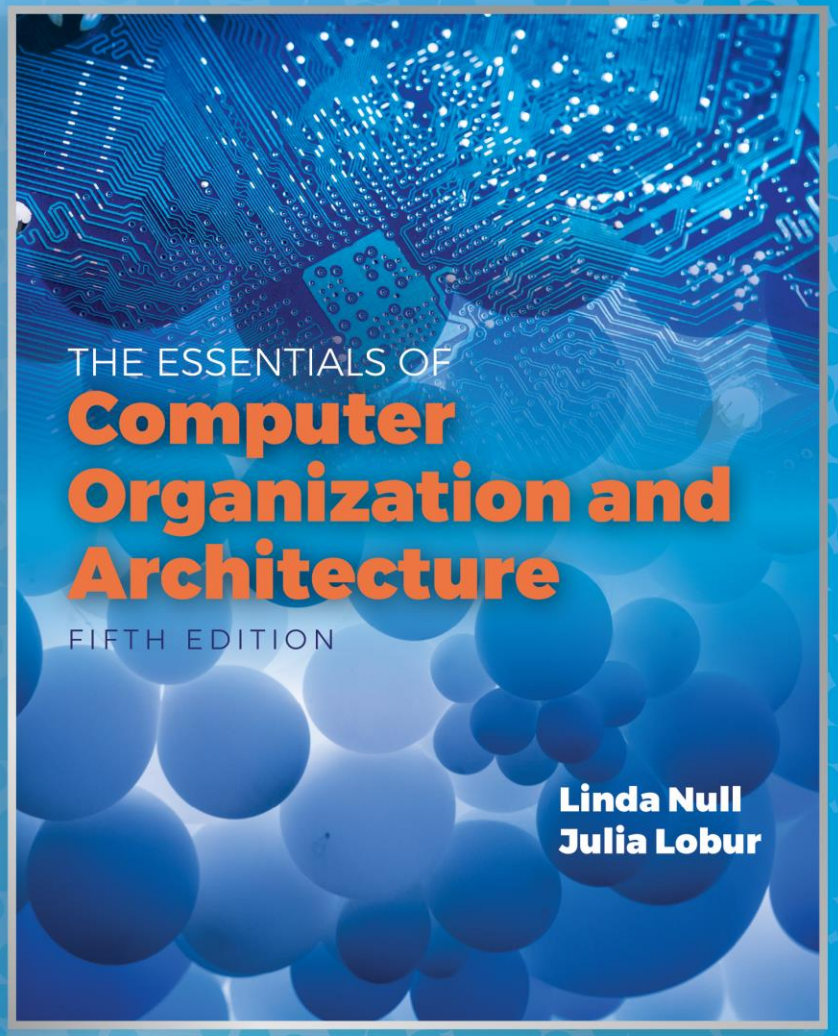
This is the third lecture of Chapter 5

Chapter 5

A Closer Look at Instruction Set Architectures



Quick review of last lecture

- Evaluate expression using different instruction formats
- Expanding opcodes
- Instruction type

5.4 Addressing (1 of 6)

- 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 (2 of 6)

- *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 (3 of 6)

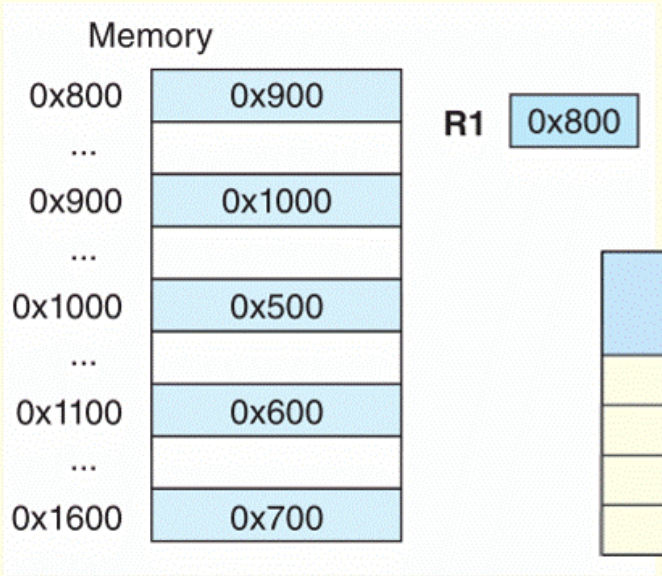
- *Indexed addressing* uses a register (implicitly or explicitly) as an offset, which is added to the address in the operand to determine the effective address of the data.
- *Based addressing* is similar except that a base register is used instead of an index register.
- The difference between these 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 (4 of 6)

- 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.

