RISC

Introduction

Since the development of the stored program computer around 1950, there are few innovations in the area of computer organization and architecture. Some of the major developments are:

- **The Family Concept**: Introduced by IBM with its system/360 in 1964 followed by DEC, with its PDP-8. The family concept decouples the architecture of a machine from its implementation. A set of computers are offered, with different price/performance characteristics that present the same architecture to the user.

- **Microprogrammed Control Unit**: Suggested by Wilkes in 1951, and introduced by IBM on the S/360 line in 1964. Microprogramming eases the task of designing and implementing the control unit and provide support for the family concept.

- **Cache Memory**: First introduced commercially on IBM S/360 Model 85 in 1968. The insertion of this element into the memory hierarchy dramatically improves performance.

- **Pipelining**: A means of introducing parallelism into the essentially sequential nature of a machine instruction program. Examples are instruction pipelining and vector processing.

- **Multiple Processors**: This category covers a number of different organizations and objectives.

One of the most visual forms of evolution associated with computers is that of programming languages. Even more powerful and complex high level programming languages has been developed by the researcher and industry people.

The computer designers intend to reduce this gap and include large instruction set, more addressing mode and various HLL statements implemented in hardware. As a result the instruction set becomes complex. Such complex instruction sets are intended to-

- Ease the task of the compiler writer.
- Improve execution efficiency, because complex sequences of operations can be implemented in microcode.
- Provide support for even more complex and sophisticated HLLs.
To reduce the gap between HLL and the instruction set of computer architecture, the system becomes more and more complex and the resulted system is termed as **Complex Instruction Set Computer (CISC)**.

A number of studies have been done over the years to determine the characteristics and patterns of execution of machine instructions generated from HLL programs. The instruction execution characteristics involve the following aspects of computation:

- **Operation Performed**: These determine the functions to be performed by the processor and its interaction with memory.
- **Operand Used**: The types of operands and the frequency of their use determine the memory organization for storing them and the addressing modes for accessing them.
- **Execution sequencing**: This determines the control and pipeline organization.

**Operations:**

A variety of studies have been made to analyze the behavior of HLL programs. It is observed that

- Assignment statements predominate, suggesting that the simple movement of data is of high importance.
- There is also a presence of conditional statements (IF, Loop, etc.). These statements are implemented in machine language with some sort of compare and branch instruction. This suggests that the sequence control mechanism of the instruction set is important.

A variety of studies have analyzed the behavior of high level language program. The Table 8.1 includes key results, measuring the appearance of various statement types during execution which is carried out by different researchers.

**Table 8.1**: Relative Dynamic Frequency of High-Level Language operation

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign</td>
<td>74</td>
<td>67</td>
<td>45</td>
<td>38</td>
<td>42</td>
</tr>
<tr>
<td>Loop</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Call</td>
<td>1</td>
<td>3</td>
<td>15</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>IF</td>
<td>20</td>
<td>11</td>
<td>29</td>
<td>43</td>
<td>36</td>
</tr>
<tr>
<td>GOTO</td>
<td>20</td>
<td>9</td>
<td>--</td>
<td>3</td>
<td>--</td>
</tr>
<tr>
<td>Other</td>
<td>--</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>
These results are instructive to the machine instruction set designers, indicating which type of statements occur most often and therefore should be supported in an “optimal” fashion.

From these studies one can observe that though a complex and sophisticated instruction set is available in machine architecture, common programmer may not use those instructions frequently.

**Operands:**

Researches also studied the dynamic frequency of occurrence of classes of variables. The results showed that majority of references are single scalar variables. In addition references to arrays/structures required a previous reference to their index or pointer, which again is usually a local scalar. Thus there is a predominance of references to scalars, and these are highly localized.

It is also observed that operation on local variables is performed frequently and it requires a fast accessing of these operands. So, it suggests that a prime candidate for optimization is the mechanism for storing and accessing local scalar variables.

**Procedure Call:**

The procedure calls and returns are important aspects of HLL programs. Due to the concept of modular and functional programming, the call/return statements are becoming a predominate factor in HLL program.
It is known facts that call/return is a most time consuming and expensive statements. Because during call we have to restore the current state of the program which includes the contents of local variables that are present in general purpose registers. During return, we have to restore the original state of the program from where we start the procedure call.

Thus, it will be profitable to consider ways of implementing these operations efficiently. Two aspects are significant, the number of parameters and variables that a procedure deals with, and the depth of nesting.

**Implications:**

A number of groups have looked at these results and have concluded that the attempt to make the instruction set architecture close to HLL is not the most effective design strategy. Generalizing from the work of a number of researchers three elements emerge in the computer architecture.

