative analysis on RISC has been done on “toy” machines, rather than commercial products.
- Most commercially available “RISC” machines possess a mixture of RISC and CISC characteristics.
- The controversy has died down to a great extent
    - As chip densities and speeds increase, RISC systems have become more complex
    - To improve performance, CISC systems have increased their number of general-purpose registers and increased emphasis on instruction pipeline design.