Chapter 5 – A Closer Look at Instruction Set Architectures

CS 271 Computer Architecture
Purdue University Fort Wayne

1

Chapter 5 Objectives

- Understand the factors involved in instruction set architecture design.
- Gain familiarity with memory addressing modes.
- Understand the concepts of instruction-level pipelining and its affect upon execution performance.

2

5.1 Introduction

- This chapter builds upon the ideas in Chapter 4.
- We present a detailed look at different instruction formats, operand types, and memory access methods.
- We will see the interrelation between machine organization and instruction formats.
- This leads to a deeper understanding of computer architecture in general.

3

Outline

- Instruction formats
- Instruction types
- Addressing
- Instruction pipelining
- Real-world examples of ISAs

4

5.2 Instruction Formats

Instruction sets are differentiated by the following:

- Number of bits per instruction.
- Stack-based or register-based.
- Number of explicit operands per instruction.
- Operand location.
- Types of operations.
- Type and size of operands.

5

5.2 Instruction Formats

Instruction set architectures are measured according to:

- Main memory space occupied by a program.
- Instruction complexity.
- Instruction length (in bits).
- Total number of instructions in the instruction set.

6

5.2 Instruction Formats

In designing an instruction set, consideration is given to:

- Instruction length.
  - Whether short, long, or variable.
- Number of operands.
- Number of addressable registers.
- Memory organization.
  - Whether byte- or word- addressable.
- Addressing modes.
  - Choose any or all: direct, indirect or indexed.

7

5.2 Instruction Formats

- Byte ordering, or *endianness*, is another major architectural consideration.
- If we have a two-byte integer, the integer may be stored so that the least significant byte is followed by the most significant byte or vice versa.
  - In *little endian* machines, the least significant byte is followed by the most significant byte.
  - *Big endian* machines store the most significant byte first (at the lower address).

8

5.2 Instruction Formats

- As an example, suppose we have the hexadecimal number 0x12345678.
- The big endian and small endian arrangements of the bytes are shown below.

Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

9

5.2 Instruction Formats

- A larger example: A computer uses 32-bit integers. The values 0xABCD1234, 0x00FE4321, and 0x10 would be stored sequentially in memory, starting at address 0x200 as below.

Address	Big Endian	Little Endian
0x200	AB	34
0x201	CD	12
0x202	12	CD
0x203	34	AB
0x204	00	21
0x205	FE	43
0x206	43	FE
0x207	21	00
0x208	00	10
0x209	00	00
0x20A	00	00
0x20B	10	00

10

5.2 Instruction Formats

- Big endian:
  - Is more natural.
  - The sign of the number can be determined by looking at the byte at address offset 0.
  - Strings and integers are stored in the same order.
- Little endian:
  - Makes it easier to place values on non-word boundaries.
  - Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic.

11

5.2 Instruction Formats

- The next consideration for architecture design concerns how the CPU will store data.
- We have three choices:
  1. A stack architecture
  2. An accumulator architecture
  3. A general purpose register architecture.
- In choosing one over the other, the tradeoffs are simplicity (and cost) of hardware design with execution speed and ease of use.

12

5.2 Instruction Formats

- In a stack architecture, instructions and operands are implicitly taken from the stack.
  - A stack cannot be accessed randomly.
- In an accumulator architecture, one operand of a binary operation is implicitly in the accumulator.
  - One operand is in memory, creating lots of bus traffic.
- In a general purpose register (GPR) architecture, registers can be used instead of memory.
  - Faster than accumulator architecture.
  - Efficient implementation for compilers.
  - Results in longer instructions.

13

5.2 Instruction Formats

- Most systems today are GPR systems.
- There are three types:
  - Memory-memory where two or three operands may be in memory.
  - Register-memory where at least one operand must be in a register.
  - Load-store where no operands may be in memory.
- The number of operands and the number of available registers has a direct affect on instruction length.

14

5.2 Instruction Formats

- Stack machines use one- and zero-operand instructions.
- **`LOAD`** and **`STORE`** instructions require a single memory address operand.
- Other instructions use operands from the stack implicitly.
- **`PUSH`** and **`POP`** operations involve only the stack's top element.
- Binary instructions (e.g., **`ADD`**, **`MULT`**) use the top two items on the stack.

15

5.2 Instruction Formats

- Let's see how to evaluate an infix expression using different instruction formats.
- With a three-address ISA, (e.g.,mainframes), the infix expression,

  `Z = X × Y + W × U`

  might look like this:

```
MULT R1,X,Y
MULT R2,W,U
ADD  Z,R1,R2
```

16

5.2 Instruction Formats

- In a two-address ISA, (e.g.,Intel, Motorola), the infix expression,

  Z = X × Y + W × U

  might look like this:

```
LOAD R1,X
MULT R1,Y
LOAD R2,W
MULT R2,U
ADD  R1,R2
STORE Z,R1
```

Note: One-address ISAs usually require one operand to be a register.

