Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

Instruction Set Architecture

Schedule

- **Today**
  - Closer look at instruction sets
- **Thursday**
  - Brief discussion of real ISAs
  - Quiz 4 (over Chapter 5, i.e. HW #10 and HW #11)
    - Endianness?
    - Infix vs postfix notation?
    - Instructions / expanding opcodes?
    - Addressing modes?
    - Basic pipelines?
    - RISC vs CISC?

Problem 5.2 – Endianness

- 32-bit number 0x456789A1 starting at address 0x10
  - **How is this saved in memory on a big endian system? On a little endian system?**

Address	Big-Endian	Little-Endian
0x10	45	A1
0x11	67	89
0x12	89	67
0x13	A1	45

One byte (8 bits) per location!

Related Problem

Addr	Value
0x10	45
0x11	67
0x12	89
0x13	A1

- **If the data starting at address 10 is interpreted on a little-endian system as an IEEE 754 single-precision value, what is the decimal value?**
- Read off number in correct order (0xA1896745) and convert to binary:
  - 1010 0001 1000 1001 0110 0111 0100 0101
- Interpret:
  - Sign: 1 (negative)
  - Exp: 01000011 (67 -127 = -60)
  - Significand: **1.**00010010110011101000101
- Result: **-1.00010010110011101000101 x 2^{-60}**

Problem 5.9(c) – Infix to Postfix

- **Convert from infix to postfix (RPN) notation:**
  5 × (4 + 3) × 2 - 6
- 5 × (4 3 +) × 2 - 6
- (5 4 3 + ×) × 2 - 6
- 5 4 3 + × 2 × - 6
- **5 4 3 + × 2 × 6 -**

Problem 5.11(c) – Postfix to Infix

- **Convert from postfix to infix notation:**
  3 5 7 + 2 1 – × 1 + +
  - Use a stack!

5.15 – Expanding Opcodes

- Example computer:
  - 11 bit long instructions
  - 4-bit long address fields
- **Can we fit the following instructions into the specified instruction format?**
  - 5 2-address instructions
  - 45 1-address instructions
  - 32 0-address instructions
- *Let's look at the raw bits and see…*

Instruction Types

Instruction types

- **7 broad categories** of processor instructions:
  - Data movement
  - Arithmetic
  - Boolean
  - Bit manipulation
  - I/O
  - Control transfer
  - Special purpose

Take 3 minutes and brainstorm examples of each

Instruction Types – Data Movement

- **Data movement**
  - Moves data between memory, registers, or both
- Examples
  - MARIE instructions: LOAD X and STORE X
  - PUSH and POP instructions
  - EXCHANGE: swap two values
  - May be different instructions for different sizes or types of data (LOADINT and LOADFLT)

Instruction Types - Arithmetic

- **Arithmetic**
  - Operations which involve the ALU to perform a calculation
- Examples
  - MARIE instructions: ADD X, SUBT X, ADDI X
  - MULTIPLY and DIVIDE
  - INCREMENT and DECREMENT: add or subtract 1 from a value
  - NEGATE: unary minus
  - Integer and floating point instructions
  - Some instruction sets even include scientific operations (SINE, SQRT, etc)

Instruction Types – Boolean

- **Boolean**
  - Logical operations on groups of bits
- Examples
  - AND X
    - Performs "bit-wise" operations

ACC	0	1	1	0	1	1	0	0
X	1	1	1	1	0	0	0	0
ACC	0	1	1	0	0	0	0	0

  - OR, NOT, XOR, COMPARE instructions

Instruction Types – Bit Manipulation

- **Bit manipulation**
  - Non-Boolean operations on bits
- Examples
  - ROTATE and SHIFT instructions
- ROTATE moves all bits left or right, and bits which are "shoved out" one side get "shoved in" the other
  - Example: ROTATEL 3 / rotate left 3 bits

ACC	0	1	0	0	0	0	1	1
ACC	0	0	0	1	1	0	1	0

Instruction Types – Bit Manipulation

- SHIFT moves all bits left or right, and bits which are "shoved out" are discarded
- For left shifts, 0's are shifted in
- For right shifts, the bits shifted in depends on whether the shift is logical or arithmetic
  - Logical: Shift in 0's
  - Arithmetic: Copy the leftmost bit (sign bit)
    - Thus, a negative number stays negative!

