Microprocessors & Interfacing

Lecturer : Annie Guo

Lecture Overview

- Course Introduction
  - A whole picture of the course
- Basics of Computing with Microprocessor Systems

Course Organization

- Lecture:
  - Microprocessor fundamentals (1.5 week)
  - Assembly programming (3 weeks)
  - I/O devices and Interfacing (5 weeks)
  - Development and extended topics on microprocessors application (2 weeks)
  - Possibly, advanced topics
- Lab:
  - Four lab exercises
    - Start in week 2.
    - Set up the simulation environment at home and form lab groups (two students per group) before week 2.
- Project design:
  - Microprocessor application

Aims of the Course

- After completing the course, you should
  - Understand the basic concepts and structures of microprocessors, and its operational principles
  - Gain assembly programming skills
  - Understand how hardware and software interact with each other
  - Know how to use microprocessors to solve problems
  - Be familiar with the development of microprocessor applications

Expectation (1)

- Lectures
  - Concepts
  - Principles
  - Problem solving approaches and techniques

Expectation (2)

- Labs
  - Lab tools
    - AVR studio development environment
      - Development, simulation and debug
    - AVR lab board
      - Devices, ports, and connections
      - Programming and testing
  - Lab exercises
    - Prepare before lab
    - Finish in lab
    - Marked off by the lab tutor
      - Late penalties
        - » **20%** off for one-week late
        - » Late more than one week, your work is only marked as completion for eligibility of passing this course.

Expectation (3)

- Homework
  - Study questions provided after each lecture
    - attempt all questions

Assessment

- Four lab exercises must be completed and marked off
  - 20%, working in pairs but marked individually
- Mid-term exam (class test)
  - 20%
- Project design
  - 15%, working individually
- Final exam
  - 45%
- To pass the course,
  - (result >=50)&(lab compl.)&(final_exam>=40)

And …

- Main references:
  - **Fredrick M. Cady: Microcontrollers and Microcomputers —Principles of Software and Hardware Engineering**
  - **AVR documents (available on course website)**
    - Data Sheet
    - Instruction Set
  - Additional materials provided on the course website
- Lecture notes
  - Posted each week before lecture

Resources for help

- Course website
  - www.cse.unsw.edu.au/~cs9032
- Lecturer
  - Lecture break
  - Consultation
    - Thur. 14:00—16:00
- Lab tutors

NOTE

- From time to time I will send announcements to the class through *comp9032-list*,
- Please make sure your **email account** is working and check it frequently.

Microprocessors & Interfacing

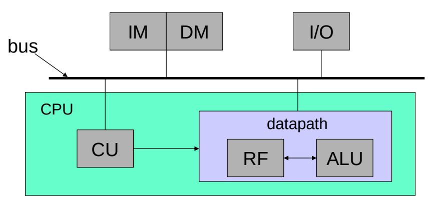
Basics of Computing with Microprocessor Systems

Lecturer: Annie Guo

Lecture Overview

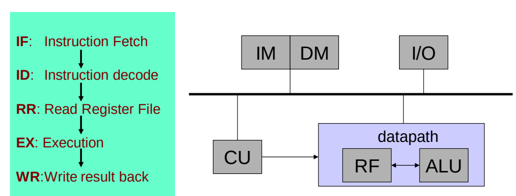
- Microprocessor Hardware Structures
- Data Representation
  - Number representation
- Instruction Set Architecture

Fundamental Hardware Components in Computing System



- **ALU**: Arithmetic and Logic Unit
- **RF**: Register File (a set of registers)
- **CU**: Control Unit
- **IM/DM**: Instruction/Data Memory
- **I/O**: Input/Output Devices

Execution Cycle



Note: Steps can be merged/broken down/expanded

Microprocessors

- A *microprocessor* is the datapath and control unit on a single chip.
- If a microprocessor, its associated support circuitry, memory and peripheral I/O components are implemented on a single chip, it is a *microcontroller*.
  - We use AVR microcontroller as the example in our course

