THE ESSENTIALS OF

Computer Organization and Architecture

THIRD EDITION

Linda Null
Julia Lobur

Chapter 5

A Closer Look at Instruction Set Architectures

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.

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.

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.

Say you have 8 instructions in total, therefore you need at least 3 bits to represent 8 different instructions

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.

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**.

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).

5.2 Instruction Formats

- As an example, suppose we have the **hexadecimal** number 12345678.
- 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

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.

Appending zeros at the next 16 bits

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.

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.

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.

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.

5.2 Instruction Formats

- 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).

5.2 Instruction Formats

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

$$Z = (X \times Y) + (W \times U),$$

becomes:

$$Z = X\ Y \times W\ U \times +$$

in postfix notation.

5.2 Instruction Formats

- Example: Convert the infix expression (2+3) - 6/3 to postfix:

2 3+ - 6/3	The sum 2 + 3 in parentheses takes precedence; we replace the term with 2 3 +.

5.2 Instruction Formats

- Example: Convert the infix expression (2+3) - 6/3 to postfix:

2 3+ - 6 3/	The division operator takes next precedence; we replace 6/3 with 6 3 /.

5.2 Instruction Formats

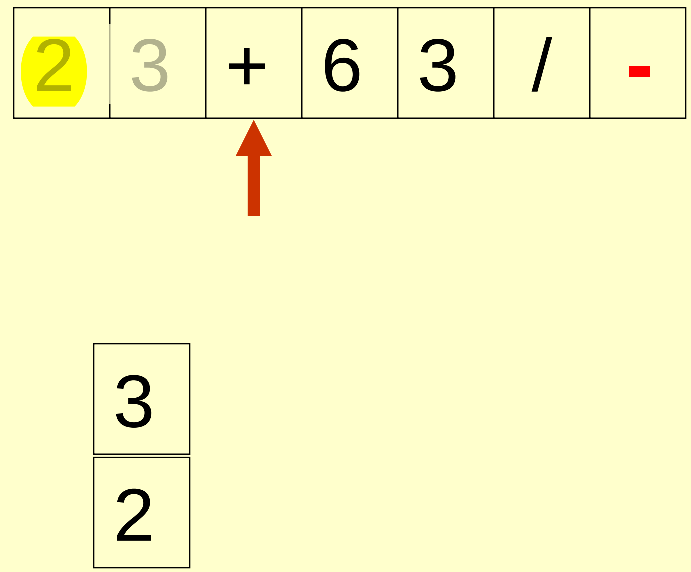
- Example: Convert the infix expression (2+3) - 6/3 to postfix:

2 3+ 6 3/ -	The quotient 6/3 is subtracted from the sum of 2 + 3, so we move the - operator to the end.

5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

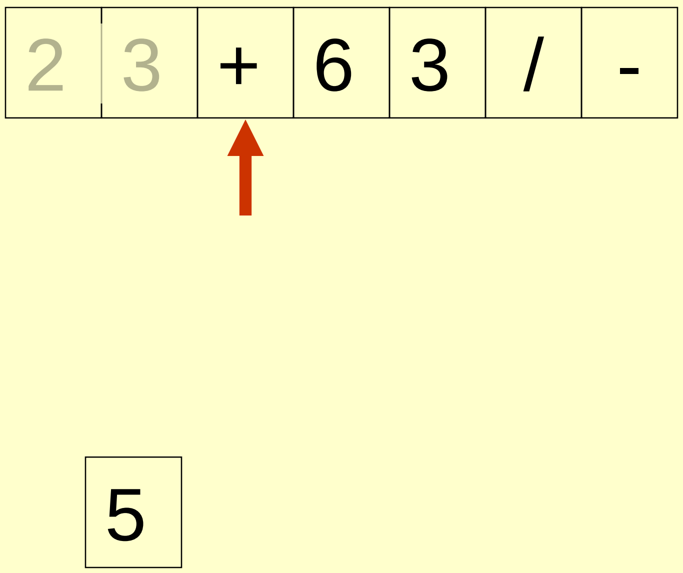
Scanning the expression from **left** to **right**, push operands onto the stack, until an operator is found