- **First,** use a large number of registers or use a compiler to optimize register usage. This is intended to optimize operand referencing.
- **Second,** careful attention needs to be paid to the design of instruction pipelines. Because of the high proportion of conditional branch and procedure call instructions, a straight forward instruction pipeline will be inefficient. This manifests itself as a high proportion of instructions that are perfected but never executed.
- **Third,** a simplified (reduced) instruction set is indicated. It is observed that there is no point to design a complex instruction set which will lead to a complex architecture. Due to the fact, a most interesting and important processor architecture evolves which is termed as Reduced Instruction Set Computer (RISC) architecture.

Although RISC system have been defined and designed in a variety of ways by different groups, the key element shared by most design are these:

- A large number of general purpose registers, or the use of compiler technology to optimize register usage.
- A limited and simple instruction set.
- An emphasis on optimizing the instruction pipeline.

An analysis of the RISC architecture begins into focus many of the important issues in computer organization and architecture.

The comparison of RISC and non-RISC systems is given in the Table 8.2.
### Table 8.2: Characteristics of some CISCs, RISCs and Superscalar Processors

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>IBM 370/168</th>
<th>VAX 11/780</th>
<th>Intel 80486</th>
<th>SPARC</th>
<th>MIPS R4000</th>
<th>Power PC</th>
<th>Ultra SPARC</th>
<th>MIPS R10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Instructions</td>
<td>208</td>
<td>303</td>
<td>235</td>
<td>69</td>
<td>94</td>
<td>225</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Instruction Size (bytes)</td>
<td>2-6</td>
<td>2-57</td>
<td>1-11</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Addressing modes</td>
<td>4</td>
<td>22</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Number of general-purpose registers</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>40-520</td>
<td>32</td>
<td>32</td>
<td>40-520</td>
<td>32</td>
</tr>
<tr>
<td>Control Memory size (kbits)</td>
<td>420</td>
<td>480</td>
<td>246</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Cache size (kbits)</td>
<td>64</td>
<td>64</td>
<td>8</td>
<td>32</td>
<td>128</td>
<td>16-32</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

### Characteristics of Reduced Instruction Set Architecture:

Although a variety of different approaches to reduce Instruction set architecture have been taken, certain characteristics are common to all of them:

1. One instruction per cycle.
2. Register–to–register operations.
3. Simple addressing modes.
4. Simple instruction formats.

#### 1. One machine instruction per machine cycle:

A machine cycle is defined to be the time it takes to fetch two operands from registers, perform an ALU operation, and store the result in a register.
With simple, one-cycle instructions there is little or no need of microcode, the machine instructions can be hardwired. Hardware implementation of control unit executes faster than the microprogrammed control, because it is not necessary to access a microprogram control store during instruction execution.

2. Register –to– register operations

With register–to–register operation, a simple LOAD and STORE operation is required to access the memory, because most of the operation is register–to-register. Generally we do not have memory–to–memory and mixed register/memory operation.

Simple Addressing Modes

Almost all RISC instructions use simple register addressing. For memory access only, we may include some other addressing, such as displacement and PC-relative. Once the data are fetched inside the CPU, all instruction can be performed with simple register addressing.

Simple Instruction Format

Generally in most of the RISC machine, only one or few formats are used. Instruction length is fixed and aligned on word boundaries. Field locations, especially the opcode, are fixed. With fixed fields, opcode decoding and register operand accessing can occur simultaneously. Simplified formats simplify the control unit.

Design Issues of RISC

The use of a large register file:

For fast execution of instructions, it is desirable of quick access to operands.

There is large proportion of assignment statements in HLL programs, and many of these are of the simple form $A \leftarrow B$. Also there is significant number of operand accesses per HLL Statement.

Also it is observed that most of the accesses are local scalars. To get a fast response, we must have an easy access to these local scalars, and so the use of register storage is suggested.
Since registers are the fastest available storage devices, faster than both main memory and cache, so the uses of registers are preferable. The register file is physically small, and on the same chip as the ALU and Control Unit. A strategy is needed that will allow the most frequently accessed operands to be kept in registers and to minimize register-memory operations.

Two basic approaches are possible; one is based on software and the other on hardware.

- **The software approach** is to rely on the compiler to maximize register uses. The compiler will attempt to allocate registers to those variables that will be used the most in a given time period.
- **The hardware approach** is simply to use more registers so that more variables can be held in registers for longer period of time.
  In hardware approach, it uses the concept of register windows.

**Register Window:**

The use of a large set of registers should decrease the need to access memory. The design task is to organize the registers in such a way that this goal is realized.

Due to the use of the concept of modular programming, the present day programs are dominated by call/return statements. There are some local variables present in each function or procedure.

1. 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, the parameters must be passed.
2. 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.
3. There are also some global variables which are used by the module or procedure.

