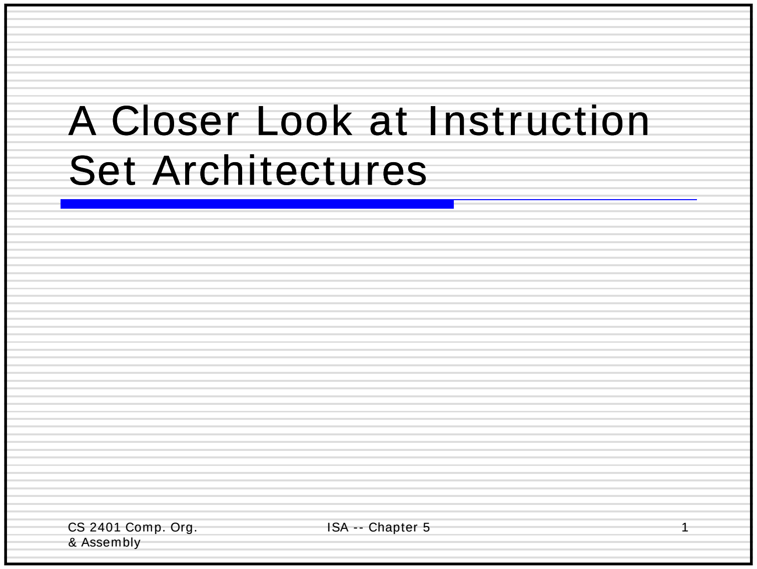
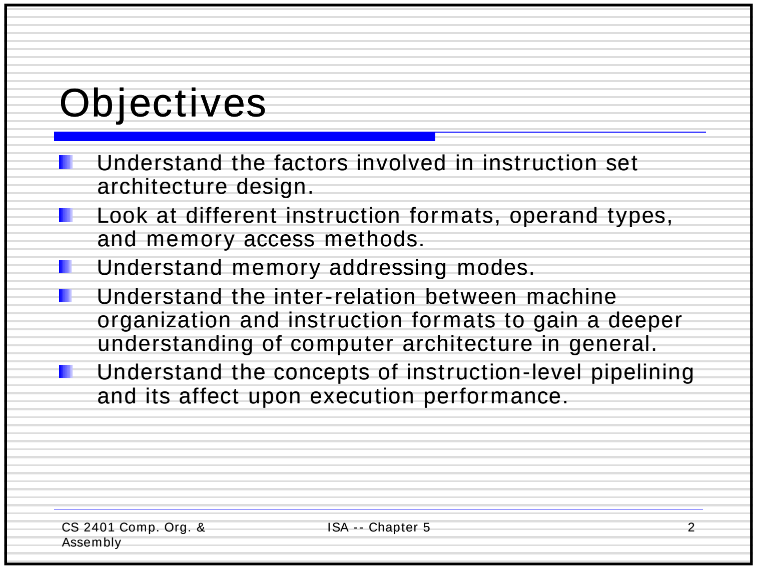
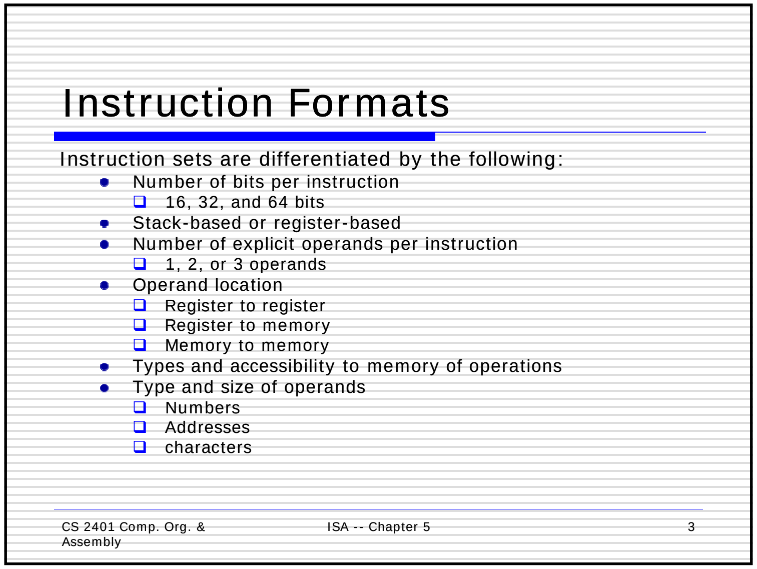


A Closer Look at Instruction Set Architectures



Objectives

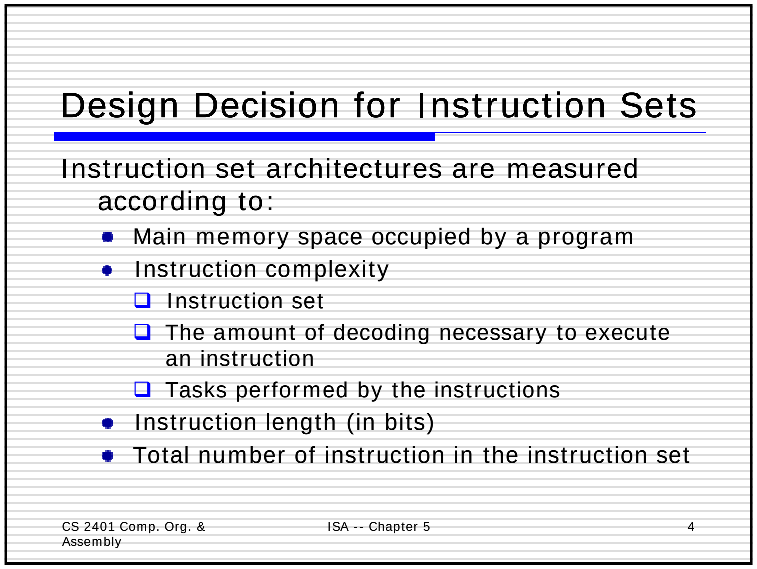
- Understand the factors involved in instruction set architecture design.
- Look at different instruction formats, operand types, and memory access methods.
- Understand memory addressing modes.
- Understand the inter-relation between machine organization and instruction formats to gain a deeper understanding of computer architecture in general.
- Understand the concepts of instruction-level pipelining and its affect upon execution performance.