5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

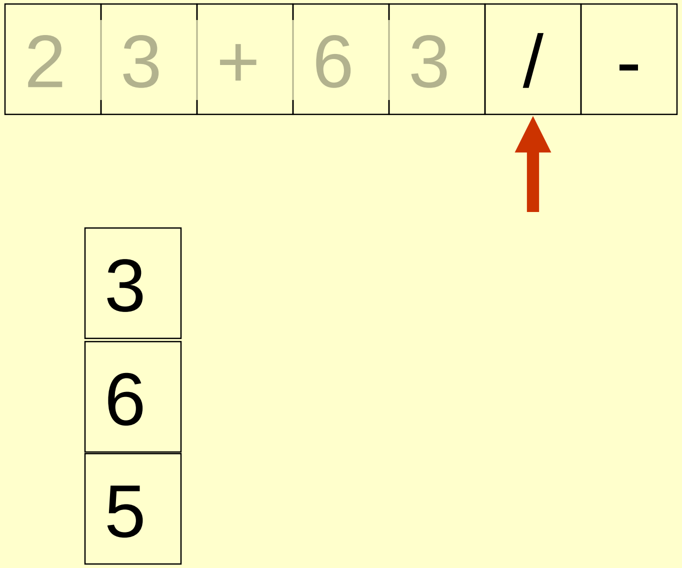
Pop the two operands and carry out the operation indicated by the operator. Push the result back on the stack.



5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

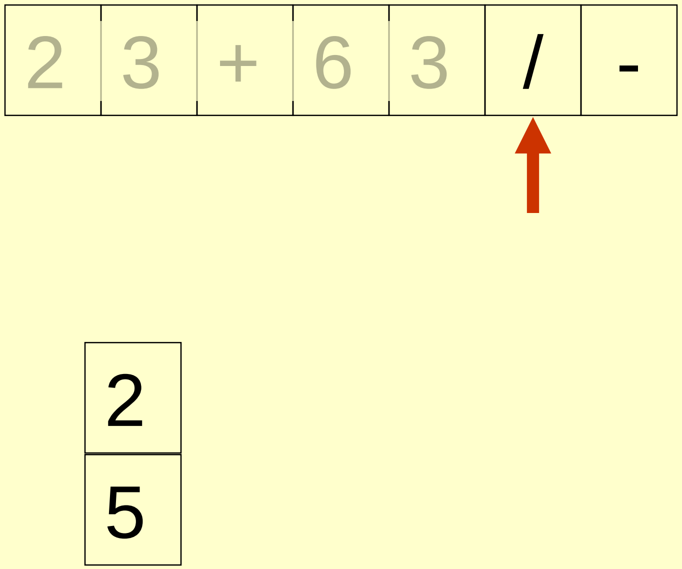
Push operands until another operator is found.



5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

Carry out the operation and push the result.

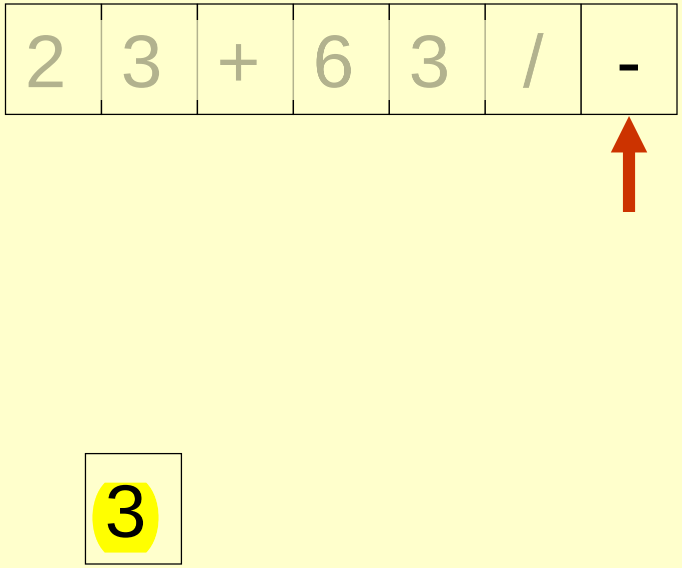


5.2 Instruction Formats

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

Finding another operator, carry out the operation and push the result.

