

ISA Issues

Prof. Loh

CS3220 - Processor Design - Spring 2005

February 15, 2005

1 What is an “Architecture”?

When we refer to “Architecture,” we often use it as an abbreviation for an *instruction set architecture* (ISA). The ISA is the interface between hardware and software. The ISA acts as a contract between the hardware and the programmer/compiler/assembler. The architecture is a specification for what the processor will do, and how the software must communicate to the processor what it wants done. Note that the “architecture” states only *what* will be done, but not *how*.

The “how” part of a processor is typically considered the *microarchitecture* and can involve techniques such as pipelining, superscalar execution, caching, and other techniques that are functionally invisible to the user. By functionally invisible, we mean that a program executed on two processors with different microarchitectures produce the same result. One might produce the result faster or require less power than the other, but at the end of the day, “Hello World!” shows up in either case.

2 RISC

The design goals and principles behind reduced instruction set computers (RISC) are:

- Smaller is faster
- Simplicity favors regularity
- Make the common case fast

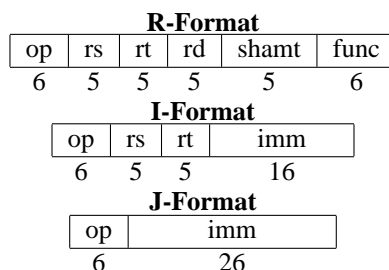
Regularity will basically apply to instruction sizes and instruction formats.

2.1 MIPS

MIPS is a common RISC ISA. It used to be more popular as it was used in the processors for SGI workstations. Today, MIPS processors tend to show up more often in the embedded domain rather than in the high-performance arena.

MIPS has 32 integer registers, which requires 5-bit specifiers to indicate a source or destination register. MIPS uses a “3-address” format, where two “addresses” specify the sources, and the other address specifies the index. Corresponding to these addresses are only a limited number of *addressing modes*.

MIPS has a fairly regular instruction encoding. All instructions are 32 bits long (4 bytes), and there are only three instruction formats:



To further increase the regularity of encoding, note that across the different formats, when a field is used more than once (e.g. rs exists in both the R- and I-formats), the field appears in the same location. This simplifies the job of the decoder because it can always check the same bits for a given field. Similarly, the opcode is always in the same place, which is needed to determine the format (R/I/J) of the instruction.

The design of an instruction set architecture involves different tradeoffs. Larger instructions provide more bits to encode more information (e.g. more opcodes, longer immediates), but comes at the cost of a larger overall instruction footprint (i.e. your program will require more bytes to store in memory).

The I-format provides 16-bit immediates. A consequence is that programs that use constants must be able to fit those constants within 16 bits. This goes along with the principle of making the common case fast. In common programs, most values do fit within 16 bits. For this common case, the constant can be embedded directly into an instruction using the I-format. In the less common case of a larger constant, MIPS allows the programmer to build a 32-bit constant by using two instructions.

In a similar vein, branching instructions that use the I- or J-formats are limited in specifying the target address. For I-format branches, the target address is specified using a *PC-relative* addressing mode. On a not-taken branch, the next instruction PC (program counter) is equal to the current instruction address plus the size of an instruction (PC+4). For a taken branch, the target is specified as the default next-instruction address plus the sign-extended value of the immediate (PC+4+4×SExt(imm)). The ×4 is due to instructions being 4 bytes long. This means that I-format branches can only jump to a target that is within $\approx \pm 32K$ instructions. For a target that is further away, the compiler must use a different kind of jump (such as one that reads the target from a register), or must “daisy-chain” multiple smaller branches together.

J-format branches/jumps use a slightly different addressing mode called *pseudo-direct* addressing. The target of the jump is computed by taking the four most significant bits of the current PC, the 26 bits specified by the J-format immediate, and the last two bits are zero (again due to all instructions being four bytes wide). This allows the programmer to jump anywhere within a 256MB section of memory, although one must be careful when jumping near the boundaries.

These branch/jump targets seem like they may pose some difficulties to the compiler or programmer. This is usually not the case, because the common case is that branch targets are nearby.

RISC architectures are usually *load-store architectures*, which means all data coming from/going to memory must go through a load or store instruction. Any other instruction that manipulates data in any way such as computation or branches can only read the data from registers or immediates. To add a value from memory, a load instruction must first move that value into a register before the addition can proceed. This constraint makes the hardware simpler.

RISC architectures favor hardware simplicity, but this comes at some cost. The complexity is pushed to the compiler. The compiler must find efficient ways to decompose high-level programming tasks into the small simple

RISC instructions. This may also cause some degree of code bloat because common tasks must be encoded as a potentially long sequence of many small RISC operations.