Instruction Formats

Instruction sets are differentiated by the following:

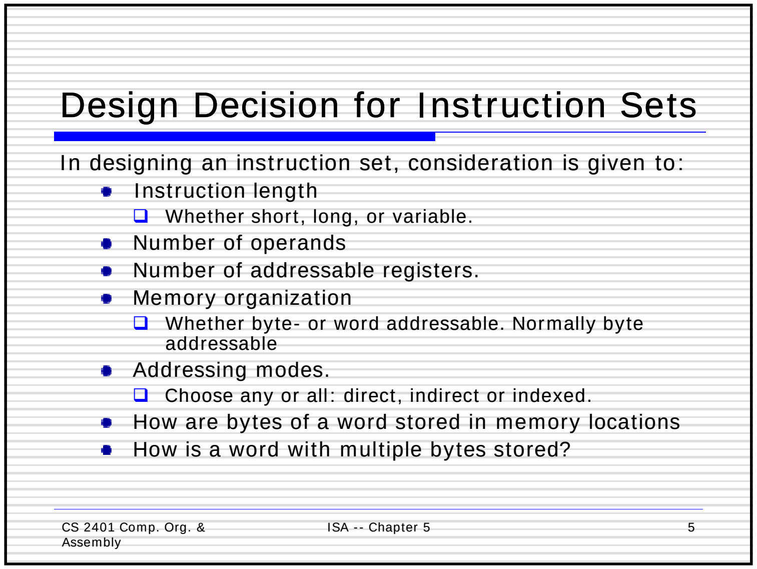
- Number of bits per instruction
  - 16, 32, and 64 bits
- Stack-based or register-based
- Number of explicit operands per instruction
  - 1, 2, or 3 operands
- Operand location
  - Register to register
  - Register to memory
  - Memory to memory
- Types and accessibility to memory of operations
- Type and size of operands
  - Numbers
  - Addresses
  - characters



Design Decision for Instruction Sets

Instruction set architectures are measured according to:

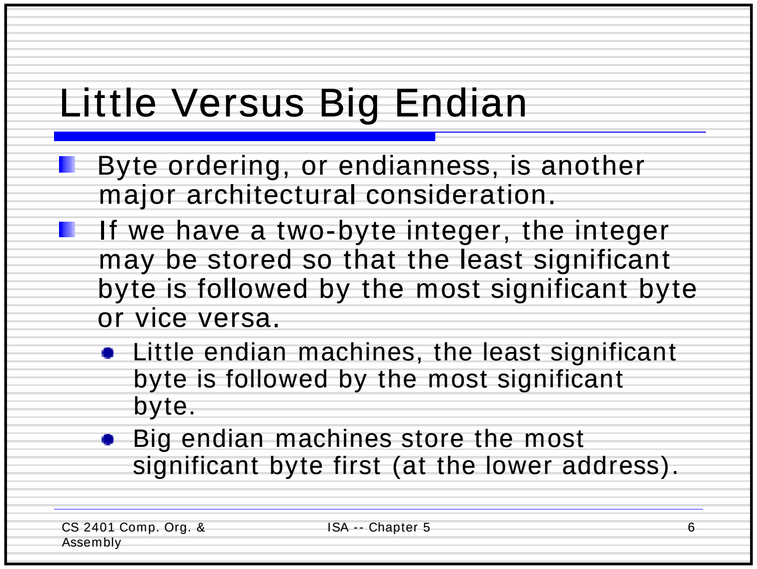
- Main memory space occupied by a program
- Instruction complexity
  - Instruction set
  - The amount of decoding necessary to execute an instruction
  - Tasks performed by the instructions
- Instruction length (in bits)
- Total number of instruction in the instruction set



Design Decision for Instruction Sets

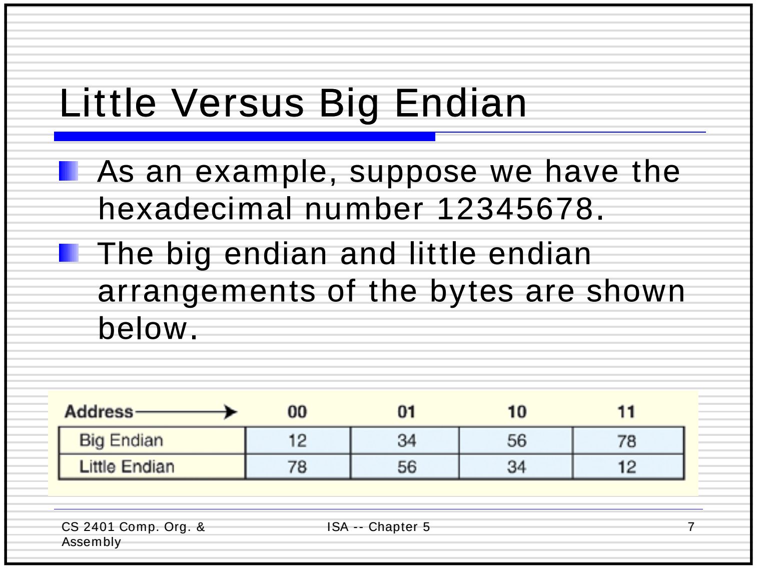
In designing an instruction set, consideration is given to:

- Instruction length
  - ❑ Whether short, long, or variable.
- Number of operands
- Number of addressable registers.
- Memory organization
  - ❑ Whether byte- or word addressable. Normally byte addressable
- Addressing modes.
  - ❑ Choose any or all: direct, indirect or indexed.
- How are bytes of a word stored in memory locations
- How is a word with multiple bytes stored?