The answer is at the top of the stack.



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
```

5.2 Instruction Formats

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

  $$Z = X \times Y + W \times U$$

  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.

5.2 Instruction Formats

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

  Z = X × Y + W × U

  looks like this:

```
LOAD X        Load X in accumlator
MULT Y        Multply Y by acc. content and store result in acc.
STORE TEMP    Store result of acc. in TEMP
LOAD W        Load W in acc.
MULT U        Multply U by acc. content and store result in acc.
ADD TEMP      Add acc. content to TEMP and store result in acc.
STORE Z       Store content of acc. in Z
```

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
PUSH Z
```

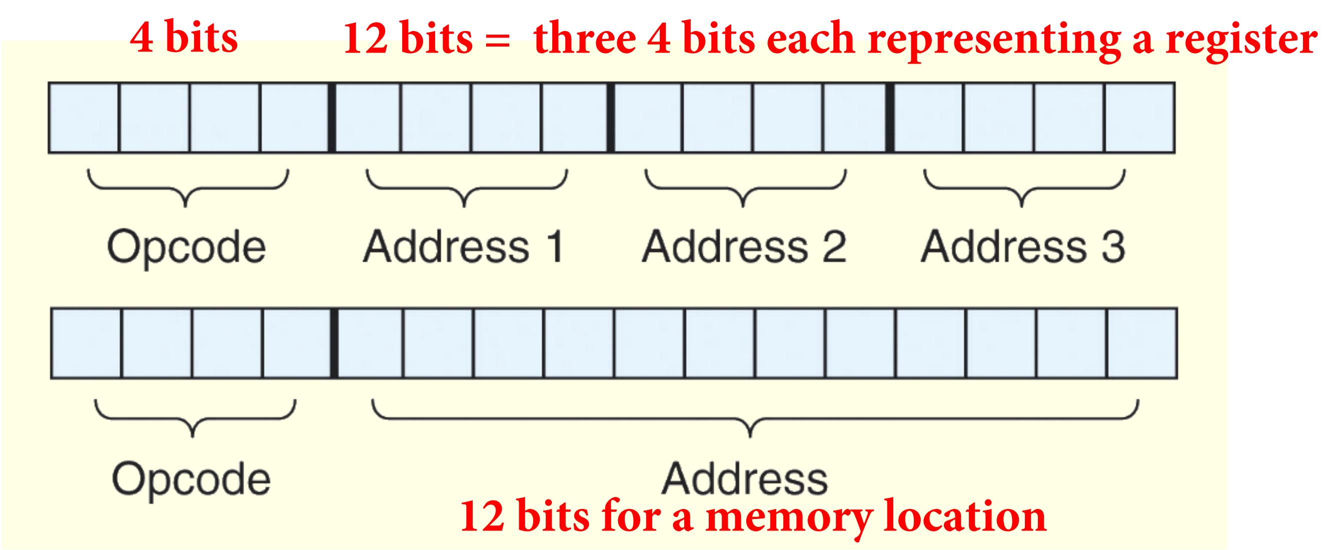
Would this program require more execution time than the corresponding (shorter) program that we saw in the 3-address ISA?

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.