17

5.2 Instruction Formats

- In a one-address ISA, like MARIE, the infix expression,

  Z = X × Y + W × U

  looks like this:

```
LOAD X
MULT Y
STORE TEMP
LOAD W
MULT U
ADD TEMP
STORE Z
```

18

5.2 Instruction Formats

- In a stack ISA, the postfix expression,

  Z = X Y × W U × +

  might look like this:

```
PUSH X
PUSH Y
MULT
PUSH W
PUSH U
MULT
ADD
POP Z
```

19

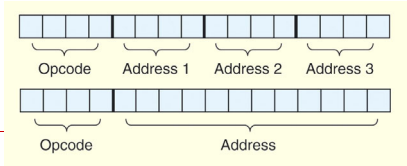
5.2 Instruction Formats

- We have seen how instruction length is affected by the number of operands supported by the ISA.
- In any instruction set, not all instructions require the same number of operands.
- Operations that require no operands, such as **HALT**, necessarily waste some space when fixed-length instructions are used.
- One way to recover some of this space is to use expanding opcodes.

20

5.2 Instruction Formats

- A system has 16 registers and 4K of memory.
- We need 4 bits to access one of the registers. We also need 12 bits for a memory address.
- If the system is to have 16-bit instructions, we have two choices for our instructions:



21

Outline

- Instruction formats
- Instruction types
- Addressing
- Instruction pipelining
- Real-world examples of ISAs

23

5.2 Instruction Formats

- If we allow the length of the opcode to vary, we could create a very rich instruction set:

```
0000 R1   R2   R3
...                    } 15 three-address codes
1110 R1   R2   R3
1111 - escape opcode
1111 0000 R1   R2
...                    } 14 two-address codes
1111 1101 R1   R2
1111 1110 - escape opcode
1111 1110 0000 R1
...                    } 31 one-address codes
1111 1111 1110 R1
1111 1111 1111 - escape opcode
1111 1111 1111 0000
...                    } 16 zero-address codes
1111 1111 1111 1111
```

5.3 Instruction types

Instructions fall into several broad categories that you should be familiar with:

- Data movement.
- Arithmetic.
- Boolean.
- Bit manipulation.
- I/O.
- Control transfer.
- Special purpose.

Can you think of some examples of each of these?

24

Outline

- Instruction formats
- Instruction types
- Addressing
- Instruction pipelining
- Real-world examples of ISAs

25

5.4 Addressing

- Addressing modes specify where an operand is located.
- They can specify a constant, a register, or a memory location.
- The actual location of an operand is its *effective address*.
- Certain addressing modes allow us to determine the address of an operand dynamically.

26

5.4 Addressing

- *Immediate addressing* is where the data is part of the instruction.
- *Direct addressing* is where the address of the data is given in the instruction.
- *Register addressing* is where the data is located in a register.
- *Indirect addressing* gives the address of the address of the data in the instruction.
- *Register indirect addressing* uses a register to store the address of the data.

27

5.4 Addressing

- *Indexed addressing* uses a register (implicitly or explicitly) as an offset, which is added to the address in the operand to determine the effective address of the data.
- *Based addressing* is similar except that a base register is used instead of an index register.
- The difference between these two is that an index register holds an offset relative to the address given in the instruction, a base register holds a base address where the address field represents a displacement from this base.

28

5.4 Addressing

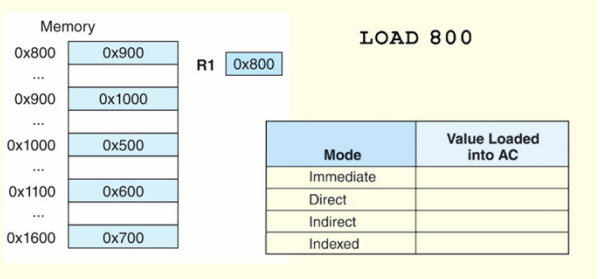
- In *stack addressing* the operand is assumed to be on top of the stack.
- There are many variations to these addressing modes including:
  - Indirect indexed.
  - Base/offset.
  - Self-relative
  - Auto increment - decrement.
- We won't cover these in detail.

Let's look at an example of the principal addressing modes.

29

5.4 Addressing

- For the instruction shown, what value is loaded into the accumulator for each addressing mode?