2.2 Alpha

Alpha is another RISC architecture designed specifically for very high performance, high clock frequency processors. Alpha and MIPS are both RISC, and being so they have many similarities. On the other hand, Alpha has several differences as well. Alpha has five major instruction formats:

Operate Format

op	ra	rb	sbz	0	func	rc
6	5	5	3	1	7	5

SBZ stands for "should be zero". The instruction is considered to be undefined if any of these bits are non-zero.

Operate Format (With Immediate)

op	ra	imm	1	func	rc
6	5	8	1	7	5

Memory Format

op	ra	rb	disp/func
6	5	5	16

Branch Format

op	ra	branch-disp
6	5	21

FP Format

op	fa	fb	func	fc
6	5	5	11	5

There are also a few additional formats for less commonly used tasks, such as for the invocation of system calls. The Alpha instructions have a different layout than MIPS, but there are similar themes:

- opcodes in same place
- register fields in same place (when present)
- all instructions are the same size (32-bits/4 bytes).

Note that while Alpha is a 64-bit architecture (64-bit registers and a 64-bit virtually addressable memory space), the instructions are still only 32 bits long.

In the Alpha instructions, the register destination field (when present) is located at the end of the instruction. This is true for both operate formats (with and without an immediate). Compare this to the MIPS encodings where in the I-format, one of the register fields that encodes a source for the R-format encodes a destination for the I-format. This kind of special case behavior adds complexity to the decoding logic.

3 CISC

RISC ISAs focus on fast access to information. This includes easy decoding of instructions, instruction caches to quickly provide new instructions, a data cache to quickly provide memory operands, and a large register file to reduce the amount of memory traffic necessary due to register spills and fills. The ability to implement caches and large register files is a result of Moore's Law providing processor designers with enough transistors to implement all of these structures.

Complex instruction set computer (CISC) instruction set architectures were largely designed in a different era where transistors were not as plentiful. As a result, caches were not available and register files tended to be smaller. This increases the average latency to read data from memory, and therefore the architectures tend to favor optimizations that reduce memory usage or make memory accesses more efficient. In other words, do as much as you can with each memory access and make it count.

From an instruction encoding format, CISC architectures tend to provide a "richer" set of instructions that are more expressive. The operations that one can perform with a single RISC instruction tend to be very simple. With CISC instructions, a single instruction may encode an entire string copy operation! A clever compiler may be able to map multiple high-level operations into a single instruction. On the other hand, this may further complicate the code generation phase of the compiler because the ISA provide multiple ways to execute the same basic operations.

3.1 x86

The Intel x86 architecture has a long history stemming from Intel's first product, the 4004. The x86 ISA is a CISC architecture with a high level of complexity. There are only eight general purpose registers.

Consider the class of "move" instructions. In x86, moves cover about five major types of operations:

1. General purpose data movement

- Register to register
- Memory to register (loads)
- Register to memory (stores)
- Immediate to register
- Immediate to memory

2. Exchanges

- Swap contents of two registers
- Swap byte order within a register

3. Stack Manipulation

- Push/pop items between a register and the stack
- Push/pop *all* eight x86 registers to/from the stack (with a single instruction!)

4. Type Conversion (move with sign/zero extend)

5. Conditional Moves

- Move register to register or memory to register based on some condition. Replaces branch+move with a single *cmov* instruction.

... and this is only for data movement instructions. Most other instructions have several formats and specific types of operations that may take a much larger number of equivalent RISC instructions to perform. CISC architectures tend to pack as much "work" into each byte as efficiently as possible, which leads to better overall code density.

This code density comes at a price. The process of decoding x86 instructions is fairly complicated and involves a great amount of logic. CISC is not complex because there are a large number of instructions, but rather they are

complex because there are many instruction formats with all sorts of special cases. There are optional components to instructions such as prefix bytes, escapes, immediates, displacements, and bytes for providing additional register specifiers or opcode bits. This leads to instructions with varying lengths.

The basic x86 instruction format looks like:

prefixes	opcode	Mod R/M	SIB	displacement	immediate
0-4 bytes	1/2 bytes	0/1 byte	0/1 byte	0/1/2/4 bytes	0/1/2/4 bytes

As a result, x86 instructions may range anywhere from 1 to 15 bytes in length. The opcode specifies the operation, and whether or not the additional Mod R/M byte is needed. Most instructions use the Mod R/M byte which specifies an addressing mode (mod) and register (R) and memory (M) operands.