Little Versus Big Endian

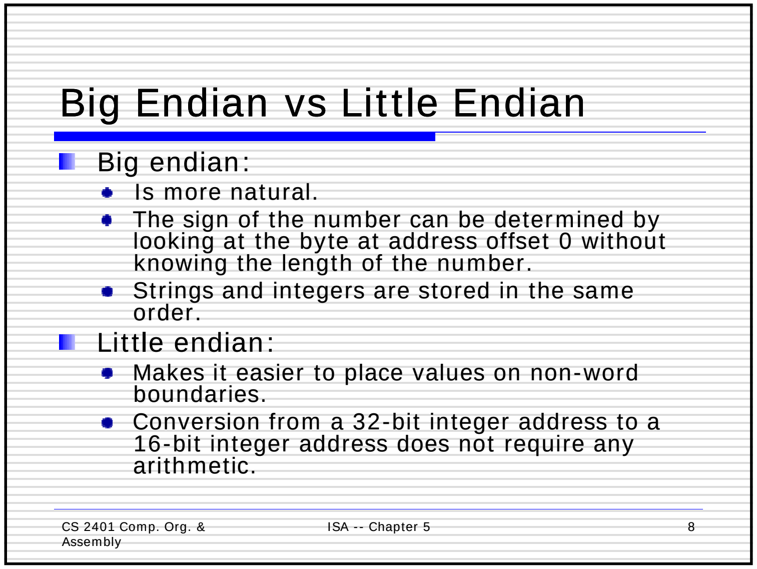
- Byte ordering, or endianness, is another major architectural consideration.
- If we have a two-byte integer, the integer may be stored so that the least significant byte is followed by the most significant byte or vice versa.
  - Little endian machines, the least significant byte is followed by the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



Little Versus Big Endian

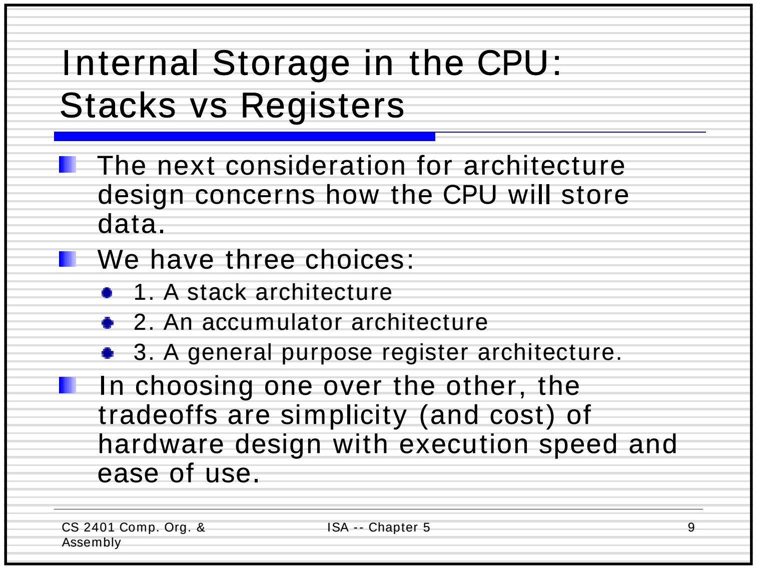
- As an example, suppose we have the hexadecimal number 12345678.
- The big endian and little endian arrangements of the bytes are shown below.

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



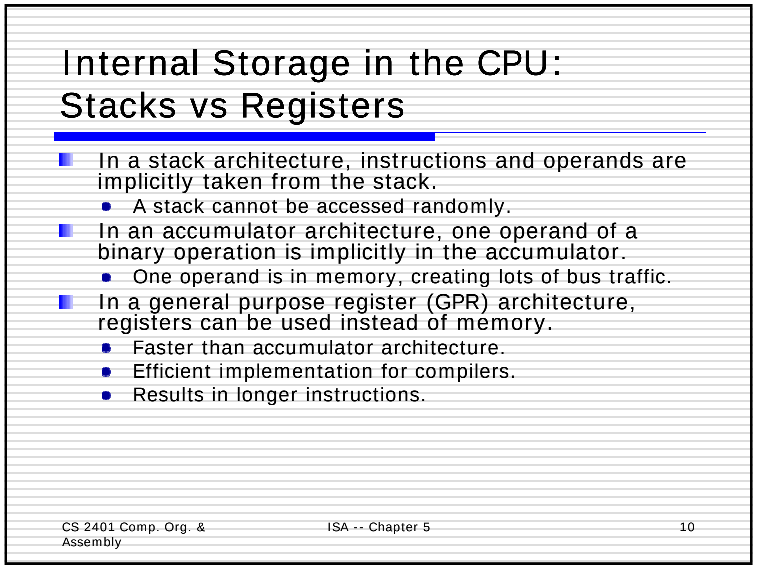
Big Endian vs Little Endian

- Big endian:
  - Is more natural.
  - The sign of the number can be determined by looking at the byte at address offset 0 without knowing the length of the number.
  - Strings and integers are stored in the same order.
- Little endian:
  - Makes it easier to place values on non-word boundaries.
  - Conversion from a 32-bit integer address to a 16-bit integer address does not require any arithmetic.



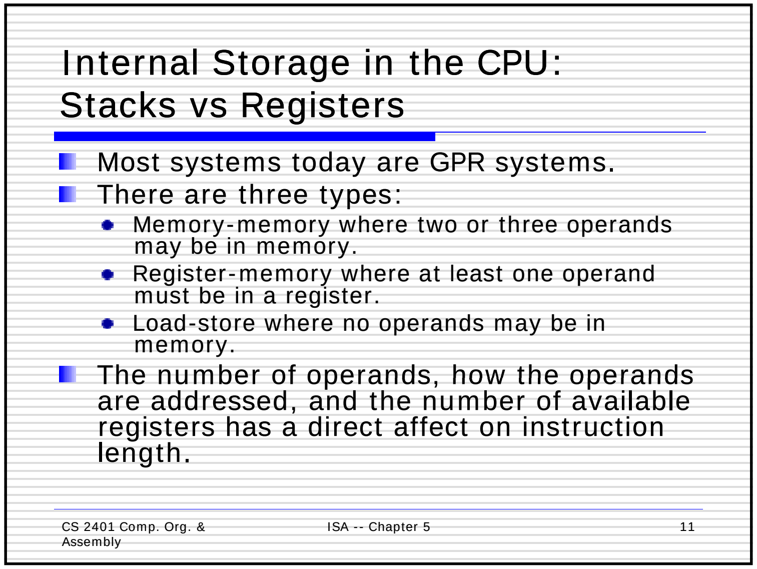
Internal Storage in the CPU: Stacks vs Registers

- 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.