Data Representation

- For a digital microprocessor system being able to compute and process data, the data must be properly represented
  - How to represent numbers for calculation?
    - Binary
  - How to represent characters, symbols and other values for processing?
    - Will be covered later

Decimal

- Example

$$\boxed{\begin{array}{l}(3597)_{10} \\ = 3\times10^3 + 5\times10^2 + 9\times10 + 7\end{array}}$$

  - The place values, from right to left, are 1, 10, 100, 1000
  - The base or radix is 10
  - All digits must be less than the base, namely, 0~9

Number Representation

- Any number can be represented in the form of

$$\boxed{\begin{array}{l}(a_n a_{n-1} \ldots a_1 a_0 . a_{-1} \ldots a_{-m})_r \\ = a_n \times r^n + a_{n-1} \times r^{n-1} + \ldots + a_1 \times r + a_0 + a_{-1} \times r^{-1} + \ldots + a_{-m} \times r^{-m} \\ \\ r : \text{radix,base} \\ 0 \le a_i < r\end{array}}$$

Binary

- Example

$$\boxed{\begin{array}{l}(1011)_2 \\ = 1\times2^3 + 0\times2^2 + 1\times2 + 1\end{array}}$$

  - All digits must be less than 2 (0~1).

Hexadecimal

- Example

$$\boxed{\begin{array}{l}(F24B)_{16} \\ = F\times16^3 + 2\times16^2 + 4\times16 + B \\ = 15\times16^3 + 2\times16^2 + 4\times16 + 11\end{array}}$$

  - All digits must be less than 16 (0~9,**A,B,C,D,E,F)**

Binary Arithmetic Operations

- Similar to decimal calculations
- Examples of addition and multiplication are given in the next two slides.

Binary Additions

- Example:
  - Addition of two 4-bit **unsigned** binary numbers. How many bits are required for holding the result?

1001+0110 = (__________)

Binary Multiplications

- Example:
  - Multiplication of two 4-bit unsigned binary numbers. How many bits are required for holding the result?

1001*0110 = (_____________________)

Binary Subtraction

- Subtraction can be defined as addition of the additive inverse:

  $a - b = a + (-b)$
- We can represent $-b$ by **two's complement** of b.
- In n-bit binary arithmetic, 2's complement of b is

  $$b^* = 2^n - b$$
  - $(b^*)^* = b$
  - The **MSB** (Most Significant Bit) of a 2's complement number is the <u>sign bit</u>
    - For example, for a 4-bit 2's complement number
    - (1001) → -7, (0111) → 7

Exercises

- Represent the following decimal numbers using 8-bit 2's complement format
  - (a) 7
  - (b) 127
  - (c) -12
- Can all the above numbers be represented by 4 bits?
- An n-bit binary number can be interpreted in two different ways: signed or unsigned. What decimal value does the 4-bit number, 1011, represent?
  - (a) if it is a signed number
  - (b) if it is an unsigned number

Examples

4-bit 2's-complement additions/subtractions

(1) 0101 + 0010 (5 + 2):

```
  0101
+ 0010
= 00111
```

(2) 0101 - 0010 (5 - 2):

```
  0101
+ 1110 (= 0010*)
= 10011
```

(3) 0010 - 0101 (2 - 5):

```
  0010
+ 1011 (= 0101*)
= 1101 (= 0011*).
```

Result means -3.

(4) -0101 - 0010 (-5 - 2):

```
  1011 (= 0101*)
+ 1110 (= 0010*)
= 11001
```

Result means -7.

Overflow in Two's-Complement

- Overflow happens when the result cannot be represented by the given number of bits.
- Assume a, b are **positive numbers** in the n-bit 2's complement system,
  - For $a+b$
    - If the MSB of $a+b$ is 1, which indicates a negative number; then the addition causes a **positive overflow**.
  - For $-a-b$
    - If the MSB of $-a-b$ is 0, which indicates a positive number; then the addition causes a **negative overflow**.