Thus the variables that are used in a program can be categorized as follows:

- **Global variables** : which is visible to all the procedures?
- Local variables : which is local to a procedure and it can be accessed inside the procedure only.
- **Passed parameters** : which are passed to a subroutine from the calling program. So, these are visible to both called and calling program.
- **Returned variable** : variable to transfer the results from called program to the calling program. These are also visible to both called and calling program.
From the studies it is observed that a typical procedure employs only a few passed parameters and local variables. Also the depth of procedure activation remains 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. The concept of overlapping register window is shown in the Figure 8.1.

![Figure 8.1: Overlapping register window](image)

At any time, only one window of registers is visible which corresponds to the currently executing procedure.

The register 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.
- 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 parameter to be passed without the actual movement of data.

To handle any possible pattern of calls and returns, the number of register windows would have to be unbounded. But we have a limited number of registers, it is not possible to provide unlimited amount of registers.
It is possible to hold the little most recent procedure activation in register windows.

Older activations must be saved in memory and later restored when the nesting depth decreases. It is observed that nesting depth is small in general.

The circular buffer of overlapping windows is shown in the Figure 8.2. The procedure call pattern is: A called B, B called C, C called D and D called E, with procedure E as active process.

The current window pointer (CWP) points to the window of currently active procedure.

The saved window pointer identifies the window most recently saved in memory.

As the nesting depth of procedure calls increases, there may not be sufficient register to accommodate the new procedure. In this case, the information of oldest procedure is stored back into memory and the saved window pointer keep tracks of the most recently saved window.

It is clear that N-Window register file can hold only N-1 procedure activations. The value of N need not be very large, because in general, the depth of procedure activation is small. In case of recursive call the depth of procedure call may increase. From survey, it is found that with 8 windows, a save or restore is needed on only 1% of the calls or returns.
Global Variables

The window scheme provides an efficient organization for storing local scalar variables in registers. Global variables are accessed by more than one procedure.

Two solutions to access the global variables:

1. Variables declared as global in an HLL can be assigned memory location by the compiler, and all machine instructions that reference these variables will use memory reference operands. This scheme is inefficient for frequently accessed global variables.
2. An alternative is to incorporate a set of global registers in the processor. These registers would be fixed in number and available to all procedures. In this case, the compiler must decide which global variables should be assigned to registers.

Compiler based Register Optimization

A small number of registers (e.g. 16-32) is available on the target RISC machine and the concept of registers window can not be used. In this case, optimized register usage is the responsibility of the compiler.

A program written is a high level language has no explicit references to registers. The objective of the compiler is to keep the operands for as many computations as possible in registers rather than main memory, and to minimize load and store operations.

To optimize the use of registers, the approach taken 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 the memory location.

The task of optimization is to decide which quantities are to be assigned to registers at any given point of time in the program. The technique most commonly used in RISC compiler is known as graph coloring.
The graph coloring problem 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 graph coloring problem is mapped to the register optimization problem of the compiler in the following way:

- 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 indicate interference.
- An attempt is then made to color the graph with \( n \) colors, where \( n \) is the number of register.
- Nodes that cannot be colored are placed in memory.
- Load and store must be used to make space for the affected quantities when they are needed.

The part 'a' of Figure 8.3 shows a program with seven symbolic registers to be compiled in three actual registers. Part ‘b’ of Figure 8.3 shows the register interference graph. A possible coloring with three colors is shown. Only, a symbolic register E is left uncolored and must be dealt with load and store.

![Figure 8.3](mywbut.com)
Large Register file versus cache

The Register file, organized into windows, acts as a small, fast buffer for holding a subset of all variables that are likely to be used the most heavily. From this point of view, the register file acts much like a cache memory.

The question therefore arises as to whether it would be simpler and better to use a cache and a small traditional register file instead of using a large register file. The Table 8.3 compares the characteristics of two approaches.

<table>
<thead>
<tr>
<th>Table 8.3: The characteristics of the two approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Large Register File</strong></td>
</tr>
<tr>
<td>1. All local scalars</td>
</tr>
<tr>
<td>2. Individual variables</td>
</tr>
<tr>
<td>3. Compiler-assigned global variables</td>
</tr>
<tr>
<td>4. Save/Restore based on procedure nesting depth</td>
</tr>
<tr>
<td>5. Register addressing</td>
</tr>
</tbody>
</table>

Problems

Q 1: What are the distinguishing characteristics of RISC organization?

Q 2: Briefly explain the basic approaches used to minimize register-memory operations on RISC machines.

Q 3: Give some reasons for shifting the paradigm from CISC to RISC.

Q 4: Explain the concept of register window to handle the procedure calls.

Q 5: If a circular register buffer is used to handle local variables for nested procedures, describe the approaches for handling global variables.

Q 6: Explain the concept of graph coloring to optimize the register uses.

Q 7: What are the differences of using large register file and cache memory?