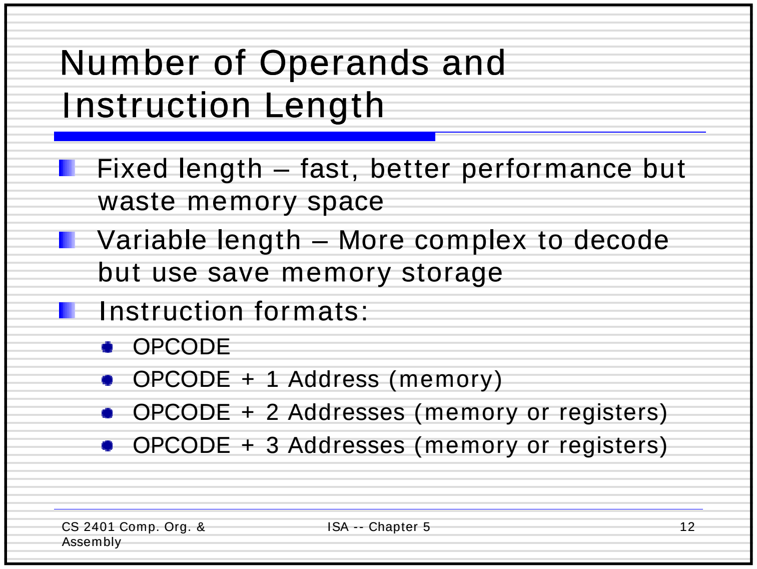
Internal Storage in the CPU: Stacks vs Registers

- 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.



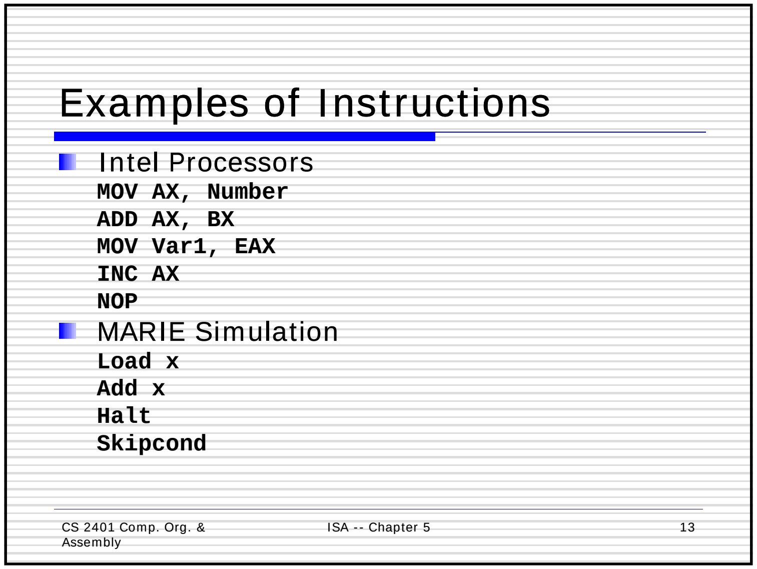
Internal Storage in the CPU: Stacks vs Registers

- 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, how the operands are addressed, and the number of available registers has a direct affect on instruction length.



Number of Operands and Instruction Length

- Fixed length – fast, better performance but waste memory space
- Variable length – More complex to decode but use save memory storage
- Instruction formats:
  - OPCODE
  - OPCODE + 1 Address (memory)
  - OPCODE + 2 Addresses (memory or registers)
  - OPCODE + 3 Addresses (memory or registers)



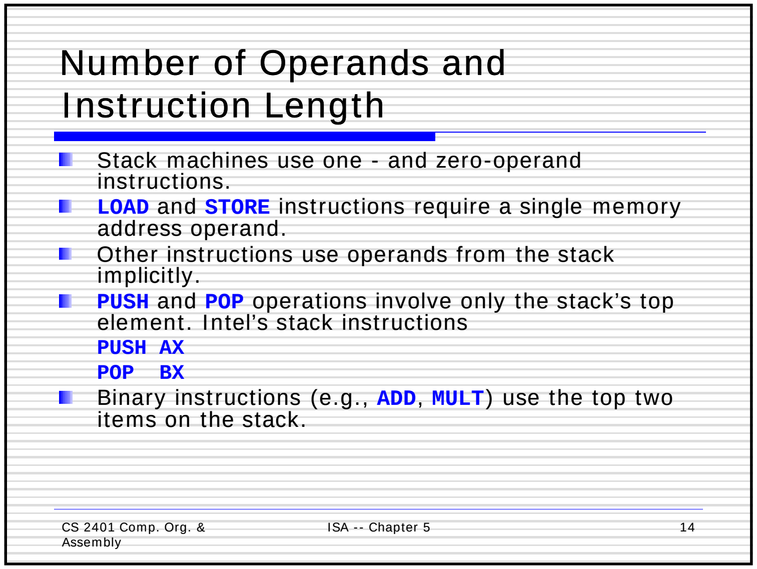
Examples of Instructions

- Intel Processors

  ```
  MOV AX, Number
  ADD AX, BX
  MOV Var1, EAX
  INC AX
  NOP
  ```