Exercises

1. Do the following calculations, where all numbers are 4-bit 2's complement numbers. Check whether there is any overflow.

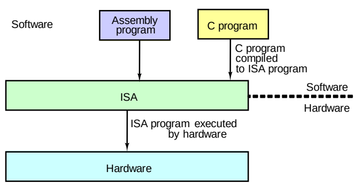
   (a) 1000-0001

   (b) 1000+0101

   (c) 0101+0110

Microprocessor Applications

- A microprocessor application system can be abstracted in a three-level architecture
  - ISA is the interface between hardware and software



Instruction Set

- Instruction set provides the vocabulary and grammar for programmer/software to communicate with the hardware machine.
- It is machine oriented
  - Different machine, different instruction set
    - For example
      - 68K has more comprehensive instruction set than ARM machine
  - Same operation, could be represented differently in different machines
    - AVR
      - Addition: *add r2, r1* ;r2 ← r2+r1
      - Branching: *breq 6* ;branch if equal condition is true
      - Load: *ldi r30, $F0* ;r30 ← F0
    - 68K:
      - Addition: *add d1,d2* ;d2 ← d2+d1
      - Branching: *breq 6* ;branch if equal condition is true
      - Load: mov #1234, d2 ;d2 ← 1234

Instructions

- Instructions can be written in two languages
  - Machine language
    - Made of binary digits
    - Used by machines
  - Assembly language
    - Plain-text representation of machine language
    - Easier to understand than machine language
    - Used by human being.

Machine Code vs. Assembly Code

- Basically, there is a one-to-one mapping between the machine code and assembly code
  - Example (AVR instruction):
    For incrementing register r16 by 1:
    - 1001010100000011 (machine code)
    - inc r16 (assembly code)
- Assembly language also includes **directives**
  - Directives
    - Instructions to the assembler
      - **Assembler** is a program to translate assembly code into machine code.
    - Example:
      - **.def** temp = r16
      - **.include** "m2560def.inc"

Instruction Set Architecture (ISA)

- ISA specifies all aspects of a computer architecture visible to a programmer
  - Instructions (just mentioned)
  - Native data types
  - Registers
  - Memory models
  - Addressing modes

Native Data Types

- Different machines support different data types in hardware
  - e.g. Pentium II:

Data Type	8 bits	16 bits	32 bits	64 bits	128 bits
Signed integer	✓	✓	✓		
Unsigned integer	✓	✓	✓		
BCD integer	✓				
Floating point			✓	✓	

  - e.g. Atmel AVR:

Data Type	8 bits	16 bits	32 bits	64 bits	128 bits
Signed integer	✓				
Unsigned integer	✓				
BCD integer					
Floating point					

Registers

- Two types
  - General purpose
  - Special purpose
    - e.g.
      - Program Counter (PC)
      - Status Register
      - Stack Pointer (SP)
      - Input/Output Registers
  - Stack Pointer and Input/Output Registers will be discussed in detail later.

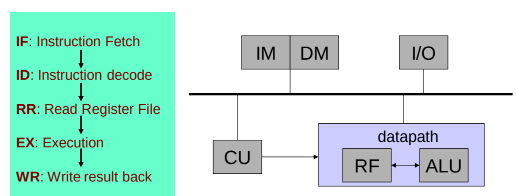
General Purpose Registers

- A set of registers in the machine
  - Used for storing temporary data/results
  - For example
    - In (68K) instruction add d3, d5, *operands are stored in general registers d3 and d5, and the result is stored in d5.*
- Can be structured differently in different machines
  - For example
    - Separate general purpose registers for data and address
      - 68K
    - Different number of registers and different size of registers
      - 32 32-bit registers in MIPS
      - 16 32-bit registers in ARM

Program Counter (PC)

- Special register
  - For storing the memory address of currently executed instruction
- Can be of different size
  - E.g. 16 bit, 32 bit
- Can be auto-incremented
  - By the instruction word size
  - Giving rise the name "counter"