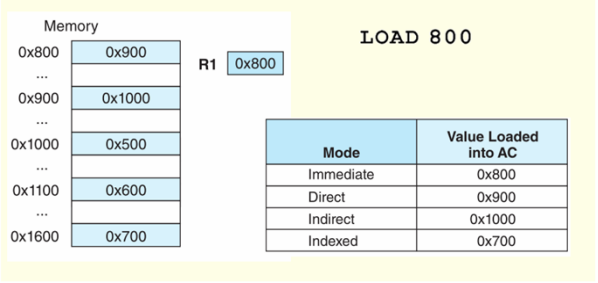
30

Outline

- Instruction formats
- Instruction types
- Addressing
- Instruction pipelining
- Real-world examples of ISAs

32

5.4 Addressing



31

5.5 Instruction Pipelining

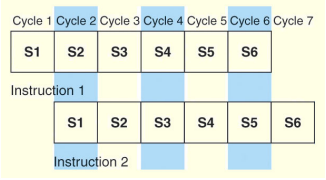
- Some CPUs divide the fetch-decode-execute cycle into smaller steps.
- These smaller steps can often be executed in parallel to increase throughput.
- Such parallel execution is called *instruction pipelining*.
- Instruction pipelining provides for *instruction level parallelism* (*ILP*)

The next slide shows an example of instruction pipelining.

33

5.5 Instruction Pipelining

- For every clock cycle, one small step is carried out, and the stages are overlapped.



S1. Fetch instruction.
S2. Decode opcode.
S3. Calculate effective address of operands.
S4. Fetch operands.
S5. Execute.
S6. Store result.

35

5.5 Instruction Pipelining

- Suppose a fetch-decode-execute cycle were broken into the following smaller steps:

1. Fetch instruction.
2. Decode opcode.
3. Calculate effective address of operands.
4. Fetch operands.
5. Execute instruction.
6. Store result.

- Suppose we have a six-stage pipeline. S1 fetches the instruction, S2 decodes it, S3 determines the address of the operands, S4 fetches them, S5 executes the instruction, and S6 stores the result.

34

5.5 Instruction Pipelining

- The theoretical speedup offered by a pipeline can be determined as follows:

  Let t_p be the time per stage. Each instruction represents a task, T, in the pipeline.

  The first task (instruction) requires $k \times t_p$ time to complete in a k-stage pipeline. The remaining $(n - 1)$ tasks emerge from the pipeline one per cycle. So the total time to complete the remaining tasks is $(n - 1)t_p$.

  Thus, to complete n tasks using a k-stage pipeline requires:

$$(k \times t_p) + (n - 1)t_p = (k + n - 1)t_p.$$

36

5.5 Instruction Pipelining

- If we take the time required to complete *n* tasks without a pipeline and divide it by the time it takes to complete *n* tasks using a pipeline, we find:

$$\text{Speedup } S = \frac{nt_n}{(k + n - 1)\, t_p}$$

- If we take the limit as *n* approaches infinity, ($k + n - 1$) approaches *n,* which results in a theoretical speedup of:

$$\text{Speedup } S = \frac{kt_p}{t_p} = k$$

37

5.5 Instruction Pipelining

- Our neat equations take a number of things for granted.
- First, we have to assume that the architecture supports fetching instructions and data in parallel.
- Second, we assume that the pipeline can be kept filled at all times. This is not always the case. Pipeline *hazards* arise that cause pipeline conflicts and stalls.

38

5.5 Instruction Pipelining

- An instruction pipeline may stall, or be flushed for any of the following reasons:
  - Resource conflicts.
  - Data dependencies.
  - Conditional branching.
- Measures can be taken at the software level as well as at the hardware level to reduce the effects of these hazards, but they cannot be totally eliminated.

39

Outline

- Instruction formats
- Instruction types
- Addressing
- Instruction pipelining
- Real-world examples of ISAs

40

5.6 Real-World Examples of ISAs

- We return briefly to the Intel and MIPS architectures from the last chapter, using some of the ideas introduced in this chapter.
- Intel introduced pipelining to their processor line with its Pentium chip.
- The first Pentium had two five-stage pipelines. Each subsequent Pentium processor had a longer pipeline than its predecessor with the Pentium IV having a 24-stage pipeline.
- The Itanium (IA-64) has only a 10-stage pipeline.

41

5.6 Real-World Examples of ISAs

- Intel processors support a wide array of addressing modes.
- The original 8086 provided 17 ways to address memory, most of them variants on the methods presented in this chapter.
- Owing to their need for backward compatibility, the Pentium chips also support these 17 addressing modes.
- The Itanium, having a RISC core, supports only one: register indirect addressing with optional post increment.

42

5.6 Real-World Examples of ISAs

- MIPS was an acronym for *Microprocessor Without Interlocked Pipeline Stages*.
- The architecture is little endian and word-addressable with three-address, fixed-length instructions.
- Like Intel, the pipeline size of the MIPS processors has grown: The R2000 and R3000 have five-stage pipelines.; the R4000 and R4400 have 8-stage pipelines.