- MARIE Simulation

  ```
  Load x
  Add x
  Halt
  Skipcond
  ```

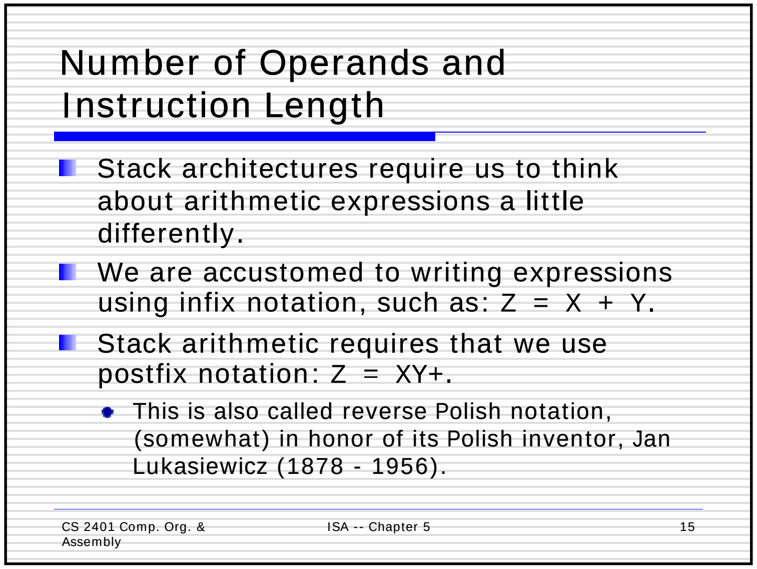


Number of Operands and Instruction Length

- 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. Intel's stack instructions

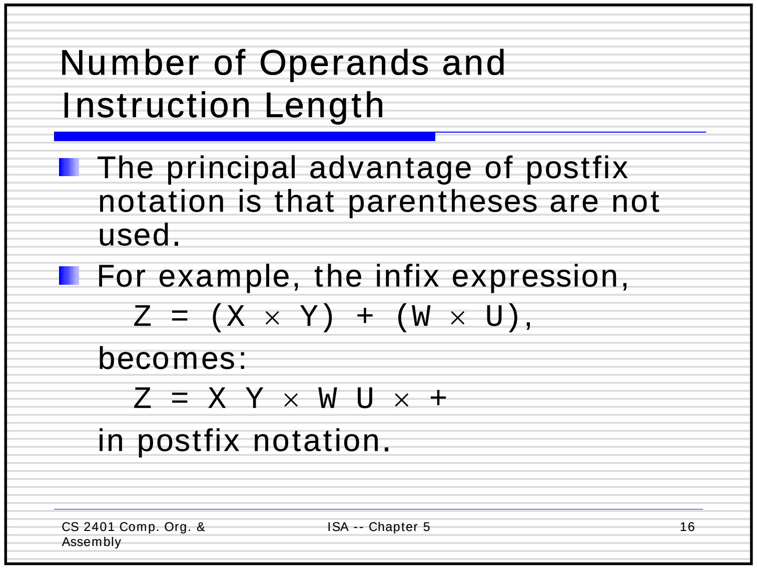
  ```
  PUSH AX
  POP  BX
  ```

- Binary instructions (e.g., **ADD**, **MULT**) use the top two items on the stack.



Number of Operands and Instruction Length

- Stack architectures require us to think about arithmetic expressions a little differently.
- We are accustomed to writing expressions using infix notation, such as: Z = X + Y.
- Stack arithmetic requires that we use postfix notation: Z = XY+.
  - This is also called reverse Polish notation, (somewhat) in honor of its Polish inventor, Jan Lukasiewicz (1878 - 1956).



Number of Operands and Instruction Length

- The principal advantage of postfix notation is that parentheses are not used.
- For example, the infix expression,

  Z = (X × Y) + (W × U),

  becomes:

  Z = X Y × W U × +

  in postfix notation.

Examples

- Convert the following expressions from infix to reverse Polish (postfix) notation,

```
 W * (U * V + Z),
(U*(X+Y))/(X+Y*(U*V))
```

becomes:

```
W U V * Z + *
U X Y + * X Y U V * * + /
```

in postfix notation.

Examples

- Convert the following expression from infix to reverse Polish (postfix) notation,

$$X = \frac{A - B + C \times (D \times E - F)}{G + H \times K}$$

becomes:

```
A B – C D E * F - * + G H K * + /
```

in postfix notation.

Number of Operands and Instruction Length

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
```

Note: The result of a binary operation is implicitly stored on the top of the stack!

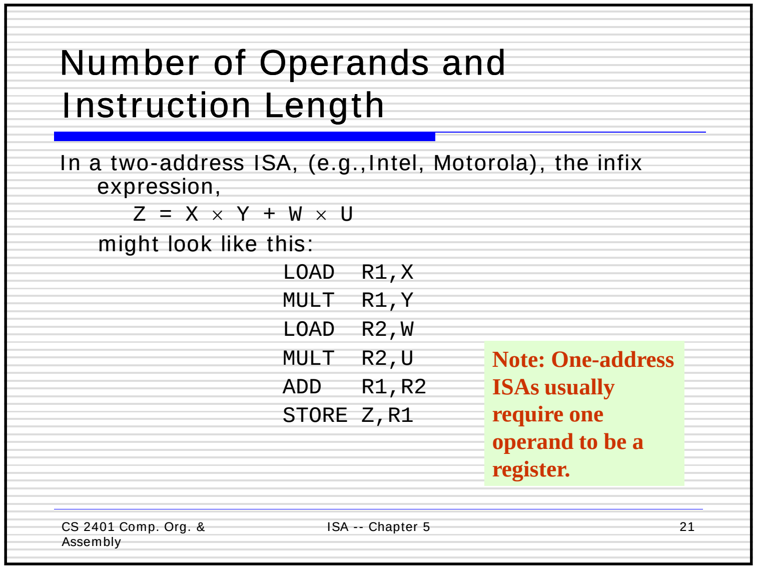
Number of Operands and Instruction Length

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
```