Recall: Execution Cycle



Status Register

- Contains a number of bits with each bit being associated with CPU operations
- Typical status bits
  - V: Overflow
  - C: Carry
  - Z: Zero
  - N: Negative
- Used for controlling the program execution flow

Memory Model

- Deals with how memory is used to store data
- Issues
  - Addressable unit size
  - Address spaces
  - Endianness
  - Alignment

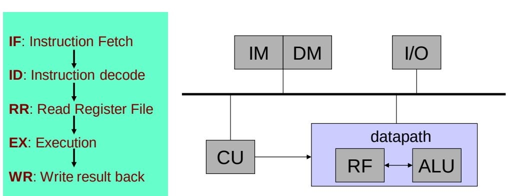
Addressable Unit Size

- Memory has units, each of which has an address
- Most basic unit size is 8 bits (1 byte)
  - Related addresses are called byte-addresses.
- Modern processors can have multiple-byte unit
  - e.g. 32-bit instruction memory in MIPs
    16-bit Instruction memory in AVR
  - Related addresses are called word-addresses.

Address Space

- The range of addresses a processor can access.
  - A processor can have one or more address spaces. For example
    - Princeton architecture or Von Neumann architecture
      - A single linear address space for both instructions and data memory
    - Harvard architecture
      - Separate address spaces for instruction and data memories

Recall: Execution Cycle



Address Space (cont.)

- Address space is not necessarily just for "memory"
  - E.g, all general purpose registers and I/O registers can be accessed through memory addresses in AVR

Endianness

- Memory objects
  - Memory objects are basic entities that can be accessed as a function of the **address** and the **length**
    - E.g. bytes, words, longwords
- For large objects (multiple bytes), there are two byte-ordering conventions
  - **Little endian** – little end (least significant byte) stored first (at lowest address)
    - Intel microprocessors (Pentium etc)
  - **Big endian** – big end (most significant byte) stored first
    - SPARC, Motorola microprocessors

Big Endian & Little Endian

- Example: 0x12345678—a long word of 4 bytes. It is stored in the memory at address 0x00000100
  - big endian:

Address	data
0x00000100	0x12
0x00000101	0x34
0x00000102	0x56
0x00000103	0x78

  - little endian:

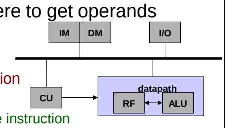
Address	data
0x00000100	0x78
0x00000101	0x56
0x00000102	0x34
0x00000103	0x12

Alignment

- Modern computer reads from or writes to a memory address in fixed sized chunks,
  - for example, word size
- Alignment means putting the data at a memory address equal to some multiple of the word size
  - for example, with AVR, data in the program memory are aligned with the word addresses.

Addressing Modes

- Instructions need to specify where to get operands from
- Some possibilities
  - an operand value is in the instruction
  - an operand value is in a register
    - the register number is given in the instruction
  - an operand value is in memory
    - address is given in instruction
    - address is given in a register
      - the register number is in the instruction
    - address is a register content plus some offset
      - register number is in the instruction
      - offset is in the instruction (or in a register)
- These ways of specifying the operand locations are called **addressing modes**



Addressing Modes (cont.)

- Some examples are given in the next slides, based on the 68K machine.
- For each addressing mode, there are
  - a general description and
  - an example to show how the address mode is used.
    - the specified addressing mode is highlighted in red

Immediate Addressing

- The operand is from the instruction itself
  - i.e the operand is immediately available from the instruction
- For example, in 68K

addw ***#99*, d7**

  - d7 ← 99 + d7; value 99 comes from the instruction
  - d7 is a register

Register Direct Addressing

- Data from a register and the register is directly given by the instruction
- For example, in 68K

addw ***d0*,d7**

  - d7 ← d7 + d0; add value in d0 to value in d7 and store result to d7
  - d0 and d7 are registers

