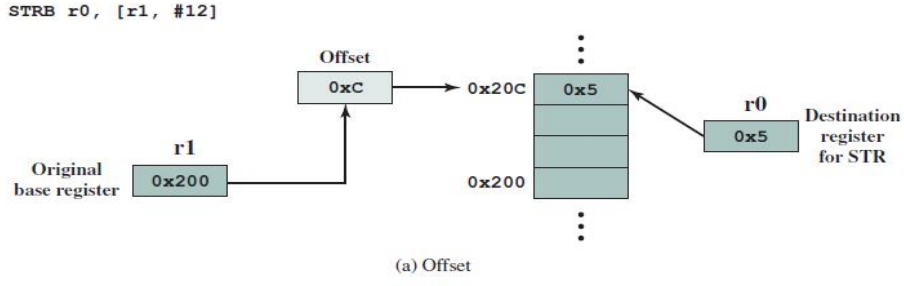
Instruction Set Architecture: Addressing Modes and Formats

ARM Addressing Modes

Load/store addressing: Load and store instructions are the only instructions that reference memory. Done indirectly through a base register plus offset. There are three alternatives with respect to indexing (Figure 13.3):

Offset: For this addressing method, indexing is not used. An offset value is added to or subtracted from the value in the base register to form the memory address. Figure 13.3a illustrates this method with the assembly language instruction STRB r0, [r1, #12]. This is the store byte instruction.

Base address is in register r1 and the displacement is an immediate value of decimal 12. The resulting address (base plus offset) is the location where the least significant byte from r0 is to be stored.

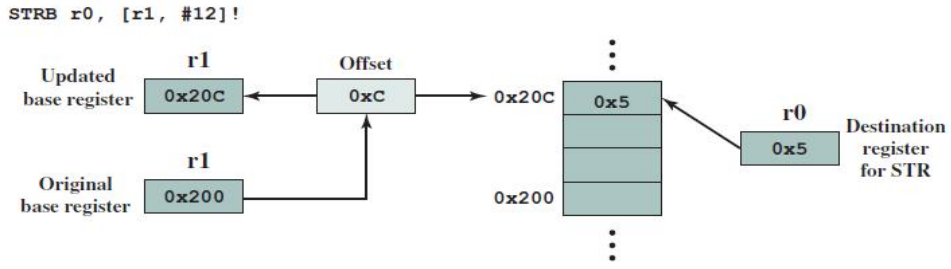


(a) Offset

ARM Addressing Modes

Preindex: Memory address is formed in same way as for offset addressing. Memory address is also written back to the base register. Base register value is incremented or decremented by the offset value.

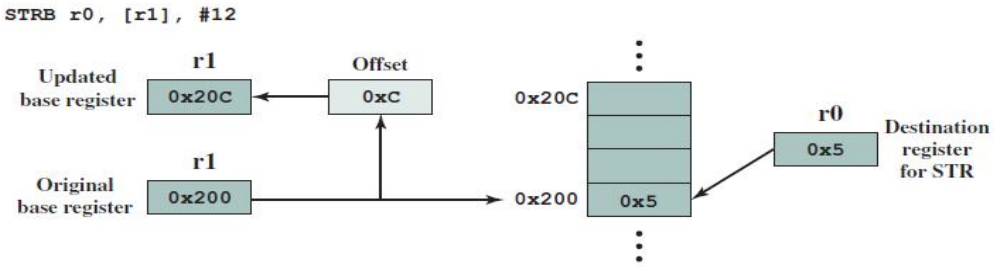
Figure 13.3b illustrates this method with the assembly language instruction STRB r0, [r1, #12]!. The exclamation point signifies preindexing.



(b) Preindex

ARM Addressing Modes

Postindex: The memory address is the base register value. An offset is added to or subtracted from the base register value and the result is written back to the base register.Figure 13.3c illustrates this method with the assembly language instruction STRB r0, [r1], #12.



(c) Postindex

Data processing instruction addressing: Use either register addressing or a mixture of register and immediate addressing. For register addressing, value in one of the register operands may be scaled using one of the five shift operators Logical Shift Left, Logical Shift Right, Arithmetic Shift Right, Rotate Right, or Rotate Right Extended (which includes the carry bit in the rotation). Amount of shift is specified as an immediate value in the instruction.

ARM Addressing Modes

Branch instructions: addressing for branch instructions is immediate addressing. The branch instruction contains a 24-bit value. For address calculation, this value is shifted left 2 bits, so that the address is on a word boundary. Effective address range is { ±32 MB from the program counter.

Load/store multiple addressing: Load a subset of the general- purpose registers from memory. Store Multiple instructions store a subset of the general- purpose registers to memory.

list of registers for the load or store is specified in a 16-bit field in the instruction with each bit corresponding to one of the 16 registers.