Number of Operands and Instruction Length

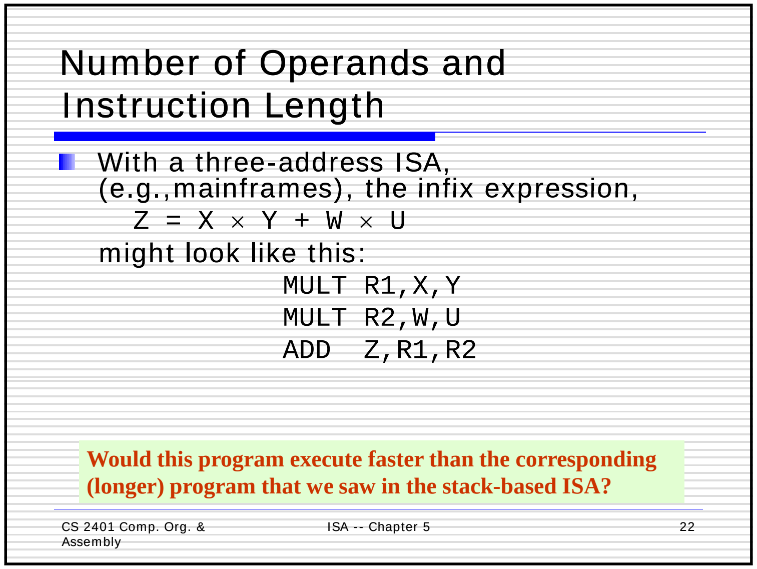
In a two-address ISA, (e.g.,Intel, Motorola), the infix expression,

```
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.



Number of Operands and Instruction Length

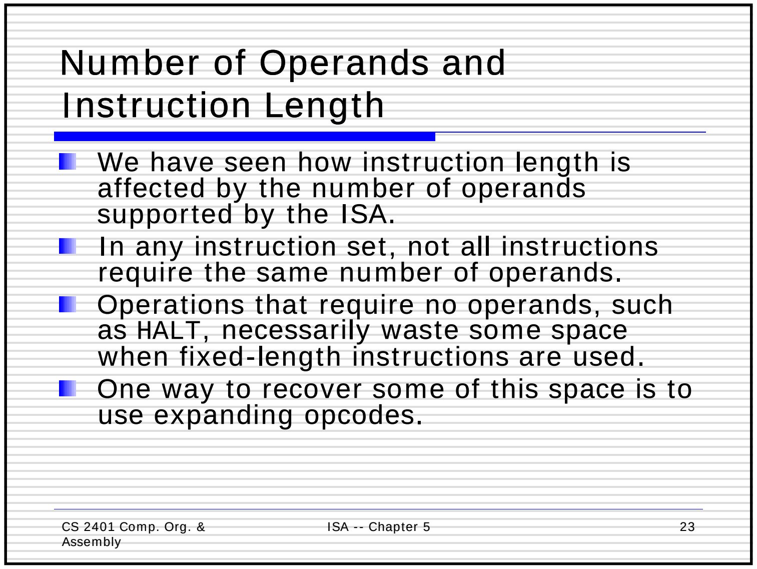
- With a three-address ISA, (e.g.,mainframes), the infix expression,

```
Z = X × Y + W × U
```

  might look like this:

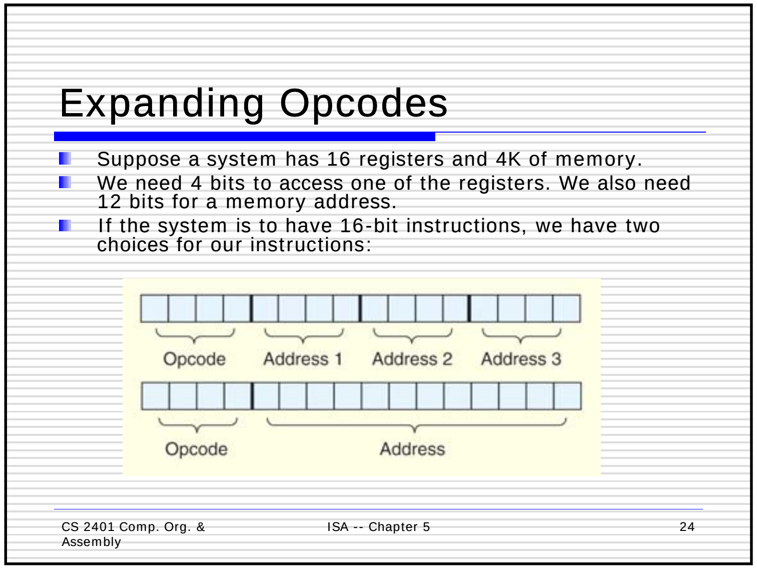
```
MULT R1,X,Y
MULT R2,W,U
ADD  Z,R1,R2
```

Would this program execute faster than the corresponding (longer) program that we saw in the stack-based ISA?



Number of Operands and Instruction Length

- 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.



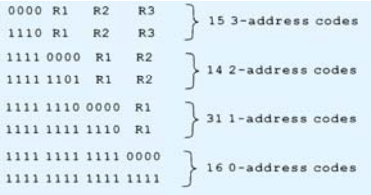
Expanding Opcodes

- Suppose 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:

Expanding Opcodes

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

- 15 instructions with 3 addresses
- 14 instructions with 2 addresses
- 31 instructions with 1 address
- 16 instructions with 0 address