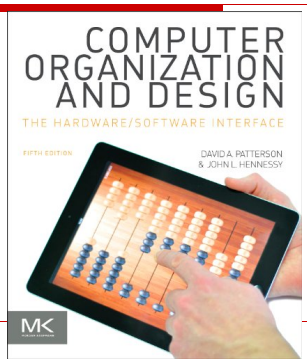
43

5.6 Real-World Examples of ISAs

- The R10000 has three pipelines: A five-stage pipeline for integer instructions, a seven-stage pipeline for floating-point instructions, and a six-state pipeline for **`LOAD/STORE`** instructions.
- In all MIPS ISAs, only the **`LOAD`** and **`STORE`** instructions can access memory.
- The ISA uses only base addressing mode.
- The assembler accommodates programmers who need to use immediate, register, direct, indirect register, base, or indexed addressing modes.

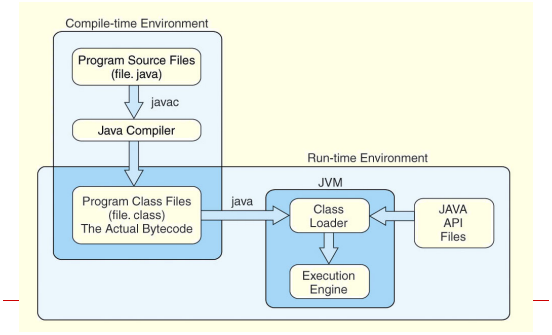
44

A Good Reference on MIPS



45

5.6 Real-World Examples of ISAs



47

5.6 Real-World Examples of ISAs

- The Java programming language is an interpreted language that runs in a software machine called the *Java Virtual Machine* (JVM).
- A JVM is written in a native language for a wide array of processors, including MIPS and Intel.
- Like a real machine, the JVM has an ISA all of its own, called *bytecode*. This ISA was designed to be compatible with the architecture of any machine on which the JVM is running.

The next slide shows how the pieces fit together.

46

5.6 Real-World Examples of ISAs

- Java bytecode is a stack-based language.
- Most instructions are zero address instructions.
- The JVM has four registers that provide access to five regions of main memory.
- All references to memory are offsets from these registers. Java uses no pointers or absolute memory references.
- Java was designed for platform interoperability, not performance!

48

5.6 Real-World Examples of ISAs

- You may not have heard of ARM but most likely use an ARM processor every day. It is the most widely used 32-bit instruction architecture:
  - 95%+ of smartphones
  - 80%+ of digital cameras
  - 40%+ of all digital television sets
- Founded in 1990, by Apple and others, ARM (Advanced RISC Machine) is now a British firm, ARM Holdings.
- ARM Holdings does not manufacture these processors; it sells licenses to manufacture.

49

5.6 Real-World Examples of ISAs

- ARM is a load/store architecture: all data processing must be performed on values in registers, not in memory.
- It uses fixed-length, three-operand instructions and simple addressing modes
- ARM processors have a minimum of a three-stage pipeline (consisting of fetch, decode, and execute);
  - Newer ARM processors have deeper pipelines (more stages). Some ARM8 implementations have 13-stage integer pipelines

50

5.6 Real-World Examples of ISAs

- ARM has 37 total registers but their visibility depends on the processor mode.
- ARM allows multiple register transfers.
  - It can simultaneously load or store any subset of the16 general-purpose registers from/to sequential memory addresses.
- Control flow instructions include unconditional and conditional branching and procedure calls
- Most ARM instructions execute in a single cycle, provided there are no pipeline hazards or memory accesses.

51

Chapter 5 Conclusion

- ISAs are distinguished according to their bits per instruction, number of operands per instruction, operand location and types and sizes of operands.
- Endianness as another major architectural consideration.
- CPU can store store data based on
  1. A stack architecture
  2. An accumulator architecture
  3. A general purpose register architecture.

52

Chapter 5 Conclusion

- Instructions can be fixed length or variable length.
- To enrich the instruction set for a fixed length instruction set, expanding opcodes can be used.
- The addressing mode of an ISA is also another important factor. We looked at:
  - Immediate – Direct
  - Register – Register Indirect
  - Indirect – Indexed
  - Based – Stack

53

Chapter 5 Conclusion

- A *k*-stage pipeline can theoretically produce execution speedup of *k* as compared to a non-pipelined machine.
- Pipeline hazards such as resource conflicts and conditional branching prevents this speedup from being achieved in practice.
- The Intel, MIPS, JVM, and ARM architectures provide good examples of the concepts presented in this chapter.

54

End of Chapter 5

55