Instruction Types – I/O

- **Input / Output**
  - Transfer data from system to/from external devices
- Examples
  - MARIE instructions: INPUT and OUTPUT
  - Some processors have no special I/O instruction and instead use memory-mapped I/O, treating I/O devices like "special" memory

Instruction Types – Control Transfer

- **Control transfer**
  - Alter the normal sequence of program execution
- Examples
  - MARIE's JUMP, JUMPI, JNS, SKIPCOND, and HALT
  - Other processors have instructions like
    - BEQ/BNE (branch equal/not equal)
    - DJNZ (decrement and jump if not zero)
    - CJNE (compare and jump if not equal)

Instruction Types – Special Purpose

- Special purpose
  - Just about everything not covered above
  - These can provide access to special hardware specific to the CPU
    - Intel's SSE (Streaming SIMD Extensions) and AMD's 3DNow! instructions for multimedia applications
    - String manipulation instructions

Instruction Types

- One goal of instruction set design is *orthogonality*
- The instructions should be
  - **Unique** - not duplicating the function of any other instruction
  - **Consistent** - (for example, the type of operands should not depend on the type of instruction)
- Hard to implement perfectly in practice!
  - Different engineers make difference decisions on best ISA practices

50-Word Problem from HW #10

- **Describe the key design traits that classify a computer processor as either "CISC" or "RISC" design and state which part of the CPU performance equation each design attempts to optimize**

Addressing Modes

Addressing Modes

- Addressing modes specify where an operand is located
- Choices?
  - Constant?
  - Register?
  - Memory location?
- The actual location of an operand is called its **effective address**
- Certain addressing modes allow us to determine the address of an operand dynamically

Addressing Modes

- **Immediate** addressing
  - The data is part of the instruction
  - Example: ADD 1 *(where 1 is data, not an address)*
- **Direct** addressing
  - The address of the data is given in the instruction
  - Example: ADD ONE *(where "ONE" is a label)*
- **Register** addressing
  - The number / name of the register that holds the data is given in the instruction
  - Example: ADD R1

Addressing Modes

- **Indirect** addressing
  - The address of the address of the data is given in the instruction
  - Example: ADDI POINTER
- **Register indirect** addressing
  - A register stores the address of the address of the data
  - Example: ADDI R1

Addressing Modes

- **Indexed** addressing
  - Instruction names two things: **index register** (might be implicit) and an address
    - Index Register holds an offset number (the “index number”)
    - Address is a base address
  - Effective address of data = base + offset
  - Example: ADD 4(R1)

- **Based** addressing
  - **Same idea, but fields are reversed!**
  - Instruction names two things: **base register** and a displacement address
    - Base register holds the base address
    - Displacement address is the offset (“index”)
  - Effective address of data = base + offset

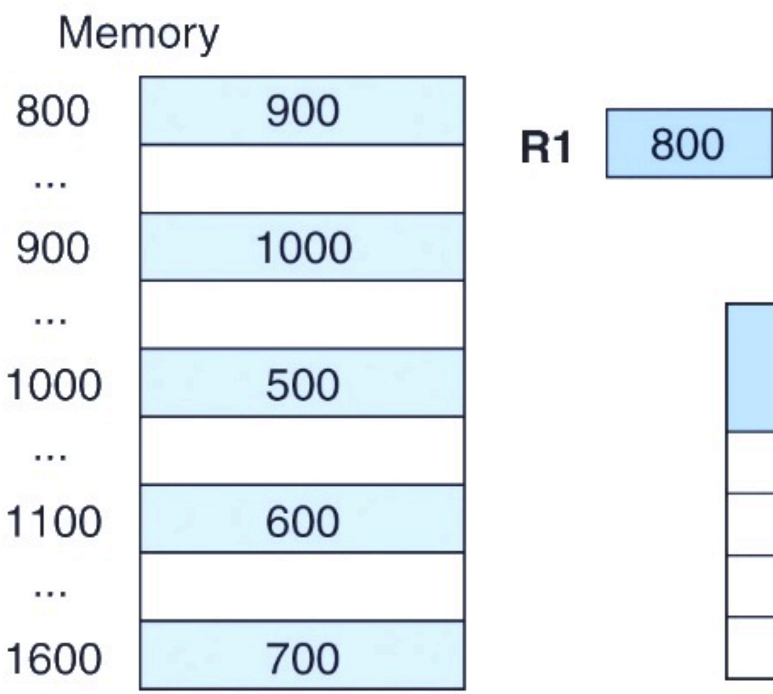
Addressing Modes

- **Stack** addressing
  - Operand is assumed to be on top of the stack
- *(Even more)* variations to these addressing modes!
  - Indirect indexed
  - Self-relative
  - Auto increment / auto decrement
  - *Too much detail for ECPE 170…*