Produce a sequential range of memory addresses. The lowest- numbered register is stored at the lowest memory address and the highest- numbered register at the highest memory address.

ARM Addressing Modes

Four addressing modes are used (Figure 13.4): increment after, increment before, decrement after, and decrement before.

A base register specifies a main memory address where register values are stored in or loaded from in ascending (increment) or descending (decrement) word locations.

Incrementing or decrementing starts either before or after the first memory access. These instructions are useful for block loads or stores, stack operations, and procedure exit sequences.

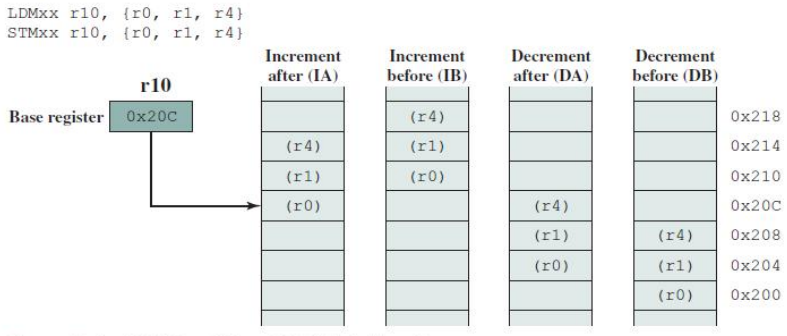


Figure 13.4 ARM Load/Store Multiple Addressing

Encoding an Instructiun Set

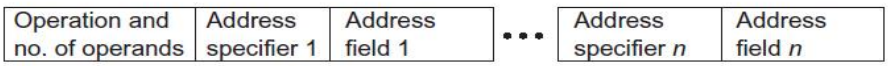
When encoding the instructions, the number of registers and the number of addressing modes both have a significant impact on the size of instructions, as the register field and addressing mode field may appear many times in a single instruction.

For most instructions many more bits are consumed in encoding addressing modes and register fields than in specifying the opcode.The architect must balance several competing forces when encoding the instruction set:

1. The desire to have as many registers and addressing modes as possible.
2. The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.
3. A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation. Many desktop and server architects have chosen to use a fixed length instruction to gain implementation benefits while sacrificing average code size.

Encoding an Instructiun Set

Figure A.18 shows three popular choices for encoding the instruction set. **Variable:** because it allows virtually all addressing modes to be with all operations. This style is best when there are many addressing modes and operations.



(A) Variable (e.g., Intel 80x86, VAX)

Fixed: because it combines the operation and the addressing mode into the opcode. Often fixed encoding will have only a single size for all instructions; it works best when there are few addressing modes and operations.

The trade-off between variable encoding and fixed encoding is size of programs versus ease of decoding in the processor. Variable tries to use as few bits as possible to represent the program, but individual instructions can vary widely in both size and the amount of work to be performed.

Let's look at an 80x86 instruction to see an example of the variable encoding:

add EAX,1000(EBX)

Encoding an Instructiun Set

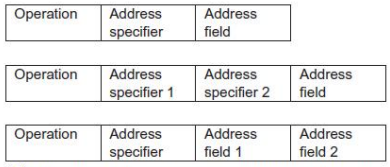
add means a 32-bit integer add instruction with two operands, and this opcode takes 1 byte. 80x86 address specifier is 1 or 2 bytes, specifying the source/destination register (EAX) and the addressing mode (displacement in this case) and base register (EBX) for the second operand. This combination takes 1 byte to specify the operands.

When in 32-bit mode, the size of the address field is either 1 byte or 4 bytes. Because1000 is bigger than 2^8, the total length of the instruction is :

1+1+4=6 bytes.

Length of 80x86 instructions varies between 1 and 17 bytes. 80x86 programs are generally smaller than the RISC architectures, which use fixed formats.

Hypbrid: reduce the variability in size and work of the variable architecture but provide multiple instruction lengths to reduce code size. This hybrid approach is the third encoding alternative.



(C) Hybrid (e.g., RISC V Compressed (RV32IC), IBM 360/370, microMIPS, Arm Thumb2)

Encoding an Instructiun Set

Reduced Code Size in RISCs: RISC computers started being used in embedded applications, the 32-bit fixed format became a liability because cost and smaller code are important.

Several manufacturers offered a new hybrid version of their RISC instruction sets, with both 16-bit and 32-bit instructions. The narrow instructions support fewer operations, smaller address and immediate fields, fewer registers, and the two-address format rather than three-address format of RISC computers.

RISC-V offers such an extension, called RV32IC, the C standing for compressed. Common instruction occurrences, such as intermediates with small values and common ALU operations with the source and destination register being identical, are encoded in 16-bit formats.

Examples: ARM Thumb and micro MIPS both claim a code size reduction of up to 40%.

