# Computer Architecture

Homework 10

**Theme: RISC**

1. **In a tabular form, explain the differences between RISC and CISC architectures.**

|  |  |  |
| --- | --- | --- |
| Feature | RISC | CISC |
| 1. Number of instructions in the instruction set | Small | Large |
| 1. Complexity of instructions | Simple | Complex |
| 1. Instruction formats | Simple |  |
| 1. Average size of Instruction | Small (usually 4 Bytes) | Large |
| 1. Variability of Instruction size | Fixed size and aligned on word boundaries | Variable size |
| 1. Addressing modes supported | small number, typically less than five | Multiple |
| 1. Addressing modes | Simple register addressing mode; others - displacement, PC-relative; Complex modes supported by software | Several complex modes supported |
| 1. Number of general purpose registers | Large | Small |
| 1. Number of machine cycles per instruction | One | Many |
| 1. Control memory size | Not applicable | Large |
| 1. Optimization of procedure calls | using register windows | using memory stack |
| 1. Operations | mostly register-to-register; only simple LOAD and STORE operations access memory locations | supports complex operations, for e.g. memory-to-memory and mixed register/memory operations |
| 1. Register usage | optimizes register usage such that frequently used operands are available in registers | leverages cache which dynamically updates it content based on instructions and data access |
| 1. Instruction pipelining | makes efficient use of pipelining leveraging simple instruction set | Difficult to optimize pipeline usage due to complex instructions |
| 1. Response to interrupts | More responsive - as interrupts are checked between rather elementary operations | Less responsive - as interrupts are restricted to instruction boundaries. To be more responsive, the architecture must define specific interruptible points and implement mechanisms for restarting an instruction |

1. **What is the rationale of using a register window system?**

Because most operand references in a program are to local scalars, the obvious approach is to store these in registers. The problem is that the definition of ***local*** changes with each procedure call and return. On every call, local variables must be saved from the registers into memory, so that the registers can be reused by the called program. Furthermore, parameters must be passed. On return, the variables of the parent program must be restored (loaded back into registers) and results must be passed back to the parent program.

The solution is based on two observations. First, a typical procedure employs only a few passed parameters and local variables. Second, the depth of procedure activation fluctuates within a relatively narrow range. To exploit these properties, multiple small sets of registers are used, each assigned to a different procedure. A procedure call automatically switches the processor to use a different fixed-size window of registers, rather than saving registers in memory. Windows for adjacent procedures are overlapped to allow parameter passing. At any time, only one window of registers is visible and is addressable as if it were the only set of registers (e.g., addresses 0 through N-1).

To handle any possible pattern of calls and returns, the number of register windows would have to be unbounded. Instead, the register windows can be used to hold the few most recent procedure activations. Older activations must be saved in memory and later restored when the nesting depth decreases. Thus, the actual organization of the register file is as a circular buffer of overlapping windows.

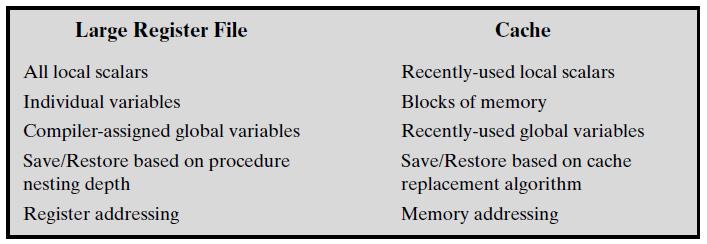
**Discuss different approaches of handling global variables.**

The register window approach is an efficient organization for storing local scalar variables in registers. However, this scheme does not address the need to store global variables, those accessed by more than one procedure.

There are two approaches to handle global variables.

1. First, variables declared as global in an HLL can be assigned memory locations by the compiler, and all machine instructions that reference these variables will use memory-reference operands. This is straightforward, from both the hardware and software (compiler) points of view. However, for frequently accessed global variables, this scheme is inefficient.
2. Second, incorporate a set of global registers in the processor. These registers would be fixed in number and available to all procedures. A unified numbering scheme can be used to simplify the instruction format. For example, references to registers 0 through 7 could refer to unique global registers, and references to registers 8 through 31 could be offset to refer to physical registers in the current window. There is an increased hardware burden to accommodate the split in register addressing. In addition, the compiler must decide which global variables should be assigned to registers.