```
0000  R1    R2    R3      }
1110  R1    R2    R3      } 15 3-address codes

1111 0000   R1    R2      }
1111 1101   R1    R2      } 14 2-address codes

1111 1110 0000   R1       }
1111 1111 1110   R1       } 31 1-address codes

1111 1111 1111 0000       }
1111 1111 1111 1111       } 16 0-address codes
```

Instruction types

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

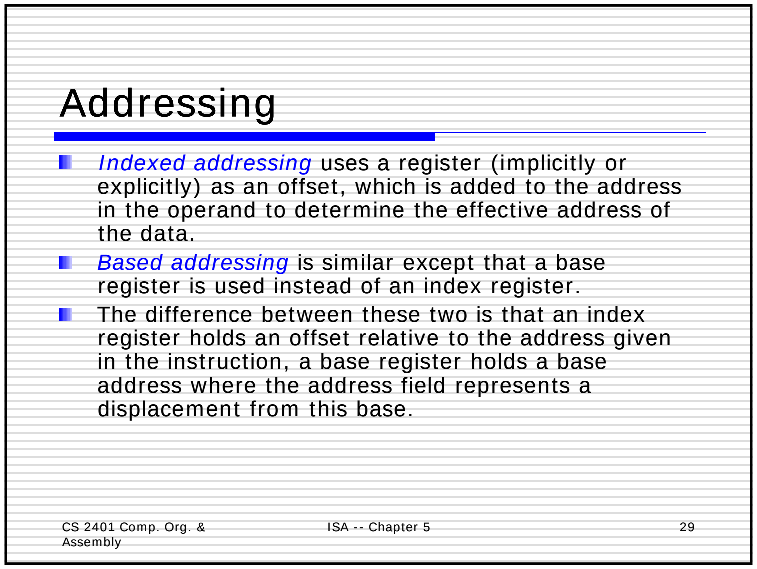
- Data movement -- **MOV, LEA**
- Arithmetic -- **ADD, MUL**
- Boolean -- **AND, XOR**
- Bit manipulation -- **SHR, RCR**
- I/O -- **IN, OUT**
- Control transfer -- **JMP, JLE**
- Special purpose -- **HLT, NOP**

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.

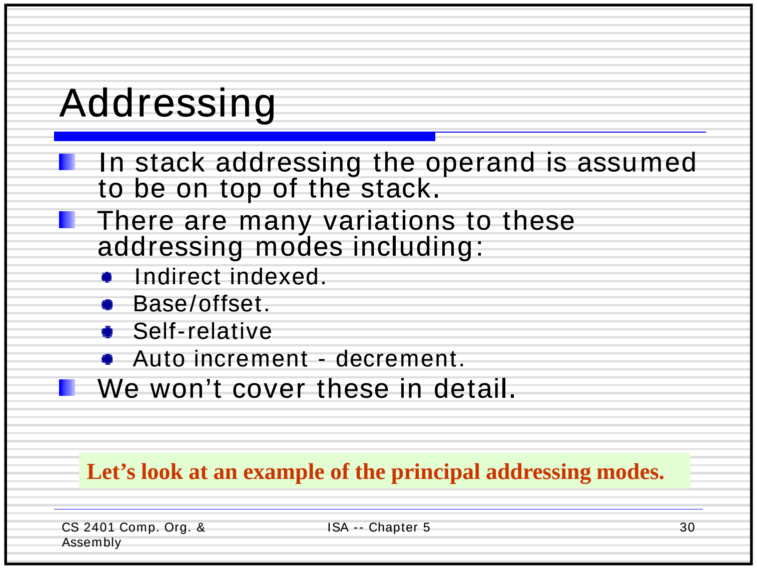
Addressing

- *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 address of the data.



Addressing

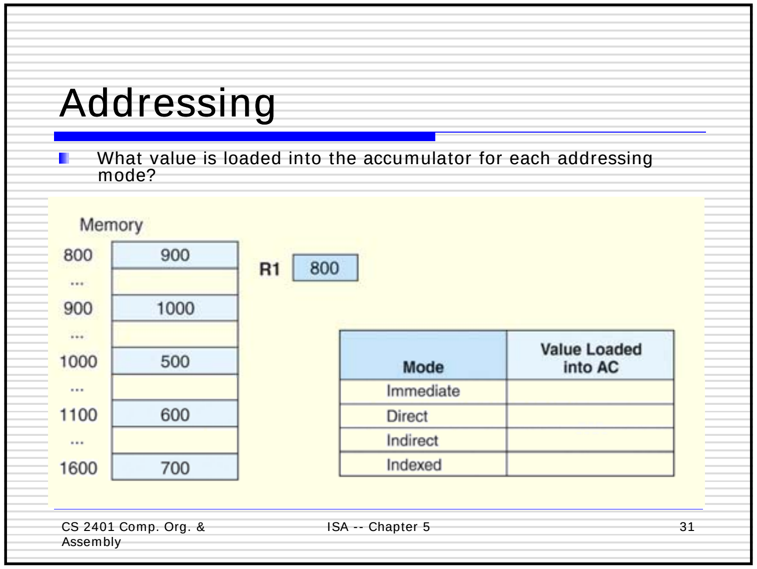
- *Indexed addressing* uses a register (implicitly or explicitly) as an offset, which is added to the address in the operand to determine the effective address of the data.
- *Based addressing* is similar except that a base register is used instead of an index register.
- The difference between these 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.



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.



Addressing

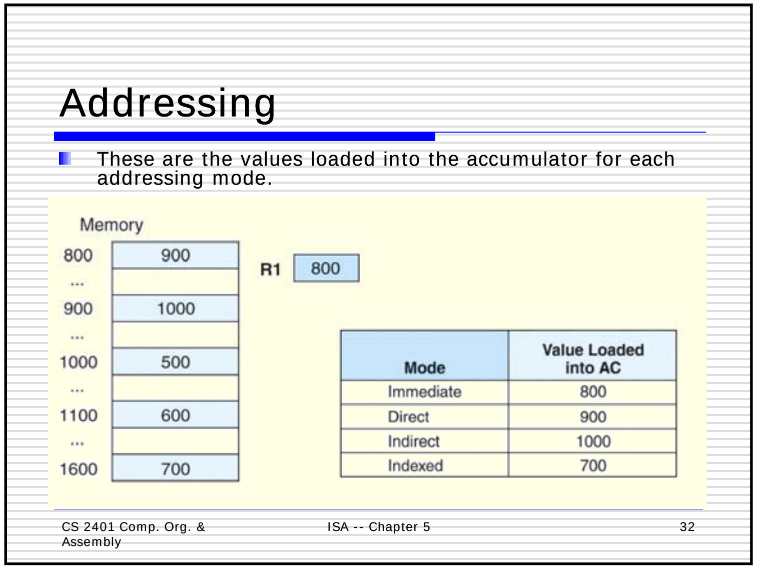
- What value is loaded into the accumulator for each addressing mode?

Memory

800	900
...	
900	1000
...	
1000	500
...	
1100	600
...	
1600	700

R1 800

Mode	Value Loaded into AC
Immediate	
Direct	
Indirect	
Indexed	



Addressing

- These are the values loaded into the accumulator for each addressing mode.

Memory

800	900
...	
900	1000
...	
1000	500
...	
1100	600
...	
1600	700

R1 800

Mode	Value Loaded into AC
Immediate	800
Direct	900
Indirect	1000
Indexed	700

Instruction-Level 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-level pipelining.
- This term is sometimes abbreviated ILP in the literature.

The next slide shows an example of instruction-level pipelining.

Instruction-Level Pipelining

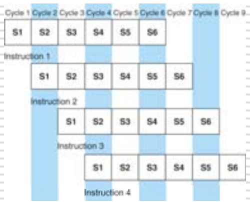