Encoding an Instructiun Set

IBM compresses its standard instruction set and then adds hardware to decompress instructions as they are fetched from memory on an instruction cache miss.

Instruction cache contains full 32-bit instructions, but compressed code is kept in main memory, ROMs, and the disk. Advantage of a compressed format, such as RV32IC microMIPS and Thumb2 is that instruction caches act as if they are about 25% larger, while IBM's CodePack means that compilers need not be changed to handle different instruction sets and instruction decoding can remain simple.

CodePack starts with run-length encoding compression on any PowerPC program and then loads the resulting compression tables in a 2 KB table on chip. Every program has its own unique encoding.

To handle branches,PowerPC creates a hash table in memory that maps between compressed and uncompressed addresses. Like a TLB, it caches the most recently used address maps to reduce the number of memory accesses. IBM claims an overall performance cost of 10%, resulting in a code size reduction of 35%–40%.

Role of Compilers

Programming is done in high-level languages for desktop and server applications means most instructions executed are the output of a compiler, an instruction set architecture is essentially a compiler target.

Previously,architectural decisions were often made to ease assembly language programming or for a specific kernel. Compiler will significantly affect the performance of a computer.

Phase Ordering Problem: compilers have to choose which procedure calls to expand inline before they know the exact size of the procedure being called. Compiler writers call this problem the phase-ordering problem.

How this ordering of transformations interact with the instruction set architecture?

Global common subexpression elimination:T his optimization finds two instances of an expression that compute the same value and saves the value of first computation in a temporary. It then uses the temporary value, eliminating the second computation of the common expression. For this optimization to be significant, the temporary must be allocated to a register. Otherwise, the cost of storing the temporary in memory and later reloading may negate the savings gained by not recomputing the expression.

Structure of Compilers

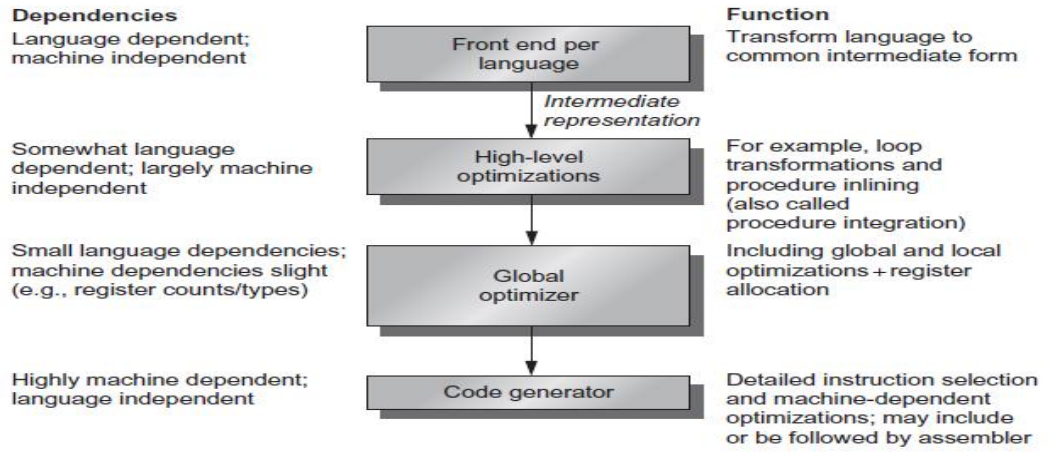


Figure A.19 Compilers typically consist of two to four passes, with more highly optimizing compilers having more passes. This structure maximizes the probability that a program compiled at various levels of optimization will produce the same output when given the same input. The optimizing passes are designed to be optional and may be skipped when faster compilation is the goal and lower-quality code is acceptable. A *pass* is simply one phase in which the compiler reads and transforms the entire program. (The term *phase* is often used interchangeably with *pass*.) Because the optimizing passes are separated, multiple languages can use the same optimizing and code generation passes. Only a new front end is required for a new language.

Structure of Compilers

Optimizations performed by modern compilers can be classified by style of transformation as follows:

- ✓ High-level optimizations are often done on the source with output fed to later optimization passes.
- ✓ Local optimizations optimize code only within a straight-line code fragment (called a basic block by compiler people).
- ✓ Global optimizations extend the local optimizations across branches and introduce a set of transformations aimed at optimizing loops.
- ✓ Register allocation associates registers with operands.
- ✓ Processor-dependent optimizations attempt to take advantage of specific architectural knowledge.

Register Allocation

Central role that register allocation plays, both in speeding up the code and in making other optimizations useful.

Register allocation algorithms: a re based on a technique called graph coloring. Graph coloring is to construct a graph representing the possible candidates for allocation to a register and then to use the graph to allocate registers.

Problem is how to use a limited set of colors so that no two adjacent nodes in a dependency graph have the same color. The emphasis in the approach is to achieve 100% register allocation of active variables.