The format of the Mod R/M byte is:

mod	reg	R/M	← name
MM	rrr	mmm	← bit abbreviations
2 bits	3 bits	3 bits	

The mod field specifies one of four addressing modes. The reg field specifies one of the eight general purpose registers. Depending on the mode, the R/M field specifies either the other register operand, or a register operand that is used to compute the address of the actual data operand. In general:

Mode=00: No-displacement addressing. Use the address indicated by the contents of register mmm.

Mode=01: 8-bit immediate. Use the address from register mmm, and add to it the sign-extended value of the byte at the end of this instruction.

Mode=10: 32-bit immediate. Same as previous, but use the 4-byte value at the end of the instruction.

Mode=11: Register-to-Register. Use the value in register mmm. x86 uses a two-address accumulator format. This means that the sources and destinations are specified with only two addresses, and the result “accumulates” in one of the source addresses (i.e. the destination is one of the sources). For example, a register-to-register add would take the form of $R_{rrr} = R_{rrr} + R_{mmm}$, where the original contents of the source R_{rrr} gets overwritten with the result.

As with many things in x86, there are exceptions to the rules. The most interesting involves the SIB byte. In modes 00, 01 or 10, if register mmm is equal to 0x4, then it means that there is additional addressing information that follows in the next byte (the SIB byte). SIB stands for scale, index and base and has the following format:

scale	index	base
ss	iii	bbb
2 bits	3 bits	3 bits

Working backwards, bbb specifies a register similar to the rrr and mmm fields of the Mod R/M byte. Similarly, iii specifies a register as well. The scale field indicates a scaling factor or a shift amount. We then define the value si to be equal to $R_{iii} \ll ss$ (the value of the register specified by iii shifted (scaled) by ss). This translates into multiplying register iii by 1, 2, 4, or 8.

At the end of the day, the final address is computed as follows:

$$\text{address} = (R_{iii} \ll ss) + R_{bbb}[+\text{Displacement}]$$

The displacement is only added for modes 01 and 10 where a displacement is provided. The overall data operand is then the value located at this address. Again, there is a special case where iii is equal to 0x4. In this case, instead of using $si = R_{iii} \ll ss$, the value of zero is used instead. This results in using only R_{bbb} , plus possibly a displacement if necessary. This covers the case where the programmer really wanted to use R_4 in the Mod R/M byte but it was treated as an escape to the SIB.

For reference, the register mapping for fields rrr, mmm, iii and bbb are:

register specifier	register used
000	EAX
001	ECX
010	EDX
011	EBX
100	ESP
101	EBP
110	ESI
111	EDI

The example we used in class was the single x86 instruction:

Move	Mod R/M	SIB	displacement	immediate
11000111	10000100	11000011	X X X X	X X X X

This 11-byte instruction results in the following high-level operation:

$$* ((EAX \ll 3) + EBX + \text{displacement}) = \text{immediate}$$

In a RISC architecture, it would take 8 instructions to perform the same operations (assuming EBX is in R2, EAX is in R3):

```

lui R1 = disp[31:16]
ori R1 = R1 | disp[15:0]
add R1 = R1 + R2
shli R3 = R3 << 3
add R3 = R3 + R1
lui R1 = imm[31:16]
ori R1 = R1 | imm[15:0]
st [R3] ← R1

```

These eight instructions take up a total of 32 bytes, which is almost three times the size of the single CISC instruction.

3.2 x86-64

The 64-bit extension to x86 provides several major changes to the architecture:

- The 8 basic registers are extended to be 64 bits wide
- 8 new general purposes registers are now available
- 8 extra SSE registers are available
- New prefix to indicate 64-bit operation

By default, most addresses are treated as 64-bit values. By default, all operations (adds, multiplies, etc.) are only on 32-bit values. The new “REX” prefix is used to indicate a 64-bit instruction:

0100	wRIB
------	------

The first four bits indicate the REX prefix. The width field (w) indicates whether the instruction should use 64-bit data (w=0) or 32-bit data (w=1). The remaining three bits are additional register specifier bits for the rrr (Mod R/M), iii (SIB) and bbb (SIB) register specifiers. By combining these four bits (one from REX plus the original three), any of the 16 general purpose registers can now be specified. For example, concatenating the B-bit from the REX prefix byte with the bbb bits from the SIB byte would specify register R_{Bbbb} . The width specifier allows one to write 32-bit applications that make use of the additional registers but not 64-bit operations.

While the 64-bit extensions may appear to be an ugly hack on top of x86, it has the important benefit that it (mostly) maintains backward compatibility with applications written for 32-bit versions of the x86 architecture.