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

Addressing Modes Example

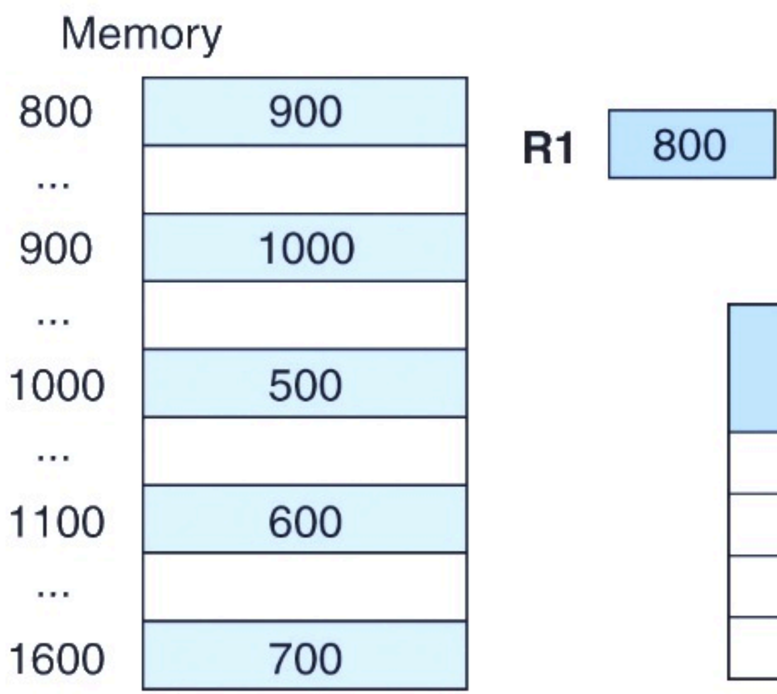
- **For the instruction shown, what value is loaded into the accumulator for each addressing mode?**
  - **Assume R1 is implied for Indexed mode…**




```
LOAD 800
```

Mode	Value Loaded into AC
Immediate	
Direct	
Indirect	
Indexed	

Addressing Modes Example

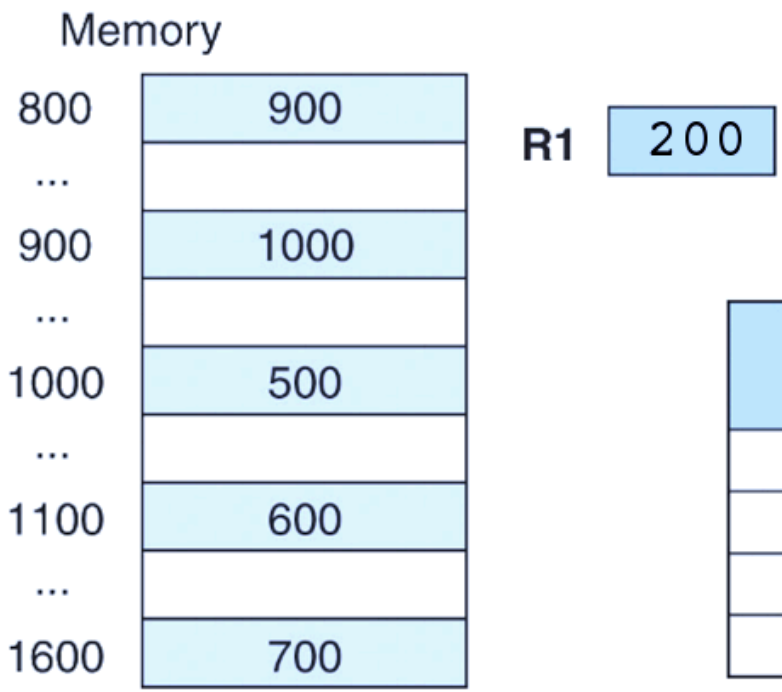


LOAD 800

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

Addressing Modes Exercise

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



LOAD 900

Mode	Value Loaded into AC
Immediate	
Direct	
Indirect	
Indexed	

Instruction Pipelining

Instruction Pipelining

- Some CPUs divide the fetch-decode-execute cycle into smaller steps
  - These steps can often to be executed in parallel to increase processor throughput (i.e. more instructions per cycle!)
- Called **instruction pipelining**
  - Provides for **instruction level parallelism (ILP)**
  - Executing more than one instruction at a time