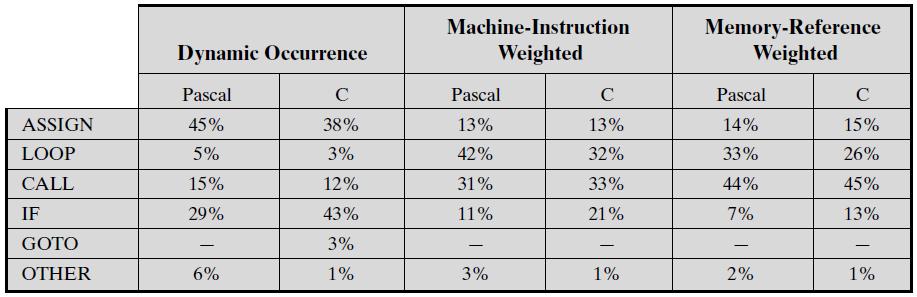
**Discuss the pros and cons of replacing the large register file with a cache.**



Above table compares characteristics of the two approaches to store the most frequently used data - large register file and cache. The window-based register file holds all the local scalar variables of the most recent N-1 procedure activations. The cache holds a selection of recently used scalar variables.

* The register file should save time, because all local scalar variables are retained. On the other hand, the cache may make more efficient use of space, because it is reacting to the situation dynamically.
* Caches generally treat all memory references alike, including instructions and other types of data. Thus, savings in these other areas are possible with a cache and not a register file.
* A register file may make inefficient use of space, because not all procedures will need the full window space allotted to them.
* Inefficiency with cache - data are read into the cache in blocks. Whereas the register file contains only those variables in use, the cache reads in a block of data, some or much of which will not be used.
* The cache is capable of handling global as well as local variables. There are usually many global scalars, but only a few of them are heavily used. A cache will dynamically discover these variables and hold them. If the window-based register file is supplemented with global registers, it too can hold some global scalars. However, it is difficult for a compiler to determine which globals will be heavily used.
* With the register file, movement of data between registers and memory is determined by the procedure nesting depth. Because this depth usually fluctuates within a narrow range, the use of memory is relatively infrequent. Most cache memories are set associative with a small set size. Thus, there is the danger that other data or instructions will overwrite frequently used variables.
* Based on the discussion so far, the choice between a large window-based register file and a cache is not clear-cut. There is one characteristic, however, in which the register approach is clearly superior and which suggests that a cache-based system will be noticeably slower. This distinction shows up in the amount of addressing overhead experienced by the two approaches.
* To reference a local scalar in a window based register file, a “virtual” register number and a window number are used. These can pass through a relatively simple decoder to select one of the physical registers. To reference a memory location in cache, a full-width memory address must be generated. The complexity of this operation depends on the addressing mode. Even if the cache is as fast as the register file, the access time will be considerably longer. Thus, from the point of view of performance, the window-based register file is superior for local scalars.

1. **In a tabular form give the percentage frequency of dynamic occurrence of the most frequently occurring instructions.**



The second and third columns in above table show the relative frequency of occurrence of various HLL instructions in a variety of programs; the data were obtained by observing the occurrences in running programs rather than just the number of times that statements occur in the source code. Hence these are dynamic frequency statistics.

To obtain the data in columns four and five (machine-instruction weighted), each value in the second and third columns is multiplied by the number of machine instructions produced by the compiler. These results are then normalized so that columns four and five show the relative frequency of occurrence, weighted by the number of machine instructions per HLL statement.

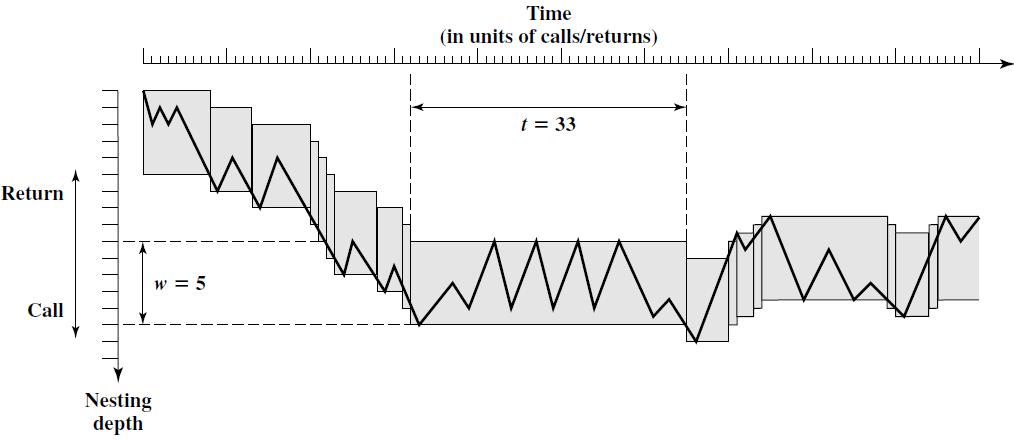
Similarly, the sixth and seventh columns are obtained by multiplying the frequency of occurrence of each statement type by the relative number of memory references caused by each statement.