Graph coloring works best when there are at least 16 general-purpose registers available for global allocation for integer variables and additional registers for floating point.

Compiler based Register Optimization

Graph coloring problem: Given a graph consisting of nodes and edges, assign colors to nodes such that adjacent nodes have different colors, and minimize the number of different colors. This problem is adapted to the compiler problem in the following way.

First program is analyzed to build a register interference graph. Nodes of graph are 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.

Compiler based Register Optimization

Figure 15.4 , assume a program with six symbolic registers to be compiled into three actual registers. Figure 15.4a shows time sequence of active use of each symbolic register. Dashed horizontal lines indicate successive instruction executions.

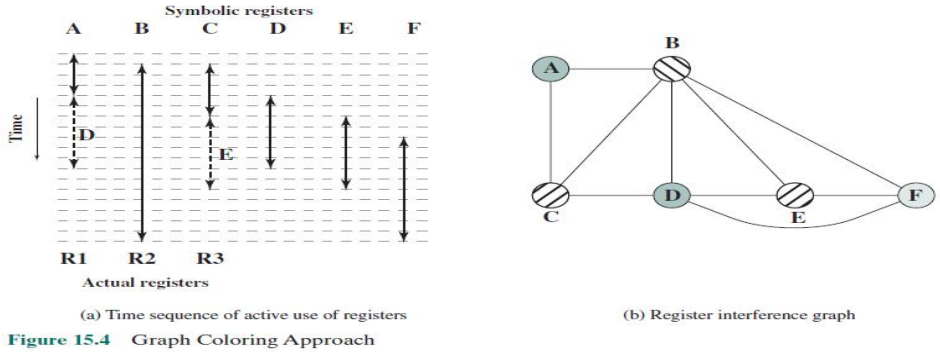


Figure 15.4 Graph Coloring Approach

Figure 15.4b shows register interference graph (shading and stripes are used instead of colors). A possible coloring with three colors is indicated. Symbolic registers A and D do not interfere, compiler can assign both of these to physical register R1. Symbolic registers C and E can be assigned to register R3. One symbolic register F is left uncolored and dealt with using loads and stores.

Impact of Compiler Technology on Architect's Decisions

There are two important questions: how are variables allocated and addressed? How many registers are needed to allocate variables appropriately?

To address these questions,three separate areas in which current high-level languages allocate their data:

Stack: is used to allocate local variables. The stack is grown or shrunk on procedure call or return. Objects on the stack are addressed relative to the stack pointer and are primarily single variables rather than arrays. It is used for activation records, not as a stack for evaluating expressions. Values are almost never pushed or popped on the stack.

Global data: area is used to allocate statically declared objects, such as global variables and constants. A large percentage of these objects are arrays or other aggregate data structures.

Heap: is used to allocate dynamic objects that do not adhere to a stack discipline. Objects in heap are accessed with pointers and are typically not scalars.

Impact of Compiler Technology on Architect's Decisions

Register allocation is more effective for stack-allocated objects than for global variables.

Register allocation is impossible for heap-allocated objects because they are accessed with pointers. Global variables and some stack variables are impossible to allocate because they are aliased.There are multiple ways to refer to the address of a variable, making it illegal to put it into a register.

For example, consider the following code sequence, where & returns the address of a variable and * dereferences a pointer:

```
p =&a   - gets address of a in p
a =...  - assigns to a directly
*p =... - uses p to assign to a
...a... - accesses a
```

The variable **a** could not be register allocated across the assignment to ***p** without generating incorrect code. Aliasing causes a problem because it is difficult to decide what objects a pointer may refer to.

A compiler must be conservative; some compilers will not allocate any local variables of a procedure in a register when there is a pointer that may refer to one of local variables.

Use of a Large Register File

Overlapping Register Window: Procedure call 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. Figure 15.1 shows w indow is divided into three fixed- size areas.

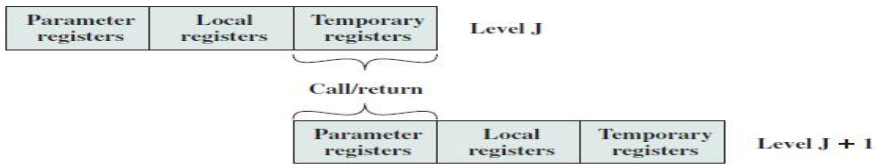


Figure 15.1 Overlapping Register Windows

Parameter registers: hold parameters passed down from 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 next lower level. Temporary registers at one level are physically same as parameter registers at next lower level. This overlap permits parameters to be passed without actual movement of data. Parameter and local registers at level J are disjoint from local and temporary registers at level J + 1.

Use of a Large Register File

Actual organization of the register file is a circular buffer of overlapping windows. The circular organization is shown in Figure 15.2, which depicts a circular buffer of six windows.