- Suppose a fetch-decode-execute cycle were broken into the following smaller steps:

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

Instruction-Level Pipelining

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



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

Instruction-Level 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:

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

Instruction-Level 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:

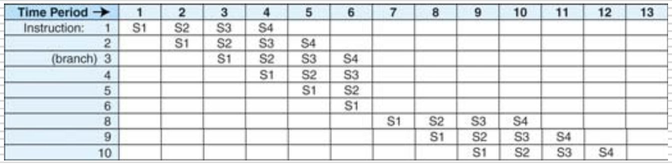
$$\text{Speedup } S = \frac{kt_p}{t_p} = k$$

Instruction-Level 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.

Instruction-Level Pipelining

I3: conditional branch to I8



Time Period →	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction: 1	S1	S2	S3	S4									
2		S1	S2	S3	S4								
(branch) 3			S1	S2	S3	S4							
4				S1	S2	S3							
5					S1	S2							
6						S1							
8							S1	S2	S3	S4			
9								S1	S2	S3	S4		
10									S1	S2	S3	S4	

Instruction-Level 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.

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.

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.

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.

Real-World Examples of ISAs

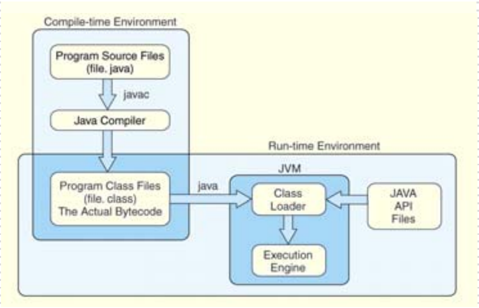
- The R10000 has three pipelines: A five-stage pipeline for integer instructions, a seven-stage pipeline for floating-point instructions, and a six-state pipeline for LOAD/STORE instructions.
- In all MIPS ISAs, only the LOAD and STORE instructions can access memory.
- The ISA uses only base addressing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

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.

Real-World Examples of ISAs

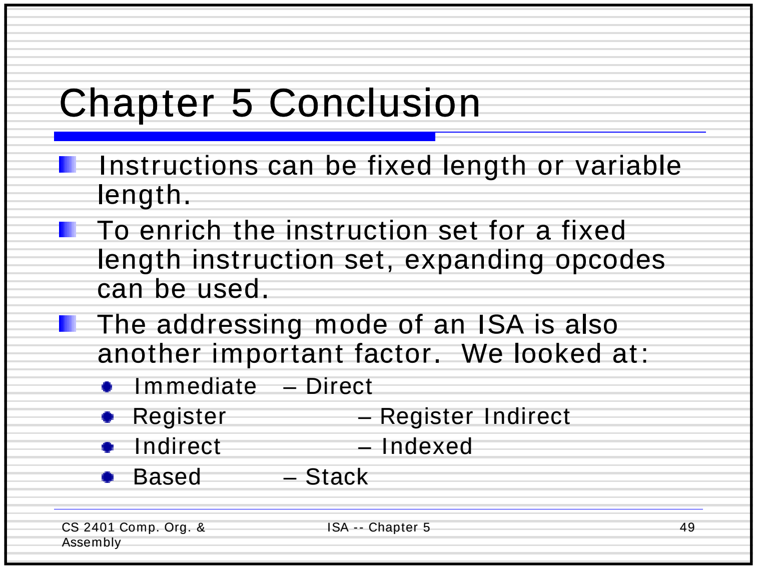


Real-World Examples of ISAs

- 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!

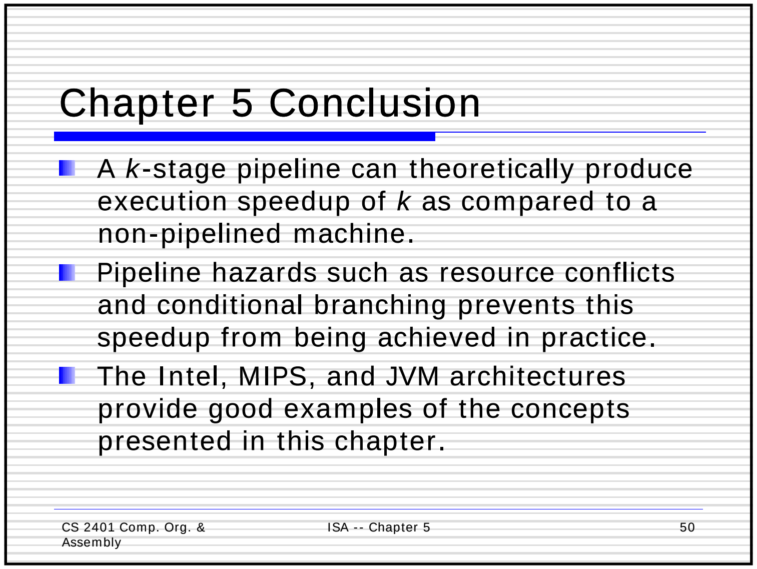
Chapter 5 Conclusion

- 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.



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



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, and JVM architectures provide good examples of the concepts presented in this chapter.