The data in columns four through seven provide surrogate measures of the actual time spent executing the various statement types. The results suggest that the procedure call/return is the most time consuming operation in typical HLL programs.

This table indicates the relative significance of various statement types in an HLL, when that HLL is compiled for a typical contemporary instruction set architecture. Some other architecture could conceivably produce different results.

1. **Considering the call-return pattern in Figure 4.21, how many overflows and underflows (each of which causes a register save/restore) will occur with a window size of: 4, 7, and 16?**

Shown below is the figure 4.21 from book.



In a register window approach, with a window size of w, an overflow occurs if a new nested procedure call is encountered when the circular register window buffer is full. In such case, the content in registers assigned to oldest procedure call are stored in memory. This is referred as an overflow. In a similar manner, an underflow occurs when control returns to the procedure call whose registers' content is currently stored in memory and not available in circular register window buffer. In such scenario, data needs to be restored from memory to registers in circular window buffer.

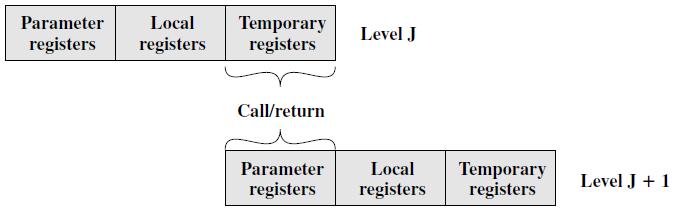
In the above figure, nested procedure call sequence is shown for the program.

With a window size of 4, there would be a total of 24 overflows and underflows.

With a window size of 7, there would be a total of 9 overflows and underflows.

With a window size of 16, there would be no overflows and no underflows, as the maximum nested procedure depth of program is 15. All procedure calls can be assigned a register window buffer.

1. **In the discussion of Figure 15.2, it was stated that only the first two portions of a window are saved or restored. Why is it not necessary to save the temporary registers?**



Shown in the picture above are overlapping register windows. At any time, only one window of registers is visible and is addressable as if it were the only set of registers (e.g., addresses 0 through N-1).

The window is divided into three fixed-size areas. **Parameter registers** hold parameters passed down from the procedure that called the current procedure and hold results to be passed back up. **Local registers** are used for local variables, as assigned by the compiler. **Temporary registers** are used to exchange parameters and results with the next lower level (procedure called by current procedure). The temporary registers at one level are physically the same as the parameter registers at the next lower level. This overlap permits parameters to be passed without the actual movement of data. Except for the overlap, the registers at two different levels are physically distinct. That is, the parameter and local registers at level J are disjoint from the local and temporary registers at level J+1.

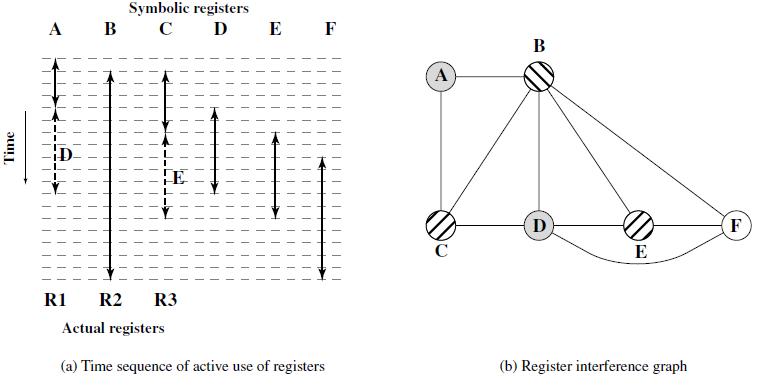
To handle any possible pattern of calls and returns, the number of register windows would have to be unbounded. Instead, the register windows can be used to hold the few most recent procedure activations. Older activations must be saved in memory and later restored when the nesting depth decreases. Thus, the actual organization of the register file is as a circular buffer of overlapping windows.

The temporary registers at level J are physically the same as the parameter registers at the next lower level J+1. The purpose of temporary registers at a level is to exchange parameters and results with the next lower level. These are shared physically between two adjacent levels and the content of registers is not required (temporary) by procedure call at level J, but is required (in the form of parameters) by procedure call at level J+1. Hence the data in these set of registers is not copied to/from memory during register window save/restore process.