Buffer is filled to a depth of 4 (A called B; B called C; C called D) with procedure D active.

Current- window pointer (CWP): points to window of active procedure. Register references by a machine instruction are offset by this pointer to determine actual physical register.

Saved-window pointer (SWP): identifies the window most recently saved in memory.

If procedure D now calls procedure E, arguments for E are placed in D's temporary registers (the overlap between w3 and w4) and the CWP is advanced by one window.

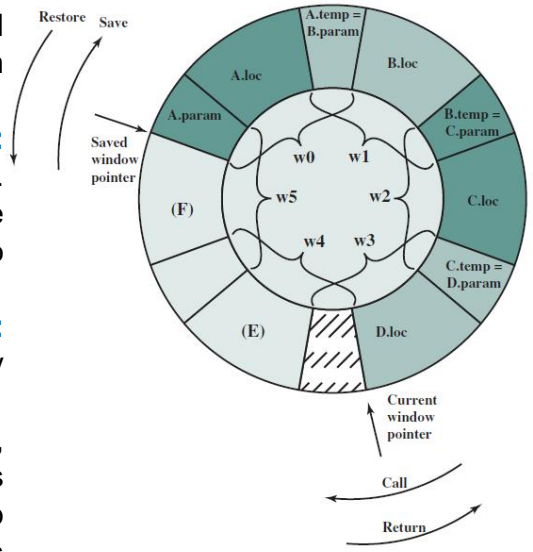


Figure 15.2 Circular-Buffer Organization of Overlapped Windows

Use of a Large Register File

If procedure E then makes a call to procedure F, call cannot be made with current status of buffer. This is because F's window overlaps A's window.

If F begins to load its temporary registers, it will overwrite the parameter registers of A (A.in).

When CWP is incremented (modulo 6) it becomes equal to SWP, an interrupt occurs, and A's window is saved. Only the first two portions (A.in and A.loc) need be saved. SWP is incremented and the call to F proceeds.A similar interrupt can occur on returns.

For example,when B returns to A, CWP is decremented and becomes equal to SWP. This causes an interrupt that results in the restoration of A's window.

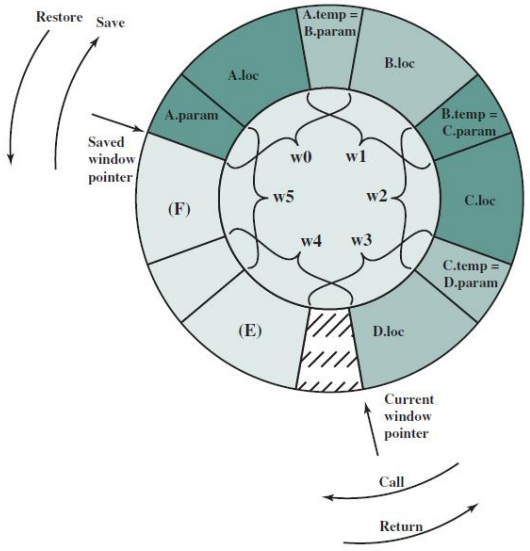


Figure 15.2 Circular-Buffer Organization of Overlapped Windows

Large Register File vs Cache

Register file acts much like a cache memory, although a much faster memory. Table 15.5 compares characteristics of the two approaches

Table 15.5 Characteristics of Large-Register-File and Cache Organizations

Large Register File	Cache
All local scalars	Recently-used local scalars
Individual variables	Blocks of memory
Compiler-assigned global variables	Recently-used global variables
Save/Restore based on procedure nesting depth	Save/Restore based on cache replacement algorithm
Register addressing	Memory addressing
Multiple operands addressed and accessed in one cycle	One operand addressed and accessed per cycle

Large Register File vs Cache

Windows based Register File: Figure 15.3 illustrates, 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.

Reference a memory location in cache: a full-width memory address must be generated. Complexity of this operation depends on addressing mode.

In a set associative cache, a portion of the address is used to read a number of words and tags equal to the set size.

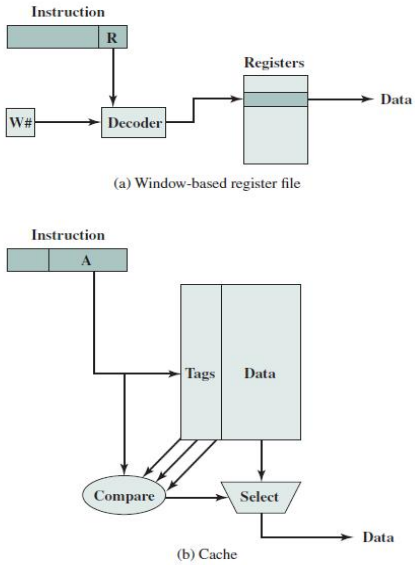


Figure 15.3 Referencing a Scalar

Large Register File vs Cache

Another portion of address is compared with tags, and one of words that were read is selected.