5.2 Instruction Formats

- A system has 16 registers and 4K of memory. $2^4 = 16$
- We need 4 bits to access one of the registers. We also need 12 bits for a memory address. $2^{12} = 4$ K memory
- If the system is to have 16-bit instructions, we have two choices for our instructions:



5.2 Instruction Formats

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

15 op. codes requiring 4 bits

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

31 op. codes requiring 5 bits since 2^5= 31

Is there something missing from this instruction set?

5.2 Instruction Formats

- Example: Given 8-bit instructions, is it possible to allow the following to be encoded?
  - 3 instructions with two 3-bit operands.
  - 2 instructions with one 4-bit operand.
  - 4 instructions with one 3-bit operand.

We need:

$3 \times 2^3 = 192$ bits for the 3-bit operands

$2 \times 2^4 = 32$ bits for the 4-bit operands

$4 \times 2^3 = 32$ bits for the 3-bit operands.

Total: 256 bits.

5.2 Instruction Formats

- With a total of 256 bits required, we can exactly encode our instruction set in 8 bits! $2^8 = 256$

We need:

$3 \times 2^3 = 192$ bits for the 3-bit operands

$2 \times 2^4 = 32$ bits for the 4-bit operands

$4 \times 2^3 = 32$ bits for the 3-bit operands.

Total: 256 bits.

One such encoding is shown on the next slide.

5.2 Instruction Formats

`00 xxx xxx`	3 instructions with two 3-bit operands
`01 xxx xxx`	
`10 xxx xxx`	
`11 - escape opcode`	
`1100 xxxx`	2 instructions with one 4-bit operand
`1101 xxxx`	
`1110 - escape opcode`	
`1111 - escape opcode`	
`11100 xxx`	4 instructions with one 3-bit operand
`11101 xxx`	
`11110 xxx`	
`11111 xxx`	

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?

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.

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 address of the data.

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.

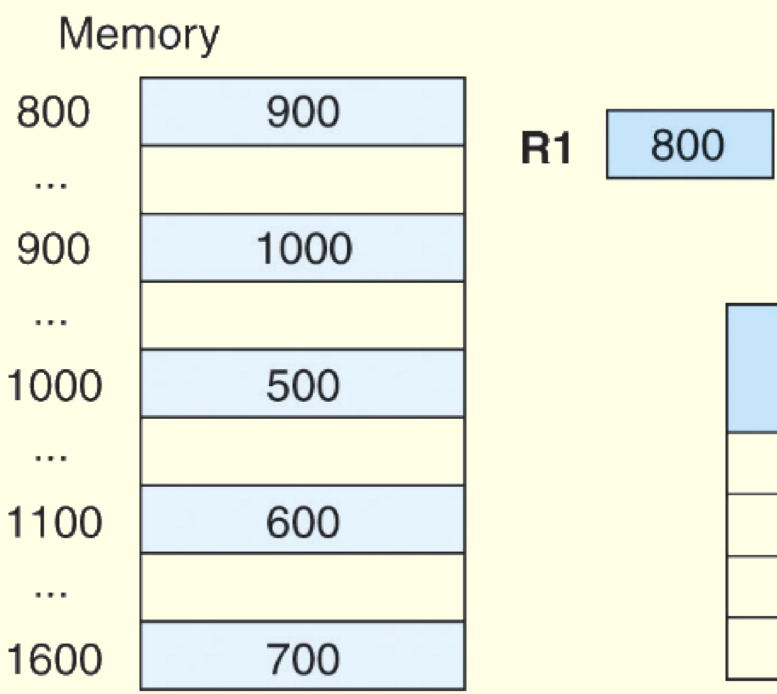
5.4 Addressing

- 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.