5.4 Addressing (5 of 6)

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

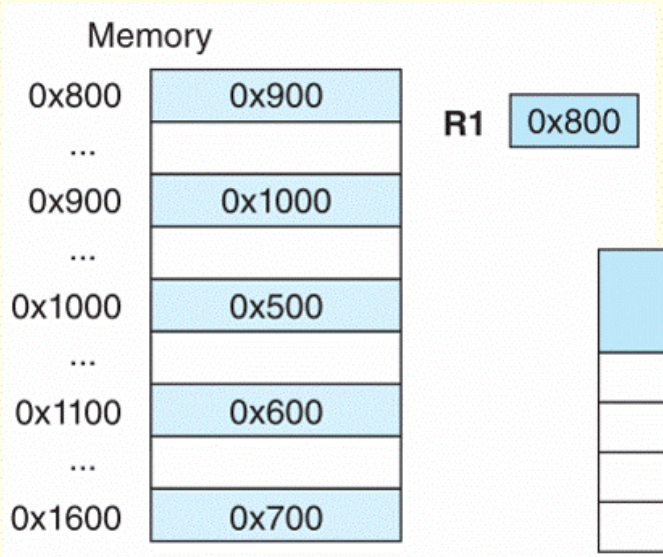


LOAD 800

Mode	Value Loaded into AC
Immediate	
Direct	
Indirect	
Indexed	

5.4 Addressing (6 of 6)

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



LOAD 800

Mode	Value Loaded into AC
Immediate	0x800
Direct	0x900
Indirect	0x1000
Indexed	0x500

5.5 Instruction Pipelining (1 of 7)

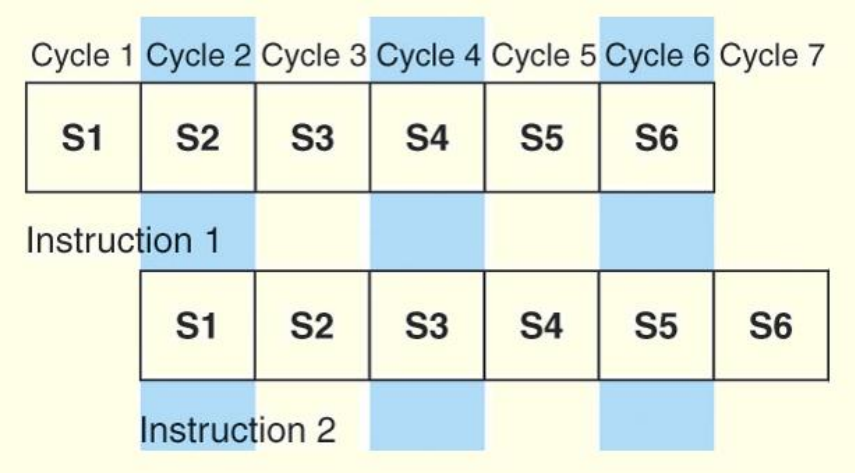
- Some CPUs divide the fetch-decode-execute cycle into smaller steps.
- These smaller steps can often be executed in parallel to increase throughput.
- Such parallel execution is called *instruction pipelining*.
- Instruction pipelining provides for *instruction level parallelism* (ILP)

5.5 Instruction Pipelining (2 of 7)

- 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 (3 of 7)

- 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 (4 of 7)

- 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 (5 of 7)

- 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:

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

- 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 (6 of 7)

- 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 (7 of 7)

- 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.

5.6 Real-World Examples of ISAs (1 of 10)

- 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 5-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.