If the cache is as fast as the register file,access time will be considerably longer.

From performance view, the window-based register file is superior for local scalars.

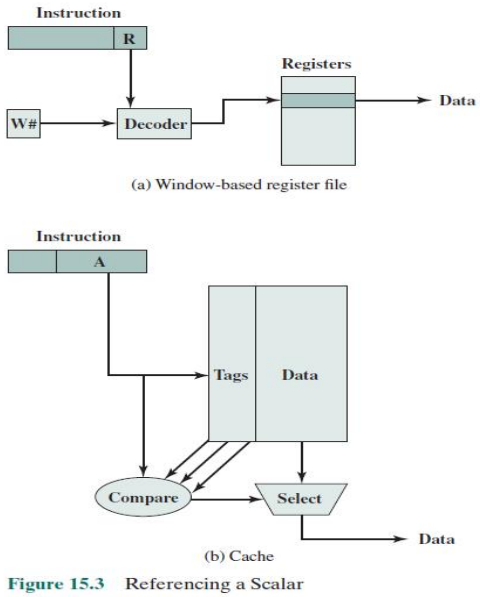


Figure 15.3 Referencing a Scalar

How Architect Can Help Compiler Writer

Some instruction set properties help the compiler writer.:

Provide regularity: Three primary components of an instruction set, operations, data types, and the addressing modes should be irrelevant.

For example, operations and addressing modes are irrelevant if for every operation one addressing mode can be applied, all addressing modes are applicable.

This regularity helps simplify code generation and is important when decision about what code to generate is split into two passes in the compiler. Example is restricting registers that can be used for a certain class of instructions.

Provide primitives not solutions: Special features that “match” a language construct or a kernel function are unusable. Attempts to support high level languages may work only with one language and required for a correct and efficient implementation of language.

How Architect Can Help Compiler Writer

Simplify trade-offs among alternatives:C ompiler writer is figuring out what instruction sequence will be best for every segment of code that arises. Previously, instruction counts or total code size might have been good metrics.

With caches and pipelining, trade-offs have become very complex. Compiler writer understand costs of alternative code sequences would help improve the code. Most difficult instances of complex trade-offs occurs in a register-memory architecture in deciding how many times a variable should be referenced before it is cheaper to load it into a register. This threshold is hard to compute and vary among models of the same architecture.

Provide instructions that bind the quantities known at compile time as constants: A compiler writer do not like processor interpreting at runtime a value that was known at compile time. Example of include instructions that interpret values that were fixed at compile time. For instance, VAX procedure call instruction (calls) dynamically interprets a mask saying what registers to save on a call, but the mask is fixed at compile time.

Compiler Support for Multimedia Instructions

SIMD instructions tend to be solutions, not primitives; they are short of registers; and data types do not match existing programming languages.

Vector architectures operate on vectors of data. Invented originally for scientific codes, multimedia kernels are vectorizable as well.

MMX with vectors of eight 8-bit elements, four 16-bit elements, or two 32-bit elements, and AltiVec (SIMD instruction set) with vectors twice that length. They are implemented as simply adjacent, narrow elements in wide registers.

These microprocessor architectures build the vector register size into the architecture: the sum of the sizes of the elements is limited to 64 bits for MMX and 128 bits for AltiVec.

When Intel decided to expand to 128-bit vectors, it added a whole new set of instructions called streaming SIMD extension (SSE).

A major advantage of vector computers is hiding latency of memory access by loading many elements at once and then overlapping execution with data transfer. Goal of vector addressing modes is to collect data scattered about memory, place them in a compact form so that they can be operated efficiently.

Compiler Support for Multimedia Instructions

Vector computers include strided addressing and gather/scatter addressing to increase the number of programs that can be vectorized.

Strided addressing: skips a fixed number of words between each access, so sequential addressing is often called unit stride addressing.

Gather and scatter: find their addresses in another vector register: think of it as register indirect addressing for vector computers.

SIMD computers support only unit strided accesses: memory accesses load or store all elements at once from a single wide memory location.

Because the data for multimedia applications are often streams that start and end in memory, strided and gather/scatter addressing modes are essential to successful vectorization.

Compiler Support for Multimedia Instructions

Example As an example, compare a vector computer with MMX for color representation conversion of pixels from RGB (red, green, blue) to YUV (luminosity chrominance), with each pixel represented by 3 bytes. The conversion is just three lines of C code placed in a loop:

```
Y = (9798*R + 19235*G + 3736*B) / 32768;
U = (-4784*R 9437*G + 4221*B) / 32768 + 128;
V = (20218*R 16941*G 3277*B) / 32768 + 128;
```

A 64-bit-wide vector computer can calculate 8 pixels simultaneously. One vector computer for media with strided addresses takes

- 3 vector loads (to get RGB)
- 3 vector multiplies (to convert R)
- 6 vector multiply adds (to convert G and B)
- 3 vector shifts (to divide by 32,768)
- 2 vector adds (to add 128)
- 3 vector stores (to store YUV)

The total is 20 instructions to perform the 20 operations in the previous C code to convert 8 pixels (Kozyrakis, 2000). (Because a vector might have 32 64-bit elements, this code actually converts up to 32×8 or 256 pixels.)