1. **Using an example, explain the use and function of a register interference graph.**

If there are only a small number of registers available on a RISC machine, then optimized register usage is the responsibility of the compiler, which tries to keep the operands for as many computations as possible in registers rather than main memory, and to minimize load-and-store operations. A brief description of the approach is as follows. Each program quantity that is a candidate for residing in a register is assigned to a symbolic or virtual register. The compiler then maps the unlimited number of symbolic registers into a fixed number of real registers. Symbolic registers whose usage does not overlap can share the same real register. If, in a particular portion of the program, there are more quantities to deal with than real registers, then some of the quantities are assigned to memory locations. Load-and-store instructions are used to position quantities in registers temporarily for computational operations. The essence of the optimization task is to decide which quantities are to be assigned to registers at any given point in the program. The technique most commonly used in RISC compilers is known as graph coloring.

The graph coloring problem in brief is as follows. Given a graph consisting of nodes and edges, assign colors to nodes such that adjacent nodes have different colors, and do this in such a way as to minimize the number of different colors. This problem is adapted to the compiler problem in the following way. First, the program is analyzed to build a ***register interference graph***. The nodes of the graph are the symbolic registers. If two symbolic registers are “live” during the same program fragment, then they are joined by an edge to depict interference. An attempt is then made to color the graph with n colors, where n is the number of registers. Nodes that share the same color can be assigned to the same register. If this process does not fully succeed, then those nodes that cannot be colored must be placed in memory, and loads and stores must be used to make space for the affected quantities when they are needed.



Shown in the figure above is a simple example of the process. Assume a program with six symbolic registers to be compiled into three actual registers. The above figure (a) shows the time sequence of active use of each symbolic register. The dashed horizontal lines indicate successive instruction executions. Figure (b) shows the register interference graph (shading and cross-hatching are used instead of colors). A possible coloring with three colors is indicated. Because symbolic registers A and D do not interfere, the compile can assign both of these to physical register R1. Similarly, symbolic registers C and E can be assigned to register R3. One symbolic register, F, is left uncolored and must be dealt with using loads and stores.

1. **Using a timing diagram explain the use of delayed branches. Explain the use of reversed instructions.**

In a RISC system, data and branch dependencies reduce the maximum possible performance with pipelining. Delayed branches and reversed instructions, are approaches to resolve this issue and improve overall performance.

Delayed branch, a way of increasing the efficiency of the pipeline, makes use of a branch that does not take effect until after execution of the following instruction (hence the term delayed). The instruction location immediately following the branch is referred to as the delay slot.

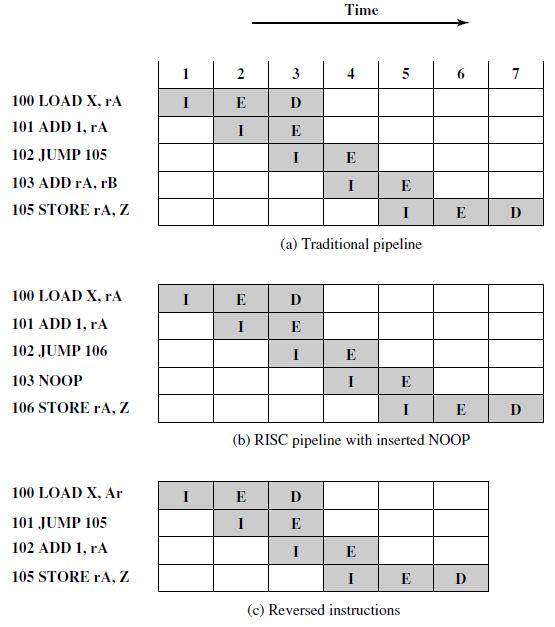


Figure (a) shows the traditional approach to pipelining. The JUMP instruction is fetched at time 3. At time 4, the JUMP instruction is executed at the same time that instruction 103 (ADD instruction) is fetched. Because a JUMP occurs, which updates the program counter, the pipeline must be cleared of instruction 103; at time 5, instruction 105, which is the target of the JUMP, is loaded.

Figure (b) shows the same pipeline handled by a typical RISC organization. The timing is the same. However, because of the insertion of the NOOP instruction, we do not need special circuitry to clear the pipeline; the NOOP simply executes with no effect.

Figure (c) shows the use of the reverse instructions approach. The JUMP instruction is fetched at time 2, before the ADD instruction, which is fetched at time 3. The ADD instruction is fetched before the execution of the JUMP instruction has a chance to alter the program counter. Therefore, during time 4, the ADD instruction is executed at the same time that instruction 105 is fetched. Thus, the original semantics of the program are retained but one less clock cycle is required for execution.

This interchange of instructions will work successfully for unconditional branches, calls, and returns. For conditional branches, this procedure cannot be blindly applied. If the condition that is tested for the branch can be altered by the immediately preceding instruction, then the compiler must refrain from doing the interchange and instead insert a NOOP. Otherwise, the compiler can seek to insert a useful instruction after the branch.