Memory Direct Addressing

- The data is from memory, the memory address is directly given by the instruction
- We use notion: *(addr)* to represent memory value at address, *addr*
- For example, in 68K

addw 0x123A, d7

  - d7 ← d7 + (0x123A); add value in memory location 0x123A to register d7

Memory Register Indirect Addressing

- The data is from memory, the memory address is given by a register, which is directly given by the instruction
- For example, in 68K

```
addw (a0),d7
```

  - d7 ← d7 + (a0); add value in memory with the address stored in register a0, to register d7
    - For example, if a0 = 100 and (100) = 123, then this adds 123 to d7

Memory Register Indirect Auto-increment

- The data is from memory, the memory address is given by a register, which is directly given by the instruction; and the value of the register is automatically increased – to point to the next memory object.
- For example, in 68K

```
addw (a0)+,d7
```

  - d7 ← d7 + (a0); a0 ← a0 + 2

Memory Register Indirect Auto-decrement

- The data is from memory, the memory address is given by a register, which is directly given by the instruction; but the value of the register is automatically decreased before such an operation.
- For example, in 68K

```
addw -(a0),d7
```

  - a0 ← a0 –2; d7 ← d7 + (a0);

Memory Register Indirect with Displacement

- Data is from the memory with the address given by the register plus a constant
  - Used in the access a member in a data structure
- For example, in 68K

```
addw a0@(8), d7
```

  - d7 ← (a0+8) +d7

Address Register Indirect with Index and Displacement

- The address of the data is sum of the initial address and the index address as compared to the initial address.
  - Used in accessing element of an array
- For example, in 68K

```
addw a0@(d3)8, d7
```

  - d7 ← (a0 + d3+8)+d7
  - With a0 as an initial address and d3 varied to dynamically point to different elements plus a constant for a certain member.

Why do we need to know instruction set?

Task: n*16, n is an integer

Processor A
op:
add +,
sub -,
lsh1 (←1-bit)

Processor B
op:
add +,
sub -,
lsh n (←n-bit)

Processor C
op:
add +,
sub -,
lsh n (←n-bit),
mul *,
div /

{ add … add } — 15 additions

{ lsh1 … lsh1 } — 4 shifts

lsh 4

mul

How do native data types affect you as a programmer?

Task: 25+3.5

Processor A
op:
add +
sub -,
lsh1 ←1-bit
rsh1 →1-bit
Data type:
integer

Processor B
op:
add +
sub -,
lsh n ←n-bit
Data type:
integer
real

25 → 25*2/2
3.5 → 3.5*2/2
50 + 7 → 57
57/2 → 28.5

25+3.5 → 28.5

How would available registers affect you as a programmer?

Task: 1023+5

Processor A
register file:
two 8-bit registers

Processor B
register file:
two 16-bit registers

Both machines have instructions:
add Rd Rr
Rd ← Rd+Rr

1023 → r0, r1
5 → ?
Sum → ?

1023 → r0
5 → r1
Sum → r0

Reading Material

- Cady "Microcontrollers and Microprocessors", Chapter 1.1, Chapter 2.2-2.4
- Cady "Microcontrollers and Microprocessors", Appendix A
- Week 1 reference: "number conversion"
  - available at the course website

Homework

Questions 1-6 are in Cady "Microcontrollers and Microprocessors",

1. Question A.4 (i)(ii) (a)(f)
2. Question A.8 (b)(c)
3. Question A.9 (a)(b)
4. Question 2.4
5. Question 3.1 (a)(c)
6. Questions 3.5, 3.7

7. Install AVR Studio at home and complete lab0
   - Available on the course website

Homework

1. Find the two's complement binary code for the following decimal numbers:

(a) 26
(b) -26

Homework

2. Find the binary code words for the following hexadecimal numbers:

(c) C0FFEE
(d) F00D

Homework

3. Prove that the two's-complement overflow cannot occur when two numbers of different signs are added.