In contrast, Intel's website shows that a library routine to perform the same calculation on 8 pixels takes 116 MMX instructions plus 6 80x86 instructions (Intel, 2001). This six-fold increase in instructions is due to the large number of instructions to load and unpack RGB pixels and to pack and store YUV pixels, because there are no strided memory accesses.

Reduced Instruction Set Architecture (RISC)

Characteristics of RISC

One instruction per cycle: 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.

RISC machine instructions should be no more complicated and execute as fast as microinstructions. Such instructions should execute faster than comparable machine instructions on other machines, because it is not necessary to access a microprogram control store during instruction execution.

Register-to-register operations: register to register with only LOAD and STORE operations accessing memory. This design feature simplifies the instruction set and control unit.

For example, a RISC instruction set may include only one or two ADD instructions (e.g., integer add, add with carry); VAX has 25 different ADD instructions.

Another benefit is that such an architecture encourages the optimization of register use, so that frequently accessed operands remain in high- speed storage.This emphasis on register- to- register operations is notable for RISC designs.

Characteristics of RISC

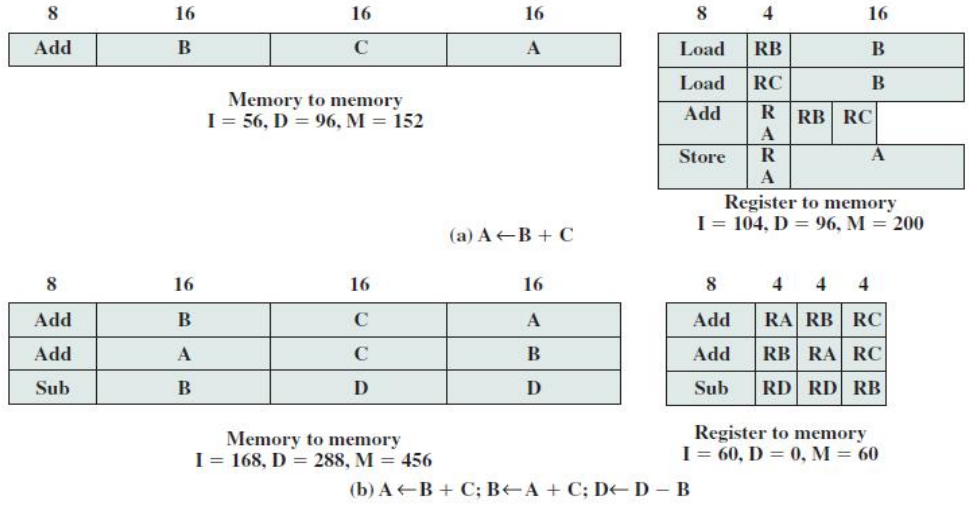


Figure 15.5 Two Comparisons of Register-to-Register and Memory-to-Memory Approaches

Characteristics of RISC

Simple addressing modes: Almost all RISC instructions use simple register addressing. Several additional modes, such as displacement and PC-relative, may be included. More complex modes can be analyze in software from the simple ones. This design feature simplifies the instruction set and the control unit.

Simple instruction formats: O nly one or a few formats are used. Instruction length is fixed and aligned on word boundaries.

Field locations, especially the opcode, are fixed. This design feature has a number of benefits. With fixed fields, opcode decoding and register operand accessing can occur simultaneously.

Simplified formats simplify the control unit. Instruction fetching is optimized because word-length units are fetched. Alignment on a word boundary also means that a single instruction does not cross page boundaries.

Performance benefits of RISC

1. With more-primitive instructions, there are more opportunities for moving functions out of loops, reorganizing code for efficiency, maximizing register utilization, and so forth. It is even possible to compute parts of complex instructions at compile time.

For example, the S/390 Move Characters (MVC) instruction moves a string of characters from one location to another. Each time it is executed, the move will depend on the length of the string, whether and in which direction the locations overlap, and what the alignment characteristics are. In most cases, these will all be known at compile time. Compiler could produce an optimized sequence of primitive instructions for this function.

2. Most instructions generated by a compiler are relatively simple anyway. A control unit built specifically for those instructions and using little or no microcode could execute them faster than a comparable CISC.

Performance benefits of RISC

3.Use of instruction pipelining can be applied much more effectively with a reduced instruction set.