5.4 Addressing

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



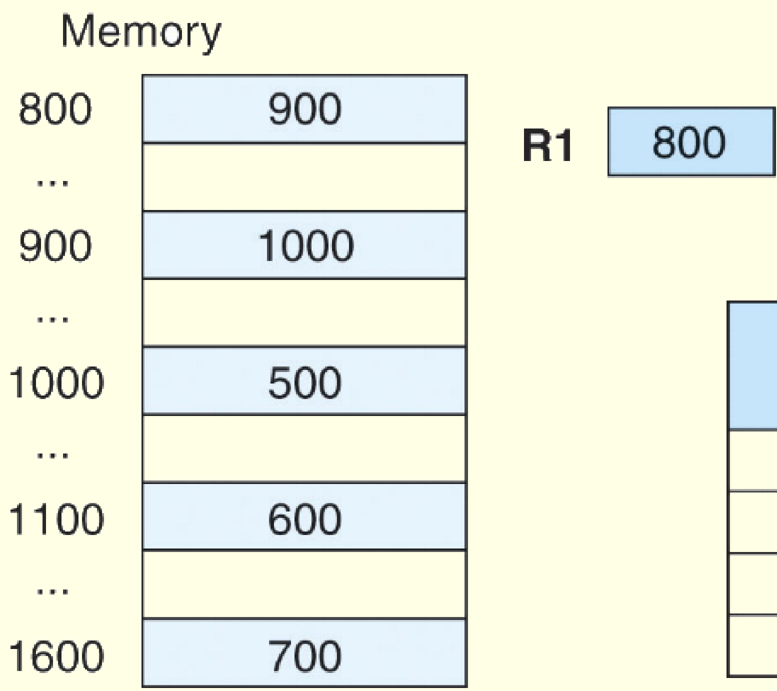
`LOAD 800`

AC is the accumlator

Mode	Value Loaded into AC
Immediate	
Direct	
Indirect	
Indexed	

5.4 Addressing

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



LOAD 800

Mode	Value Loaded into AC
Immediate	800
Direct	900 (2)
Indirect	1000 (3)
Indexed	700 (4)

(2) Go to Location 800 in MM
(3)Go to 800 , see the content which is 900 then use location 900 for its content and use it
(4) Use (3) plus some offset to get to the final oprand

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.

5.5 Instruction Pipelining

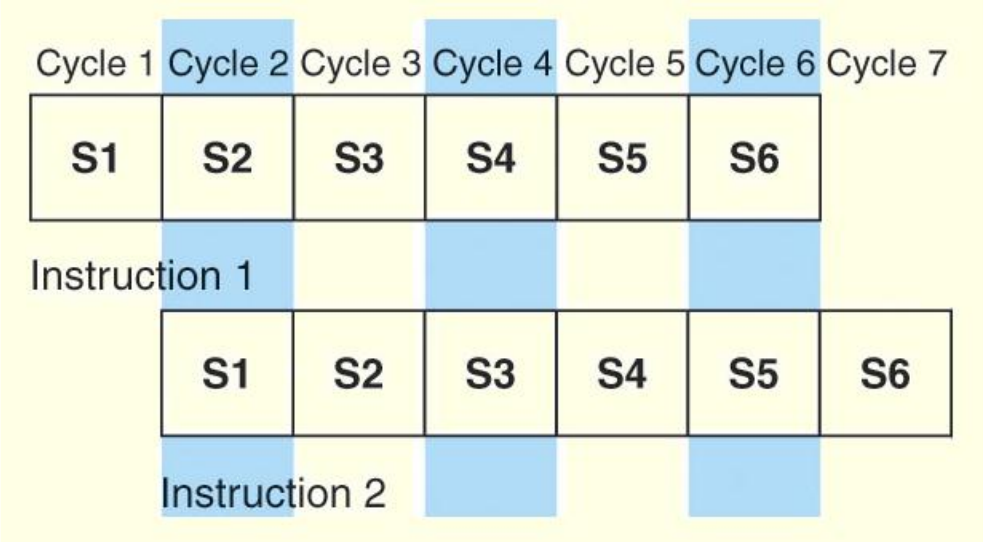
- Suppose a fetch-decode-execute cycle were broken into the following smaller steps:

  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.

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.

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:

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

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:

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

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

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

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.

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.