Instruction Pipelining Example

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

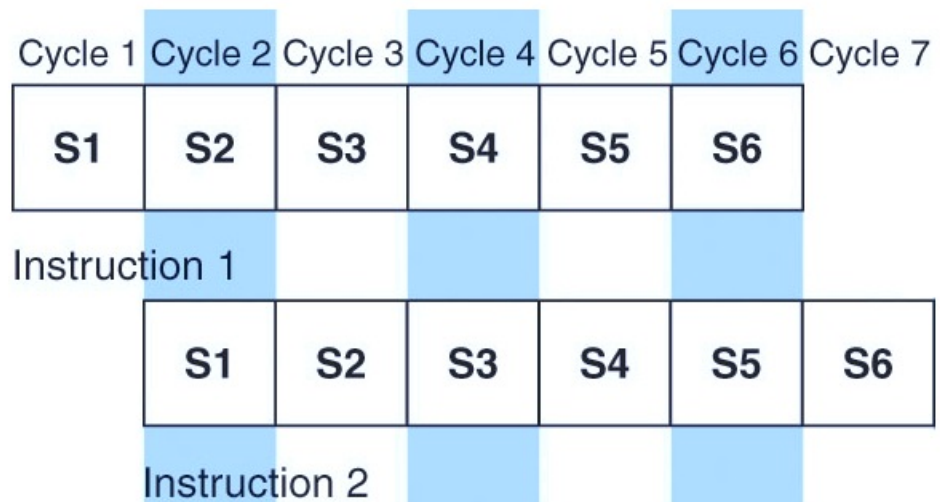
  1. Fetch instruction
  2. Decode opcode
  3. Calculate effective address of operands
  4. Fetch operands
  5. Execute instruction
  6. Store result

- We can implement this cycle with a six-stage pipeline

Instruction Pipelining Example

- 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



Speedup of Instruction Pipelining

- What is the **theoretical speedup** offered by a pipeline?
- 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:
  - $(k \times t_p) + (n - 1)t_p = (k + n - 1)t_p$

Speedup of Instruction Pipelining

- If we take the time required to complete n tasks without a pipeline ($n*t_n$) 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:

$t_n = k * t_p$

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

Speedup of Instruction Pipelining

- Example:
  - Non-pipelined CPU has a clock period t_n = 100ps
  - CPU is redesigned to be pipelined
    - k=5 stages
    - clock period t_p = 20ps
- The theoretical speed-up is 100ps/20ps = 5.
- If we execute n=1,000 sequential tasks (instructions), the actual speed-up is

$$S = \frac{nt_n}{(k+n-1)t_p} = \frac{1000 \times 100\,ps}{(5+1000-1)\times 20\,ps} = \frac{100,000\,ps}{20,080\,ps} = 4.98$$

Speedup of Instruction Pipelining

- **Exercise**
  - Suppose we have a non-pipelined CPU with a clock period t_n of 150ps
  - We redesign the CPU to be a 6 stage pipeline with a clock period t_p of 30ps.
  - **What is theoretical speed-up?**
  - **If we execute n=500 sequential tasks (instructions), what is the actual speed-up?**

Speedup of Instruction Pipelining

- The theoretical speed-up is 150ps/30ps = 5.
- If we execute n=500 sequential tasks (instructions), the actual speed-up is

$$\frac{500 \times 150\,ps}{(6 + 500 - 1) \times 30\,ps} = \frac{75{,}000\,ps}{15{,}150\,ps} = 4.950495\ldots$$

Instruction-Level Pipelining

- **Real life is not as perfect as these examples would indicate!**
- We made a huge assumption here: $t_n = k * t_p$
- If this is true, then the pipeline is perfectly balanced
  - The hardware in every stage takes the exact same amount of time to operate
- Most pipelines are not balanced
  - Some stage takes longer to operate than others
    - Example: getting data from memory is slower than decoding the opcode
  - When the pipeline is not balanced, t_p is determined by the **slowest stage**
  - If $t_n < k * t_p$, the speedup of a k-stage pipeline cannot be k

Instruction-Level Pipelining

- **Real life is even worse – there are more problems than simply having some stages be slower than others!**
- The architecture may not support fetching instructions and data in parallel
  - Need separate memories
  - More hardware = more $$

Instruction-Level Pipelining

- We might not always be able to keep the pipeline full of instructions
  - **Hazards** cause pipeline conflicts and stalls
- Example hazards
  - Data hazards (dependencies)
  - Structural hazards (resource conflicts)
  - Control hazards (conditional branching)
- **Your 50-word problem for HW #12 asks you to explain these hazards**
  - **70**-word limit for this one!

Instruction-Level Pipelining

- Hazards can cause pipeline to **stall** or **flush**
  - **Stall** – pipeline is delayed for a cycle
  - **Flush** – all instructions in pipeline are deleted
- Clever hardware or clever assembly programmers (or *optimizing* compilers) can reduce the effects of these hazards
  - But not fully eliminate them…