4.RISC processors are more responsive to interrupts because interrupts are checked between rather elementary operations. Architectures with complex instructions either restrict interrupts to instruction boundaries or must define specific interruptible points and implement mechanisms for restarting an instruction.

CISC vs RISC

Table 15.7 Characteristics of Some Processors

Processor	Number of instruction sizes	Max instruction size in bytes	Number of addressing modes	Indirect addressing	Load/store combined with arithmetic	Max number of memory operands	Unaligned addressing allowed	Max number of MMU uses	Number of bits for integer register specifier	Number of bits for FP register specifier
AMD29000	1	4	1	no	no	1	no	1	8	3[a]
MIPS R2000	1	4	1	no	no	1	no	1	5	4
SPARC	1	4	2	no	no	1	no	1	5	4
MC88000	1	4	3	no	no	1	no	1	5	4
HP PA	1	4	10[a]	no	no	1	no	1	5	4
IBM RT/PC	2[a]	4	1	no	no	1	no	1	4[a]	3[a]
IBM RS/6000	1	4	4	no	no	1	yes	1	5	5
Intel i860	1	4	4	no	no	1	no	1	5	4
IBM 3090	4	8	2[b]	no[b]	yes	2	yes	4	4	2
Intel 80486	12	12	15	no[b]	yes	2	yes	4	3	3
NSC 32016	21	21	23	yes	yes	2	yes	4	3	3
MC68040	11	22	44	yes	yes	2	yes	8	4	3
VAX	56	56	22	yes	yes	6	yes	24	4	0
Clipper	4[a]	8[a]	9[a]	no	no	1	0	2	4[a]	3[a]
Intel 80960	2[a]	8[a]	9[a]	no	no	1	yes[a]	–	5	3[a]

Notes: [a] RISC that does not conform to this characteristic.
[b] CISC that does not conform to this characteristic.

In the table, first eight processors are RISC architectures, next five are CISC, and last two are processors thought of as RISC that have many CISC characteristics

RISC

1. A single instruction size. 2. That size is typically 4 bytes.

3. A small number of data addressing modes less than five. Register and literal modes are not counted and different formats with different offset sizes are counted separately.

4. No indirect addressing requires to make one memory access to get address of another operand in memory.

5. No operations that combine load/store with arithmetic (add from memory, add to memory).

6. No more than one memory-addressed operand per instruction.

7. Does not support arbitrary alignment of data for load/store operations.

8. Maximum use of memory management unit (MMU) for a data address in an instruction.

9. Number of bits for integer register specifier equal to five or more. This means that at least 32 integer registers can be explicitly referenced at a time.

10. Number of bits for floating-point register specifier equal to four or more. This means that at least 16 floating-point registers can be explicitly referenced at a time.

Items 1 to 3 are of instruction decode complexity. Items 4 to 8 suggest the ease or difficulty of pipeliningin the presence of virtual memory requirements. Items 9 and 10 are related to to take advantage of compilers.

RISC-V

Instruction Set Organization:

RISC-V instruction set is organized as three base instruction sets that support 32-bit or 64-bit integers, and a variety of optional extensions to one of base instruction sets.

This allows RISC-V to be implemented for a wide range of applications from small embedded processor to high-end processor configurations with full support for floating point, vectors, and multiprocessor configurations.

Name of base or extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers
RV32E	Base 32-bit instruction set but with only 16 registers; intended for very low-end embedded applications
RV64I	Base 64-bit instruction set; all registers are 64-bits, and instructions to move 64-bit from/to the registers (LD and SD) are added
M	Adds integer multiply and divide instructions
A	Adds atomic instructions needed for concurrent processing; see Chapter 5
F	Adds single precision (32-bit) IEEE floating point, includes 32 32-bit floating point registers, instructions to load and store those registers and operate on them
D	Extends floating point to double precision, 64-bit, making the registers 64-bits, adding instructions to load, store, and operate on the registers
Q	Further extends floating point to add support for quad precision, adding 128-bit operations
L	Adds support for 64- and 128-bit decimal floating point for the IEEE standard
C	Defines a compressed version of the instruction set intended for small-memory-sized embedded applications. Defines 16-bit versions of common RV32I instructions
V	A future extension to support vector operations (see Chapter 4)
B	A future extension to support operations on bit fields
T	A future extension to support transactional memory
P	An extension to support packed SIMD instructions: see Chapter 4
RV128I	A future base instruction set providing a 128-bit address space

Figure A.22 RISC-V has three base instructions sets (and a reserved spot for a future fourth); all the extensions extend one of the base instruction sets. An instruction set is thus named by the base name followed by the extensions. For example, RISC-V64IMAFD refers to the base 64-bit instruction set with extensions M, A, F, and D. For consistency of naming and software, this combination is given the abbreviated name: RV64G, and we use RV64G through most of this text.